MINISTRY OF EDUCATION

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA, ROMANIA
FACULTY OF AUTOMATION AND COMPUTER SCIENCE

Unsupervised Lexical Semantic Frame Induction

LICENSE THESIS

Graduate: **Marian ȘANDOR**

Supervisor: **Conf. Dr. Eng. Emil Ștefan CHIFU**

2022

MINISTRY OF EDUCATION

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA, ROMANIA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE

DEAN,
Prof. dr. ing. Liviu MICLEA

HEAD OF DEPARTAMENT,
Prof. dr. ing. Rodica POTOLEA

Graduate: **Marian ȘANDOR**

Unsupervised Lexical Semantic Frame Induction

1. **Project proposal:** *The goal of the thesis is to develop a system to perform unsupervised lexical semantic frame induction by addressing the tasks proposed by Task 2 SemEval2019 competition*
2. **Project contents:** *Introduction - Project Context, Project Objectives and Specifications, Bibliographic Research, Analysis and Theoretical Foundation, Detailed Design and Implementation, Testing and Validation, User's Manual, Conclusions and Appendices*
3. **Place of documentation:** Technical University of Cluj-Napoca, Computer Science Department
4. **Consultants: Conf. Dr. Eng. Emil Ștefan Chifu**
5. **Data of issue of the proposal:** November 1, 2021
6. **Data of delivery :** July 8, 2022

Graduate: _____

Supervisor: _____

MINISTRY OF EDUCATION

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA, ROMANIA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE

Declarație pe proprie răspundere privind autenticitatea lucrării de licență

Subsemnatul(a) _____,
_____ legitimat(ă) cu
_____ seria _____ nr. _____
CNP _____, autorul lucrării
_____
_____
elaborată în vederea susținerii examenului de finalizare a studiilor de licență la Facultatea de Automatică și Calculatoare, Specializarea _____ din cadrul Universității Tehnice din Cluj-Napoca, sesiunea _____ a anului universitar _____, declar pe proprie răspundere, că această lucrare este rezultatul propriei activități intelectuale, pe baza cercetărilor mele și pe baza informațiilor obținute din surse care au fost citate, în textul lucrării și în bibliografie.

Declar, că această lucrare nu conține porțiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislației române și a convențiilor internaționale privind drepturile de autor.

Declar, de asemenea, că această lucrare nu a mai fost prezentată în fața unei alte comisii de examen de licență.

În cazul constatării ulterioare a unor declarații false, voi suporta sancțiunile administrative, respectiv, *anularea examenului de licență.*

Data _____

Nume, Prenume _____

Semnătura

Contents

Chapter 1. Introduction - Project Context

Natural language processing is a subfield of computer science and artificial intelligence which studies the interactions between computers and human language. The idea is to program computers to process and analyze natural language data, and by doing so, provide them the ability of "understanding" the contents of documents.

The task faces a few challenges which slow down the development process and make the desired results hard to be achieved. One of the challenges is that human language is highly ambiguous as people are great at producing language and understanding it and are capable of expressing very elaborate meanings. Another one would be the fact that languages are continuously changing and evolving.

Semantic role labeling is the process in which words or phrases are labeled such that to indicate their semantic role in the sentence. Its goal is to find the meaning of a sentence. The way it works is by detecting the arguments associated with the predicate or verb of a sentence and assign them specific roles. By performing semantic role labeling, the machine would be able to tell that the sentences "John sold a car to Mary" and "Mary bought a car from John" essentially describe the same situation.

1.1. Project Context

As stated above, semantic role labeling seeks to assign labels to the arguments associated with the predicate or verb of a sentence, but the actual labels and classification is dictated by the lexicon used. Two of the most used lexicons are VerbNet and FrameNet where the former defines more general roles while the latter aims for a fine grained classification. In both cases the verbs play an important role.

VerbNet is using Levin verb classes to systematically construct lexical entries. Classes are hierarchically organized to ensure that all their members have common semantic and syntactic properties. Each class in the hierarchy is characterized extensionally by its set of verbs, and intensionally by syntactic frames and semantic predicates and a list of typical verb arguments. An example of an annotated sentence following the VerbNet rules can be seen in Figure 1.1.

FrameNet is the computational application of the theory of Frame Semantics. The knowledge about the semantics of lexical items is modeled against the background semantic frames which they evoke. A frame is a sort of scene, a set of concepts related to each other in such a way that the presence of one of them makes all the other concepts readily available. The frames are also connected to each other by several types of formally defined relations, so they can be thought of as a network of nodes linked by different types of arcs. An example of an annotated sentence following the FrameNet rules can be seen in Figure 1.2.

[1] taken from `http://www.forum.santini.se/2016/01/1852/`

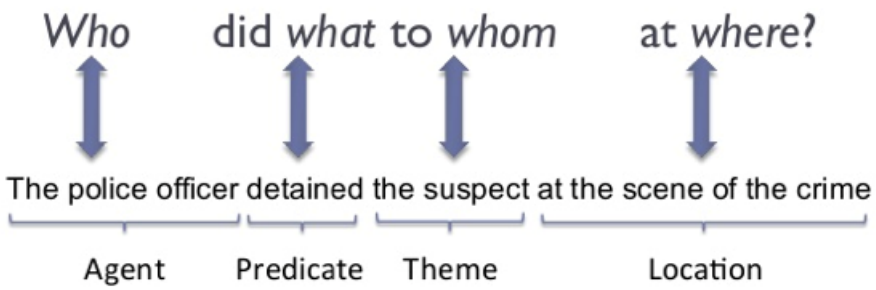


Figure 1.1: VerbNet Annotated Sentence Example [1]

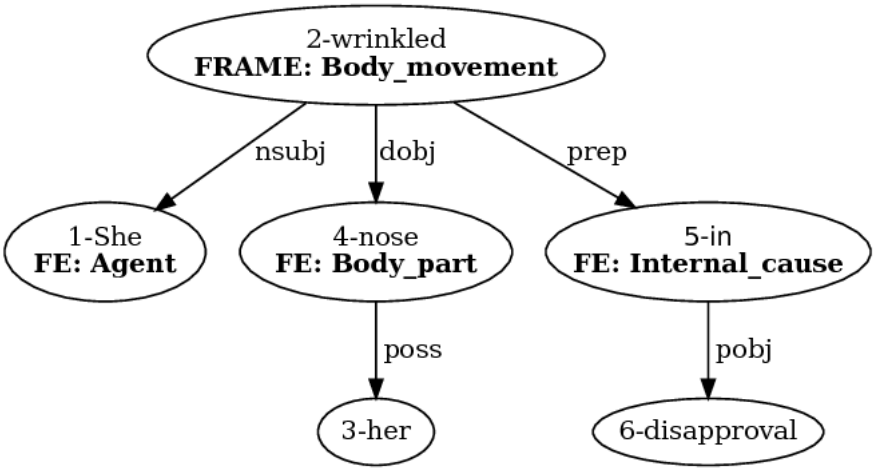


Figure 1.2: FrameNet Annotated Sentence Example [2]

Lexical frame resources have proved to be helpful in many NLP tasks such as question answering, information extraction, automatic text summarization and speech recognition. The problem is that building them for new languages and domains is resource intensive and thus expensive. Task 2 from SemEval 2019 [1] addresses this problem and suggests using unsupervised lexical frame induction methods. Lexical frame induction is the process of grouping verbs and the associated words in type-feature structures in a fully unsupervised manner.

1.2. Thesis Structure

The thesis is structured in 8 chapters and for the remaining chapters, a brief description of their contents is given below.

- **Chapter 2** Provides a more detailed view of the problem at hand. The main objectives of the task are presented.
- **Chapter 3** Presents the idea of machine learning and deep learning emphasizing on the unsupervised learning methods and the process of clustering the data. In the end

[2] taken from `https://www.researchgate.net/figure/Example-FrameNet-annotation-FEs-and-FEEs-are-annota fig1_268290484`

we will go over the word sense induction problem which has a lot in common with our task at hand.

- **Chapter 4** We present our approach for solving the problem by explaining from a theoretical point of view the operating principles of the implemented system.

- **Chapter 5** Takes a deeper dive into the implementation of the system taking a closer look to how the data is processed in order to obtained the final results.

- **Chapter 6** Describes the methods used for testing and the metrics based on which the system output is validated.

- **Chapter 7** Offers the necessary guidelines for the system to be run on the user machine. Hardware and software resources needed are stated along with the installation steps.

- **Chapter 8** Focuses on the personal contributions and achievements whilst making a critical analysis of the results. Finally, we take a look at the possibilities of improvements and further development.

Chapter 2. Project Objectives

2.1. Project Theme

The project represents our approach to the problem raised by SemEval 2019 Task 2 [1]. As stated before, the human language is highly ambiguous and this impacts badly the machines abilities to understand text. A systematic approach in which words from text are considered individually from one another would yield very poor results. Words are dependent on the context they appear in and thus it is crucial to consider not just the word but its context too when looking for understanding its meaning.

Lexical frames address this problem by creating a higher level view of the words and their context by extracting the meaning. Words, usually verbs, along with their arguments (which constitute the context) are grouped in different lexical frames which later results in a hierarchical structure compressing the relations between them. To understand better the idea, Figure 2.2 shows the Abandonment lexical frame. It is worth noting that this frame is evoked not just by the word "abandonment" but also by the following: "abandon", "abandoned", "forget", "leave".

Building a database for the lexical frames of a language is not an easy task. The process is resource intensive and having it done manually by humans can take up to years. For this reason, Task 2 from SemEval 2019 proposed the development of a system to perform lexical frame induction. In other words, given tokenized and morphosyntacally labeled text corpora the system should cluster sentences evoking different frame semantic structures in a totally unsupervised manner.

2.2. Objectives

To achieve the goal mentioned above, the problem is divided into two separate tasks. The first one (Task A) asks for grouping verbs to frame type clusters while the second task (Task B) focuses on clustering the arguments of verbs to frame-specific slots (Task B.1) or to generic roles (Task B.2).

2.2.1. Task A

Occurrences of verbs are grouped into number of clusters in such a way that verbs belonging to the same cluster evoke the same frame type. In other words, the goal is labeling verb uses in context to resemble their categorization based on Frame Semantics (Figure 2.1).

Given the sentences:
a. Trump leads the world, backward.
b. Disrespecting international laws leads to many complications.
c. Rosenzweig heads the climate impacts section at NASA's Goddard Institute.

The verbs ”to lead” from a. and ”to head” from c. should be part of the same cluster (Leadership after FrameNet) while ”to lead” from b. will be part of another cluster (Cause) along with other potential verbs such as ”originate” or ”produce”.

2.2.2. Task B.1

For this task, arguments of verbs must be grouped to a number of frame-specific slots similar to FrameNet. That is, we assume argument groupings are specific to frame types and that they are not necessarily shared with other frames. In other words, the aim is the unsupervised labeling of frames and core FEs (Figure 2.1) and because FrameNet defines FEs frame-specifically, Task B.1 entails Task A.

2.2.3. Task B.2

In contrast to Task B.1, here verb arguments are clustered into a set of generic roles that are defined independently of frame definitions. Hence, this Task is very similar to unsupervised semantic role induction. In other words, the objective is to clusterize the arguments of verbs according to generic semantic roles (e.g. Agent, Patient) similar to those present in VerbNet (Figure 2.1).

2.3. Specifications

2.3.1. Functional Requirements

- Data Preprocessing

  Original data comes in a specific format. A module should work with the dataset to extract and model the data. The system will work with the preprocessed data.

- Generate Word Embeddings

  A module will work with text data to generate vector representations of words and sentences.

- Unsupervised Classification

  The classification of the data required to obtain the final results must be performed in an unsupervised manner. The use of a clustering is needed.

- Generate Submission Files

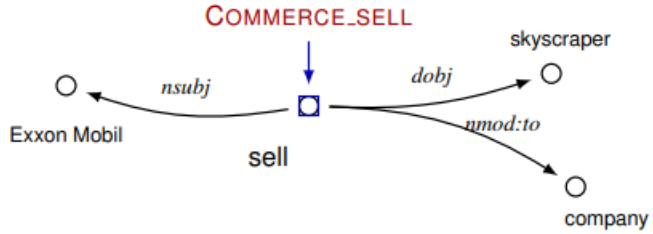
  The scorer program requires the input in a specific format. A module should process the results and generate a submission file with the right format to be given to the scorer program.

2.3.2. Non-Functional Requirements

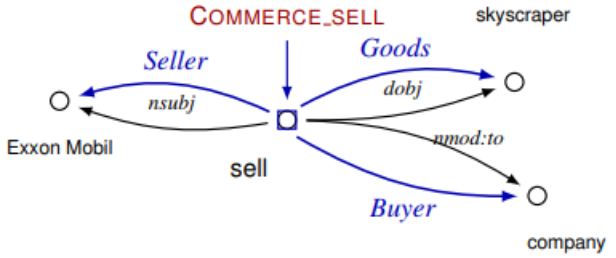
- Usability

  The system can be set-up by running just a few commands. Once that is done, the system can be tested by running the submission scripts and the same results as presented in this document will be obtained.

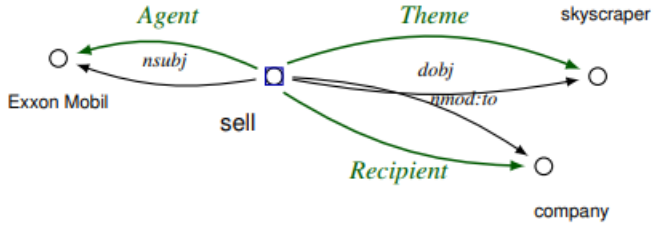
- Compatibility



(a) **Task A** - Identifying Semantic Frames: Unsupervised learned labels evaluated against FN's lexical units



(b) **Task B.1** - Full Frame Semantic Tagging: Unsupervised labels evaluated against FN's frames



(c) **Task B.2** – Case Role Labeling: Unsupervised labels evaluated against generic semantic roles (VerbNet)

Figure 2.1: Subtasks of SemEval 2019 Task 2 [1]

The system should be developed using cross-platform technologies. The system can be set run though a minimal configuration process on any machine regardless of its operating system.

- Scalability

  The embedding generation phase is the most demanding one. It can be set to run on the processor or easily configured to run on a GPU or even multiple GPUs.

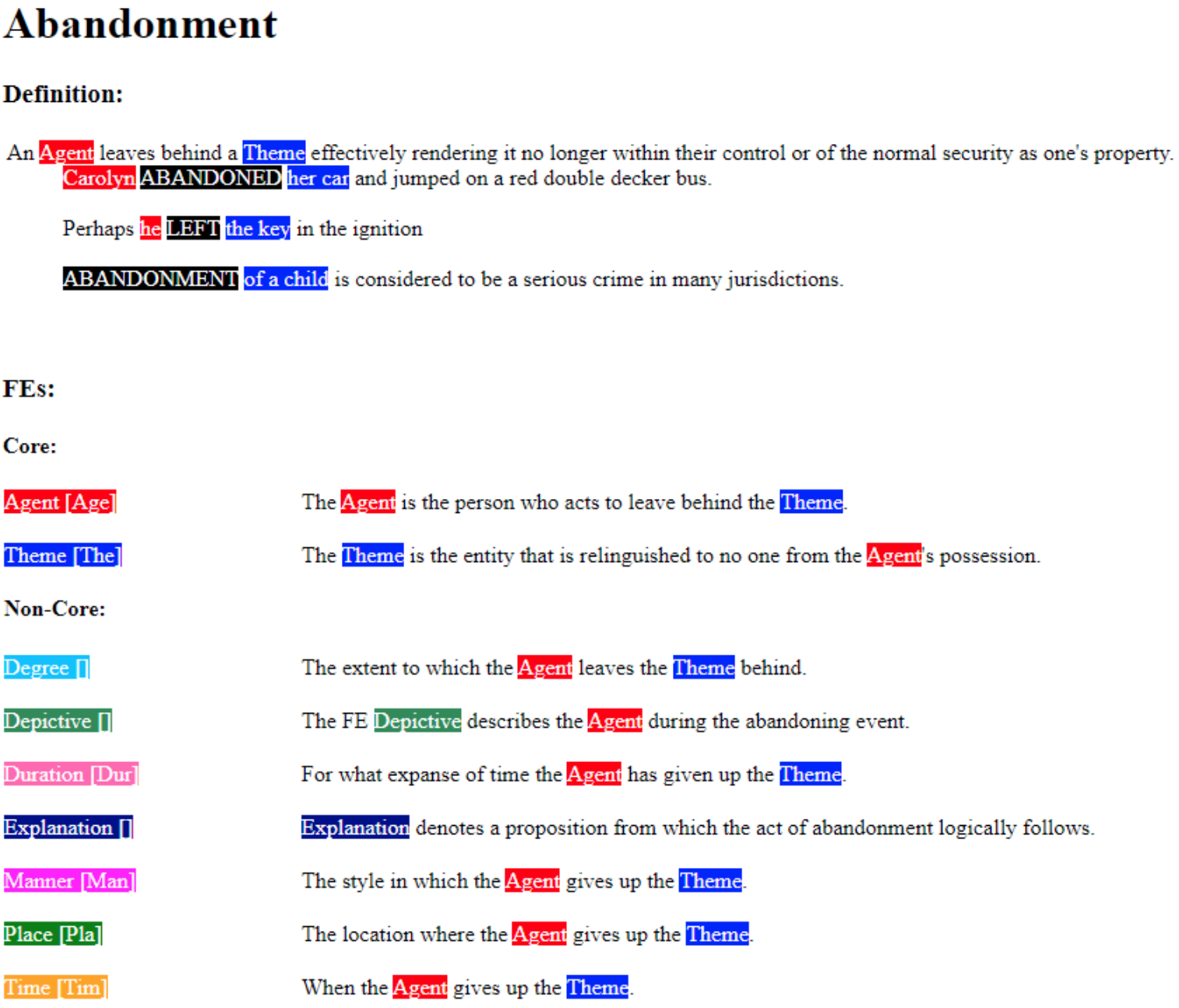
- Performance

  The system should perform better than the baseline models in all of the three tasks. The performance will be measured using the same standard procedure implemented by a scorer program.

- Documentation

  The system is accompanied by a document which covers in great detail the de-

sign and implementation of it. The documentation should provide easy to follow installation steps.



Abandonment

Definition:

An Agent leaves behind a Theme effectively rendering it no longer within their control or of the normal security as one's property.
Carolyn ABANDONED her car and jumped on a red double decker bus.

Perhaps he LEFT the key in the ignition

ABANDONMENT of a child is considered to be a serious crime in many jurisdictions.

FEs:

Core:

Agent [Age]	The Agent is the person who acts to leave behind the Theme.
Theme [The]	The Theme is the entity that is relinguished to no one from the Agent's possession.

Non-Core:

Degree []	The extent to which the Agent leaves the Theme behind.
Depictive []	The FE Depictive describes the Agent during the abandoning event.
Duration [Dur]	For what expanse of time the Agent has given up the Theme.
Explanation []	Explanation denotes a proposition from which the act of abandonment logically follows.
Manner [Man]	The style in which the Agent gives up the Theme.
Place [Pla]	The location where the Agent gives up the Theme.
Time [Tim]	When the Agent gives up the Theme.

Figure 2.2: The Abandonment Lexical Frame [1]

[1] taken from FrameNet webpage `http://sato.fm.senshu-u.ac.jp/frameSQL/fn2_15/notes/`

Chapter 3. Bibliographic Research

3.1. Lexical Resources

3.1.1. VerbNet

VerbNet [2] is a hierarchical domain-independent and broad-coverage verb lexicon having mappings to other lexical resources. In order to achieve syntactic and semantic coherence among members of a class, it is organized into verb classes extending Levin classes through refinement and addition of subclasses. VerbNet groups together verbs with identical sets of syntactic frames and semantic predicate structures. The resulting classes generate a hierarchical structure as they may inherit from a parent class.

3.1.2. FrameNet

FrameNet [3] derives from the work of Charles J. Fillmore and is based on the theory of meaning called frame semantics. The idea is that the meaning of most words can be understood on the basis of a semantic frame which is a description of a type of event, relation or entity among with the participants present in it. Words, generally verbs, evoke a frame and are called lexical units while the arguments associated with the lexical units are called frame elements and are specific to a frame. FrameNet defines more than 1200 semantic frames which are linked together by a system of frame relations creating a hierarchical structure.

3.2. Machine Learning

Machine learning is an area of artificial intelligence (AI) and computer science that focuses on using data and algorithms to mimic the way humans learn, with the goal of steadily improving accuracy. It is a crucial part of the rapidly expanding discipline of data science. Algorithms are taught to generate classifications or predictions using statistical approaches, revealing crucial insights in data mining initiatives. Following that, these insights drive decision-making within applications and enterprises, with the goal of influencing key growth metrics.

While problems in Pattern Recognition and Machine Learning can be of various types, they can be broadly classified into three categories:

- **Supervised Learning** also known as supervised machine learning, is defined by its use of labeled datasets to train algorithms that to classify data or predict outcomes accurately. As input data is fed into the model, it adjusts its weights until the model has been fitted appropriately. This occurs as part of the cross validation process to ensure that the model avoids overfitting or underfitting. Some methods

used in supervised learning include neural networks, naive bayes, linear regression, logistic regression, random forest, support vector machine (SVM), and more.

- **Unsupervised Learning** also known as unsupervised machine learning, uses machine learning algorithms to analyze and cluster unlabeled datasets. These algorithms discover hidden patterns or data groupings without the need for human intervention. Its ability to discover similarities and differences in information make it the ideal solution for exploratory data analysis, cross-selling strategies, customer segmentation, image and pattern recognition. It's also used to reduce the number of features in a model through the process of dimensionality reduction; principal component analysis (PCA) and singular value decomposition (SVD) are two common approaches for this. Other algorithms used in unsupervised learning include neural networks, k-means clustering, probabilistic clustering methods, and more (in this category fall the language models ELMo [4], BERT [5] and Word2Vec [6] over which we will go in great detail in Chapter 4).

- **Semi-supervised Learning** offers a happy medium between supervised and unsupervised learning. During training, it uses a smaller labeled data set to guide classification and feature extraction from a larger, unlabeled data set. Semi-supervised learning can solve the problem of having not enough labeled data (or not being able to afford to label enough data) to train a supervised learning algorithm.

3.2.1. Deep Learning

Because deep learning and machine learning are often used interchangeably, it's important to understand the differences. Artificial intelligence includes subfields such as machine learning, deep learning, and neural networks. Deep learning, on the other hand, is a branch of machine learning, and neural networks is a branch of deep learning.

Labeled datasets can be used to inform "deep" machine learning algorithms, also known as supervised learning, but they aren't always required. It can consume unstructured data in its raw form (e.g., text, photos) and derive the set of characteristics that separate distinct categories of data from one another automatically. It does not require human interaction to interpret data, unlike machine learning, allowing us to scale machine learning in more interesting ways. Deep learning and neural networks are credited for speeding up development in fields including computer vision, natural language processing, and speech recognition. A good visual representation of the classification processes followed by both models is presented in Figure 3.1. Here, the need of a software engineer during the feature extraction phase is highlighted in care of the machine learning algorithms.

3.2.2. Unsupervised Learning

The goal of unsupervised learning problems could be of discovering groups of similar examples within data, case in which it is called clustering, or of determining the distribution of data in the space, known as density estimation. But to put it simpler terms, "for a n-sampled space x1 to xn, true class labels are not provided for each sample, hence known as learning without teacher" [8].

Given by the constraints imposed by the datasets, unsupervised learning is considered to be harder compared to supervised learning tasks. Since there are no labels

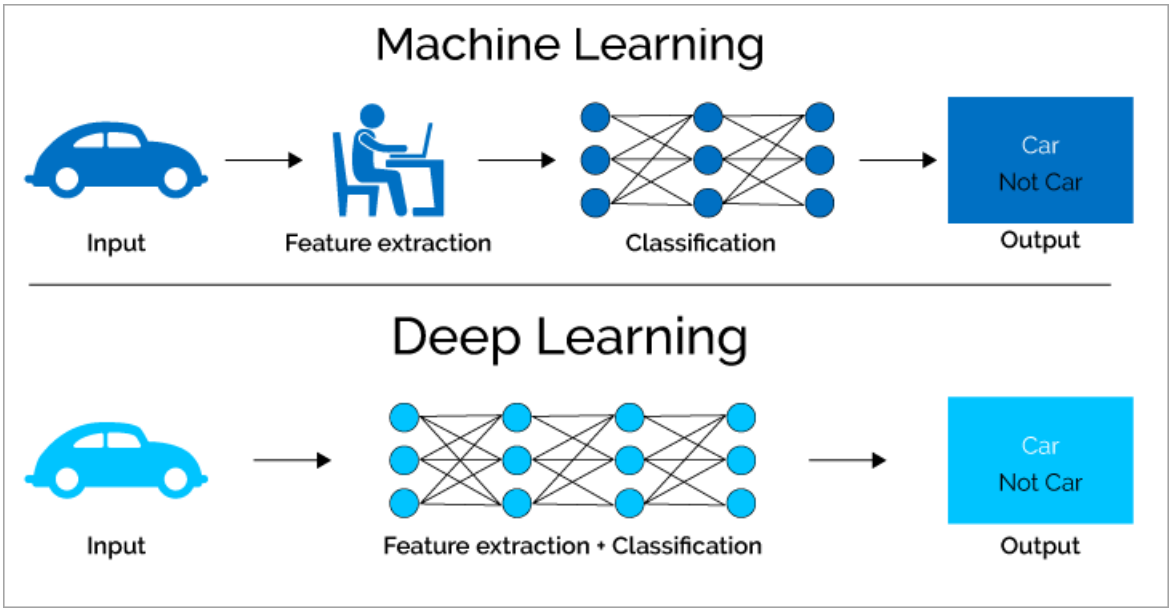


Figure 3.1: Machine Learning vs Deep Learning [7]

available, obtaining the results is not entirely the final step. Results have to be examined, usually by experts, with the goal in mind of understating the meaning of them.

However, despite of these issues, unsupervised learning is need because annotating large datasets is very costly and hence only a few examples can be labeled manually. There is also the case when we don't know in how many classes is the data divided into or we may want to use clustering algorithms to gain insights about the structure of the data.

Going deeper into the field of unsupervised learning, it can be further classified into two categories:

- **Parametric Supervised Learning** case in which a parametric distribution of the data is assumed. It assumes that sample data comes from a population that follows a probability distribution based on a fixed set of parameters. Theoretically, in a normal family of distributions, all members have the same shape and are parameterized by mean and standard deviation. That means if you know the mean and standard deviation, and that the distribution is normal, you know the probability of any future observation. Parametric Unsupervised Learning involves construction of Gaussian Mixture Models and using Expectation-Maximization algorithm to predict the class of the sample in question.

- **Non-parametric Unsupervised Learning** case in which the data is grouped into clusters, where each cluster (hopefully) says something about categories and classes present in the data. This method is commonly used to model and analyze data with small sample sizes.

3.3. Clustering

Clustering deals with finding a structure in a collection of unlabeled data and could be considered as the most important unsupervised learning problem. In other words, clustering is the process of organizing objects into groups whose members are

similar in some way. Therefore, one can say that a cluster is a collection of objects which are similar between them and are dissimilar to the objects belonging to other clusters (Figure 3.2).

The goal of clustering is to determine the internal grouping in a set of unlabeled data. But the problem is given by how do we define the correctness of the clusters. The answer is that the user should supply the criterion such that the result of clustering will suit his needs. In practice, this criterion refers to one of the proximity measures.

3.3.1. Proximity Measures

For clustering, we need to define a proximity measure for two data points. Proximity here means how similar/dissimilar the samples are with respect to each other. We can consider the similarity measure S(xi, xk) which is large in case xi, xk are similar and dissimilarity measure (usually refered to as distance) D(xi, xk) which we expect to be small if xi, xk are similar (Figure 3.3).

There are multiple similarity measures used in practice from which we identify:

- **Vectors: Cosine Distance** defined as the cosine of the angle between two samples (i.e. the dot product) Eq 3.1.
- **Sets: Jaccard Distance** defined as the size of the intersection divided by the size of the union of the sample sets Eq 3.2.
- **Points: Euclidean Distance** defined as the length of a line segment between the two points Eq 3.3.

In general a good proximity measure is very application dependent. The clusters should be invariant under the transformations "natural" to the problem.

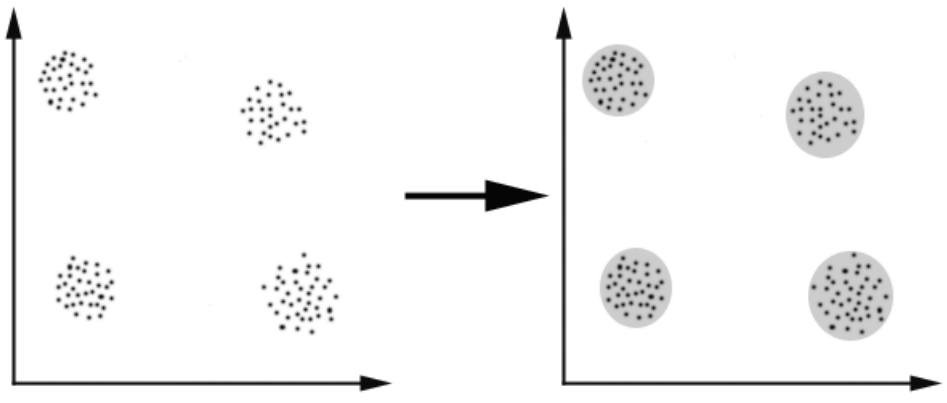


Figure 3.2: Clustering example [8]

$$s(x, x') = \frac{x^t x'}{||x||\ ||x'||} \tag{3.1}$$

Figure 3.3: Similarity and Dissimilarity measures [8]

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}, \\ \text{(If A and B are both empty, we define } J(A,B) = 1) \\ 0 <= J(A,B) <= 1 \tag{3.2}$$

$$d(x, x') = (\sum_{k=1}^{d} |x_k - x'_k|^q)^{1/q} \tag{3.3}$$

3.3.2. Clustering Algorithms

Clustering algorithms are classified in the following 4 categories:

- **Exclusive Clustering** data are grouped in an exclusive way, so that if a certain data point belongs to a definite cluster then it could not be included in another cluster.
- **Overlapping Clustering** uses fuzzy sets to cluster data, so that each point may belong to two or more clusters with different degrees of membership.
- **Hierarchical Clustering** is based on the union between the two nearest clusters. The beginning condition is realized by setting every data point as a cluster. After a few iterations it reaches the final clusters wanted.
- **Probabilistic Clustering** uses a completely probabilistic approach.

Next we go over 3 clustering algorithms that each reside in one of the classes mention above (same order).

3.3.2.1 K-Means Clustering

K-means is one of the simplest unsupervised learning algorithms that solves the well known clustering problem. The procedure follows a simple and easy way to classify a given data set through a certain number of clusters (assume k clusters) fixed a priori. The main idea is to define k centres, one for each cluster. These centroids should be placed in a smart way because of different location causes different result. So, the better choice is to place them as much as possible far away from each other. The next step is to take each point belonging to a given data set and associate it to the nearest centroid. When no point is pending, the first step is completed and an early groupage is done. At this point we need to re-calculate k new centroids as barycenters of the clusters resulting from the previous step. After we have these k new centroids, a new binding has to be done between the same data set points and the nearest new centroid. A loop has been generated. As a result of this loop we may notice that the k centroids change their location step by step until no more changes are done. In other words centroids do not move any more. The

goal is to minimize the error given by the squared error function which sums the distances from each point to the cluster centre for each of the clusters.

Although it can be proved that the procedure will always terminate, the k-means algorithm does not necessarily find the most optimal configuration, corresponding to the global objective function minimum. The algorithm is also significantly sensitive to the initial randomly selected cluster centres. It is recommended to run the algorithm multiple times to reduce this effect.

3.3.2.2 Fuzzy K-Means Clustering

In fuzzy clustering, each point has a probability of belonging to each cluster, rather than completely belonging to just one cluster as it is the case in the traditional k-means. Fuzzy k-means specifically tries to deal with the problem where points are somewhat in between centers or otherwise ambiguous by replacing distance with probability, which of course could be some function of distance, such as having probability relative to the inverse of the distance. Fuzzy k-means uses a weighted centroid based on those probabilities. Processes of initialization, iteration, and termination are the same as the ones used in k-means. The resulting clusters are best analyzed as probabilistic distributions rather than a hard assignment of labels. One should realize that k-means is a special case of fuzzy k-means when the probability function used is simply 1 if the data point is closest to a centroid and 0 otherwise.

In Figure 3.4 we compare the result of clustering the data with k-means algorithm (on the left) and with fuzzy clustering (on the right). For the data point shown as a red marked spot we can say that it belongs more to the B cluster rather than the A cluster. The value of 0.2 indicates the degree of membership to A for such data point.

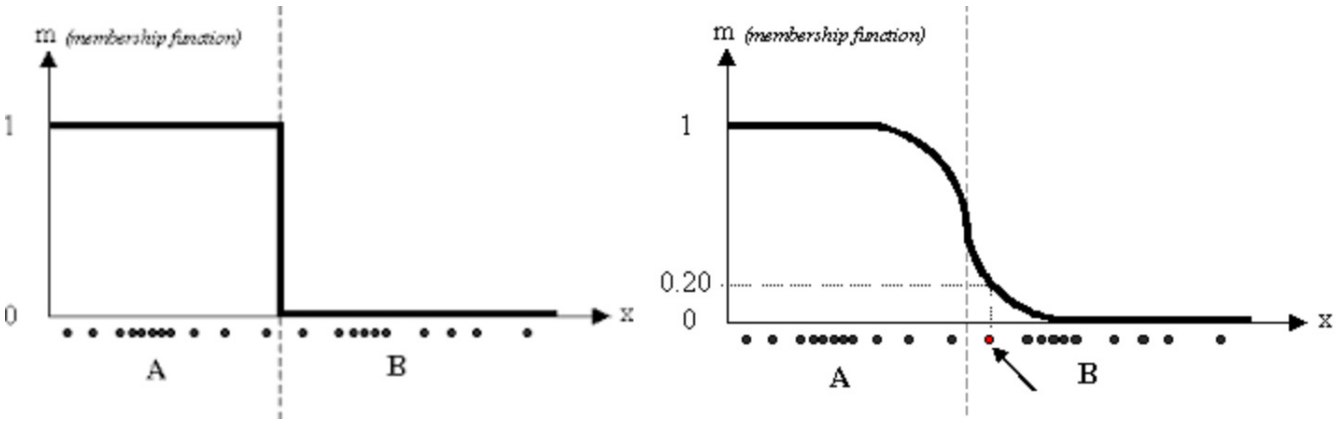


Figure 3.4: Hard vs Soft Clustering [8]

3.3.2.3 Hierarchical Clustering

Consider we have N items to be clustered and an N*N distance matrix. The algorithm starts by assigning each item to a cluster resulting in N initial clusters each containing just one item. The distances between clusters are the given by the distances between the item they contains. Next, find the closest pair of clusters and merge them into a single cluster resulting in N-1 total number of clusters. Compute the distances from the newly created cluster to each of the other clusters. Now we perform these two

steps repeatedly until all items are clustered into just a single cluster containing all N items (Figure 3.5).

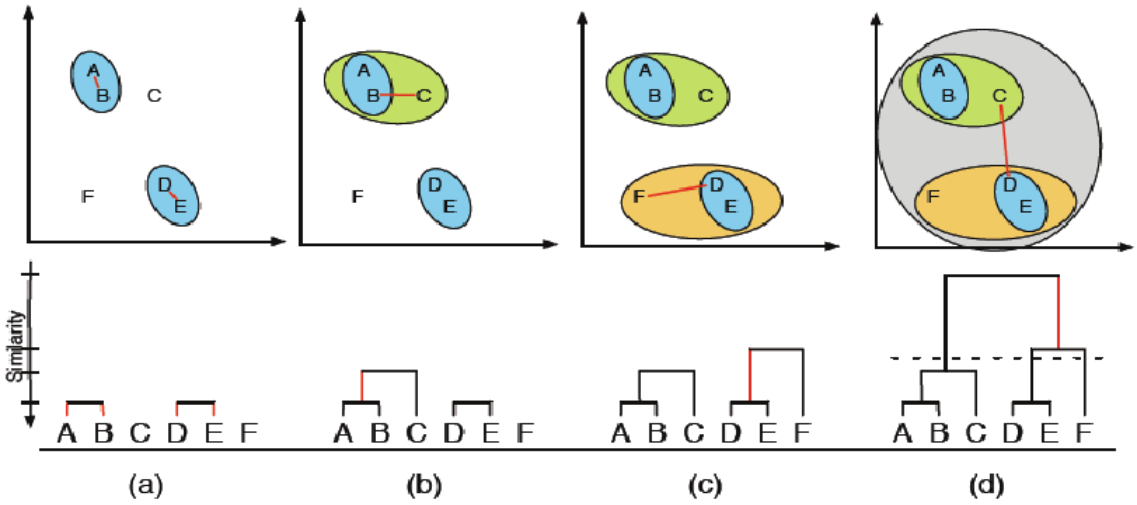


Figure 3.5: Hierarchical Agglomerative Clustering [8]

3.4. Word Sense Induction

Word Sense Induction (WSI) [9] is the task of unsupervised clustering of word usages within a sentence to distinguish senses. The idea is that a WSI system receives a set of sentences containing instances of target lemma+part-of-speech and as output generates groupings of the target usages having the same sense. If we consider the sentences:

1. I like **warm** summer evenings
2. They were greeted by a **warm** welcome
3. The waters of the lake are **warm**

We expect (1) and (3) to be grouped together into one sense while (2) to a different one.

One approach that has seen great results was based on substitute vectors. For each target instance a distribution over possible in-context probable substitutes for the word is considered. Finally, clustering is performed over these distributions. The system is based on n-gram language models (LM).

Following, noticeable improvements were seen when the n-gram LM was replaced with ElMo [4] based biLM and a new technique was used called dynamic symmetric patterns. The idea behind it is that in natural language groups of words tend to appear repeatedly in certain patterns (i.e. "This product may contain traces of **eggs** and **nuts**" which is quite common).

Further on, even better results were achieved by using contextualized vector representations generated by BERT [5]. Another way of improving the results is by using dynamic number of clusters. Up until then the number of clusters was fixed but that imposes a constraint which is not thoroughly true if we think about the fact that some words have more senses than others. Therefore, it is better to consider different number of clusters depending on the target word.

Chapter 4. Analysis and Theoretical Foundation

As stated in Chpater 2 the goal of the system is to perform unsupervised lexical semantic frame induction and the problem can be further divided in the three tasks proposed by Task 2 SemEval2019 [1]. To achieve this, data has to pass through a number of different steps organized as a pipeline which can be visualized in Figure 4.1. In the first place we need to analyze the data we will be working on with. Next, since we are dealing with text data, we should come up with a way of representing it such that we can feed it to a clustering algorithm. But before serving the embeddings to the clustering algorithm we might want to merge them with features generating vectors which will not only represent the text data but also contain insights into it. Finally, we have to decide over what clustering algorithm to use and experiment with its hyperparameters in order to achieve the best possible results. All these steps are part of the pipeline shown in Figure 4.1 and will be analyzed in more detail in what remains from this chapter.

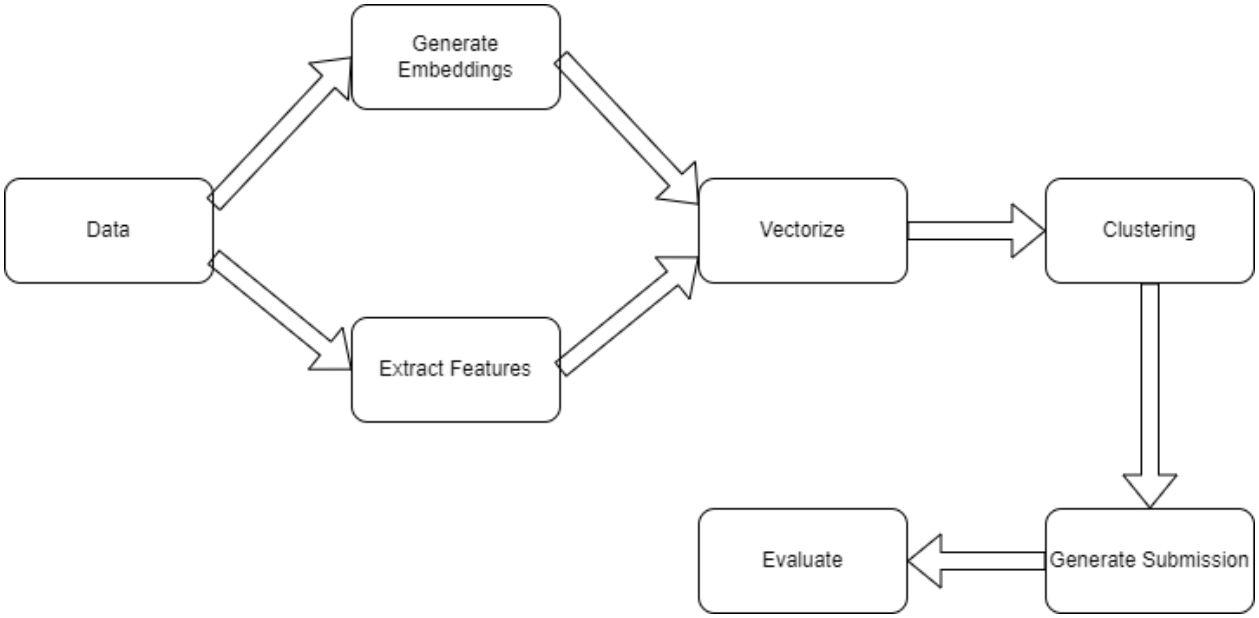


Figure 4.1: System pipeline

4.1. Dataset

The data made available by the organizers to the participants was a subset from the Treebank-3 [10]. Treebank-3 is build on the former Treebank-2 and was developed at the University of Pennsylvania being first released in 1999. It contains the followings:

- One million words of 1989 Wall Street Journal material annotated in Treebank II style.
- A small sample of ATIS-3 material annotated in Treebank II style.
- A fully tagged version of the Brown Corpus.
- Switchboard tagged, dysfluency-annotated, and parsed text.
- Brown parsed text.

The dataset provided for the competition comes with a CoNLL-U format of the tokenized and automatically-parsed (in the universal dependencies formalism) sentences in Penn Treebank (PTB) 3.0.

4.1.1. CoNLL-U Format

CoNLL-U format is a revised version of the CoNLL-X format [11] in which annotations are encoded in plain text files with three types of lines:

1. Word lines containing the annotation of a word/token in 10 fields separated by single tab characters.
2. Blank lines marking sentence boundaries.
3. Comment lines starting with hash (#).

Sentences consist of one or more word lines, and word lines contain the following fields:

1. ID: Word index, integer starting at 1 for each new sentence; may be a range for multiword tokens; may be a decimal number for empty nodes (decimal numbers can be lower than 1 but must be greater than 0).
2. FORM: Word form or punctuation symbol.
3. LEMMA: Lemma or stem of word form.
4. UPOS: Universal part-of-speech tag.
5. XPOS: Language-specific part-of-speech tag; underscore if not available.
6. FEATS: List of morphological features from the universal feature inventory or from a defined language-specific extension; underscore if not available.
7. HEAD: Head of the current word, which is either a value of ID or zero (0).
8. DEPREL: Universal dependency relation to the HEAD (root iff HEAD = 0) or a defined language-specific subtype of one.
9. DEPS: Enhanced dependency graph in the form of a list of head-deprel pairs.
10. MISC: Any other annotation.

For a better understanding, let us consider the example of the sentence "Pierre Vinken, 61 years old, will join the board as a nonexecutive director Nov. 29." taken from the PTB dataset. In Figure 4.2 we can see how the sentence has been tokenized and represented in the CoNLL-U format.

4.1.1.1 Morphological Annotation

The UPOS field contains a part-of-speech tag from the universal POS tag set, while the XPOS optionally contains a language-specific part-of-speech tag, normally from a traditional, more fine-grained tagset. If the XPOS field is used, the treebank-specific documentation should define a mapping from XPOS to UPOS values (which may be

[1] taken from the competition webpage `https://competitions.codalab.org/competitions/19159`

1	Pierre	Pierre	PROPN	NNP	_	2	compound	_	_
2	Vinken	Vinken	PROPN	NNP	_	9	nsubj	_	_
3	,	,	PUNCT	,	_	2	punct	_	_
4	61	61	NUM	CD	_	5	nummod	_	_
5	years	year	NOUN	NNS	_	6	nmod:npmod	_	_
6	old	old	ADJ	JJ	_	2	amod	_	_
7	,	,	PUNCT	,	_	2	punct	_	_
8	will	will	AUX	MD	_	9	aux	_	_
9	join	join	VERB	VB	_	0	root	_	_
10	the	the	DET	DT	_	11	det	_	_
11	board	board	NOUN	NN	_	9	dobj	_	_
12	as	as	ADP	IN	_	15	case	_	_
13	a	a	DET	DT	_	15	det	_	_
14	nonexecutive	nonexecutive	ADJ	JJ	_	15	amod	_	_
15	director	director	NOUN	NN	_	9	nmod	_	_
16	Nov.	Nov.	PROPN	NNP	_	9	nmod:tmod	_	_
17	29	29	NUM	CD	_	16	nummod	_	_
18	.	.	PUNCT	.	_	9	punct	_	_

Figure 4.2: Example of a sentence in CoNLL-U format [1]

context-sensitive and refer to other fields as well). If no language-specific tags are available, the XPOS field should contain an underscore for all words.

The FEATS field contains a list of morphological features, with vertical bar (|) as list separator and with underscore to represent the empty list. All features are represented as attribute-value pairs, with an equals sign (=) separating the attribute from the value. In addition, features should as far as possible be selected from the universal feature inventory and be sorted alphabetically by attribute names. It is possible to declare that a feature has two or more values for a given word: Case=Acc,Dat.

4.1.1.2 Syntactic Annotation

The HEAD and DEPREL fields are used to encode a dependency tree over words. The DEPREL value should be a universal dependency relation or a language-specific subtype of such a relation (defined in the language-specific documentation). As in the case of morphology, syntactic annotation is only provided for words, and tokens that are not words have an underscore in both the HEAD and DEPREL fields.

The HEAD and DEPREL values define the basic dependencies which must be strictly a tree. However, in addition to these basic dependencies, treebanks may optionally provide an enhanced dependency representation that specifies additional dependency relations, for example, when dependencies propagate over coordinate structures. The enhanced dependency representation, which in general is a graph and not a tree, is specified in the DEPS field, using a list of head-relation pairs. We use colon (:) to separate the head and relation and (as usual) vertical bar (|) to separate list items and underscore for the empty list.

4.2. Word Embedding

Word embeddings are a type of word representation that allows words with similar meaning to have a similar representation. They are in fact a class of techniques where individual words are represented as real-valued vectors in a predefined vector space. Each word is mapped to one vector and the vector values are learned in a way that resembles a neural network, and hence the technique is often lumped into the field of deep learning. The key to the approach is the idea of using a dense distributed representation for each word.

Each word is represented by a real-valued vector, often tens or hundreds of dimensions. This is contrasted to the thousands or millions of dimensions required for sparse word representations, such as a one-hot encoding.

The distributed representation is learned based on the usage of words. This allows words that are used in similar ways to result in having similar representations, naturally capturing their meaning. This can be contrasted with the crisp but fragile representation in a bag of words model where, unless explicitly managed, different words have different representations, regardless of how they are used.

4.2.1. Word2Vec

Word2vec [12] is a two-layer neural net that processes text by "vectorizing" words. Its input is a text corpus and its output is a set of vectors: feature vectors that represent words in that corpus. While Word2vec is not a deep neural network, it turns text into a numerical form that deep neural networks can understand.

The purpose and usefulness of Word2vec is to group the vectors of similar words together in vectorspace. That is, it detects similarities mathematically. Word2vec creates vectors that are distributed numerical representations of word features, features such as the context of individual words. It does so without human intervention. Given enough data, usage and contexts, Word2vec can make highly accurate guesses about a word's meaning based on past appearances. Those guesses can be used to establish a word's association with other words (i.e. "man" is to "boy" what "woman" is to "girl").

The output of the Word2vec neural net is a vocabulary in which each item has a vector attached to it, which can be fed into a deep-learning net or simply queried to detect relationships between words.

There are two different model architectures that can be used by Word2vec to create the word embeddings (Figure 4.3). The first one, a method known as continuous bag of words or CBOW, is by using context to predict a target word. The other method is called skip-gram and is using a word to predict a target context.

4.2.1.1 CBOW

Even though Word2Vec is an unsupervised model where you can give a corpus without any label information and the model can create dense word embeddings, Word2Vec internally leverages a supervised classification model to get these embeddings from the corpus.

The CBOW architecture comprises a deep learning classification model in which we take in context words as input, X, and try to predict our target word, Y. For example, if we consider the sentence "Word2Vec has a deep learning model working in the backend.", there can be pairs of context words and target words. If we consider a context window

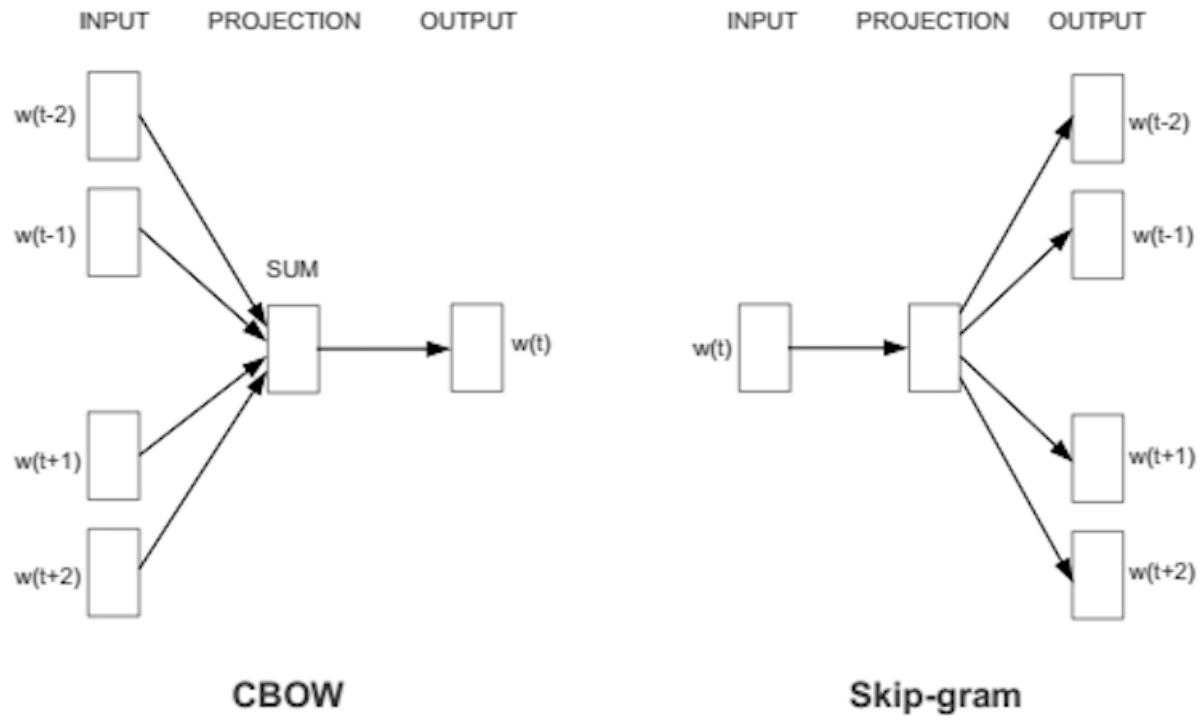


Figure 4.3: Word2Vec Training Models [12]

size of 2, we will have pairs like ([deep, model], learning), ([model, in], working), ([a, learning), deep) and so on. The deep learning model would try to predict these target words based on the context words.

Figure 4.4 along with the following steps describe how the model works:

- The context words are first passed as an input to an embedding layer (initialized with some random weights).
- The word embeddings are then passed to a lambda layer where we average out the word embeddings.
- We then pass these embeddings to a dense SoftMax layer that predicts our target word. We match this with our target word and compute the loss and then we perform backpropagation with each epoch to update the embedding layer in the process.

In the end, after the training is completed we can extract out the embeddings of the words from the embedding layer.

4.2.1.2 Skip-Gram

In the skip-gram model, given a target word, the context words are predicted. So, considering the same sentence "Word2Vec has a deep learning model working in the backend." and a context window size of 2, given the centre word 'learning', the model tries to predict ['deep', 'model'] and so on.

Since the skip-gram model has to predict multiple words from a single given word, we feed the model pairs of (X, Y) where X is our input and Y is our label. This is done by creating positive input samples and negative input samples.

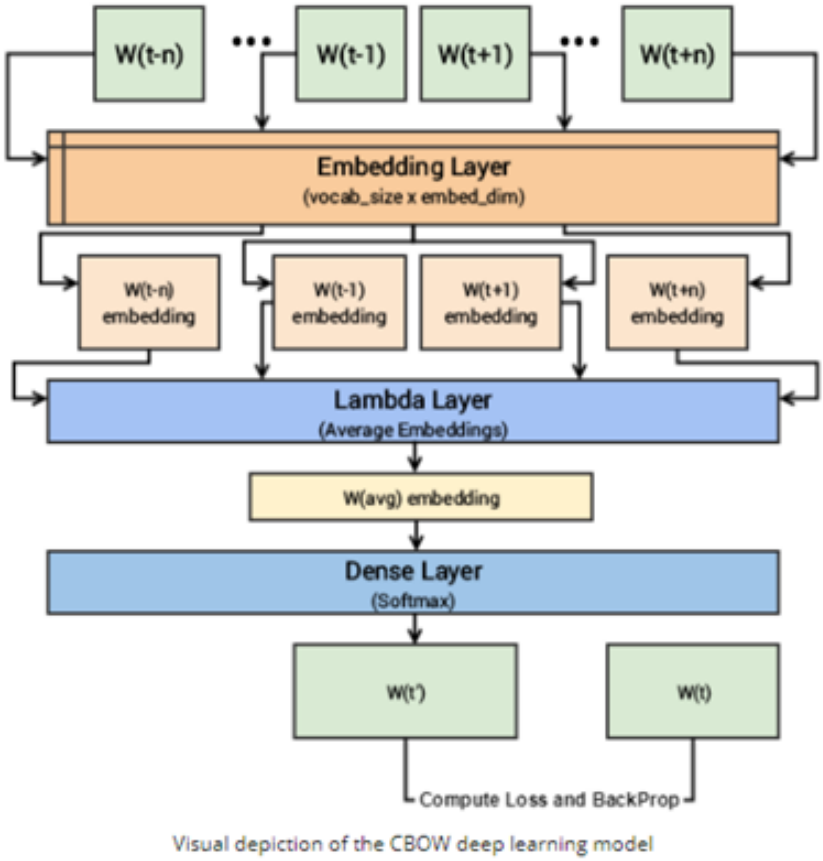


Figure 4.4: CBOW architecture [6]

Positive input samples will have the training data in this form: [(target, context),1] where the target is the target word, context represents the surrounding context words, and label 1 indicates if it is a relevant pair. Negative Input Samples will have the training data in the same form: [(target, random),0]. In this case, instead of the actual surrounding words, randomly selected words are fed in along with the target words with a label of 0 indicating that it's an irrelevant pair.

These samples make the model aware of the contextually relevant words and consequently generate similar embeddings for similar meaning words.

Figure 4.5 along with the following steps describe how the model works:

- Both the target and context word pairs are passed to individual embedding layers from which we get dense word embeddings for each of these two words.
- We then use a 'merge layer' to compute the dot product of these two embeddings and get the dot product value.
- This dot product value is then sent to a dense sigmoid layer that outputs either 0 or 1.
- The output is compared with the actual label and the loss is computed followed by backpropagation with each epoch to update the embedding layer in the process.

In the end, after the training is completed we can extract out the embeddings of the words from the embedding layer.

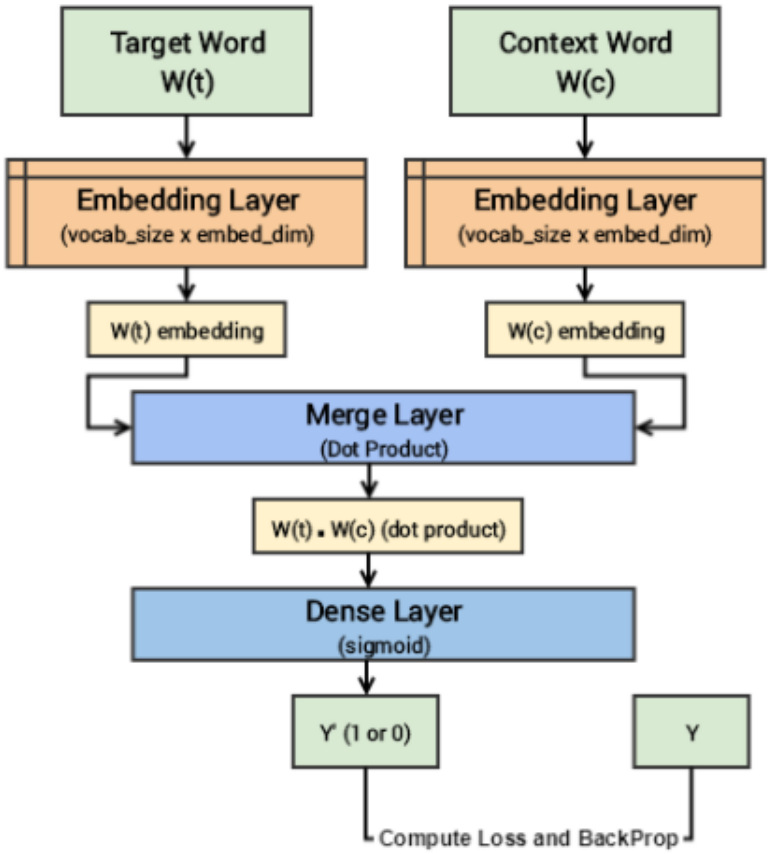


Figure 4.5: Skip-Gram architecture [6]

4.3. Contextualized Word Embedding

The idea of generating word embeddings has seen a major breakthrough once Word2Vec model was released. It made possible for great improvements in the development of plenty of tasks in the NLP field.

But static word embeddings does not account for different senses of the same word. For example, if we consider the two sentences:

1. I like appples.
2. I like Apple macbooks.

Even though the same word ”apple” appears in both of the sentences, the semantic meaning of it is totally different((1) refers to the actual fruit while (2) refers to the Silicon Valley company Apple). Static word embeddings would use the same representation for both appearances of the word ”apple” and that is not exactly what we would want.

Contextualized word embeddings assign each word a representation based on its context, thereby capturing uses of words across varied contexts and encoding knowledge that transfers across languages.

4.3.1. ELMo

ELMo was developed in 2018 by AllenNLP [4] and it goes beyond traditional embedding techniques. It uses a deep, bi-directional LSTM model to create word representations. ELMo analyses words within the context that they are used rather than just

using a dictionary of words and their corresponding vectors. It is also character based, allowing the model to form representations of out-of-vocabulary words.

This therefore means that the way ELMo is used is quite different to word2vec. Rather than having a dictionary 'look-up' of words and their corresponding vectors, ELMo instead creates vectors on the fly by passing text through the deep learning model.

The ELMo architecture begins by training a fairly sophisticated neural network language model, heavily inspired by previous work on large-scale language models. The model starts from a 2-layer bidirectional LSTM backbone. To this 2-layer network, a residual connection is added between the first and second layers. The high-level intuition is that residual connections help deep models train more successfully. The obtained language model is showed in Figure 4.6.

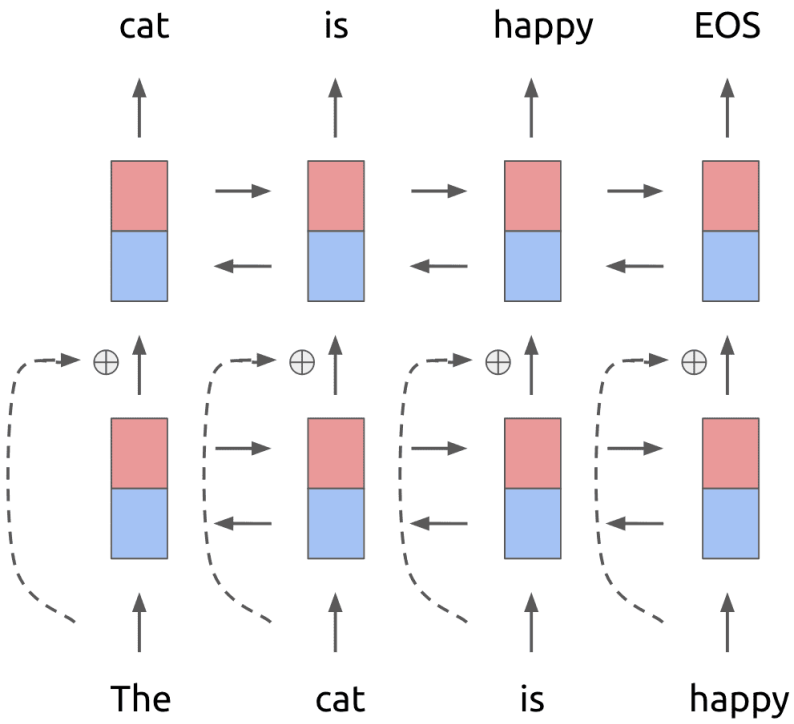


Figure 4.6: BiLM with residual connections model [13]

Now, in traditional neural language models, each token in the first input layer (in this case The cat is happy) is converted into a fixed-length word embedding before being passed into the recurrent unit. This is done either by initializing a word embedding matrix of size (Vocabulary size) x (Word embedding dimension), or by using a pretrained embedding such as Word2Vec for each token.

However, for the ELMo language model, we do something a bit more complex. Rather than simply looking up an embedding in a word embedding matrix, we first convert each token to an appropriate representation using character embeddings. This character embedding representation is then run through a convolutional layer using some number of filters, followed by a max-pool layer. Finally this representation is passed through a 2-layer highway network before being provided as the input to the LSTM layer (see Figure 4.7).

These transformations to the input token are beneficial because using character embeddings allows us to pick up on morphological features that word-level embeddings

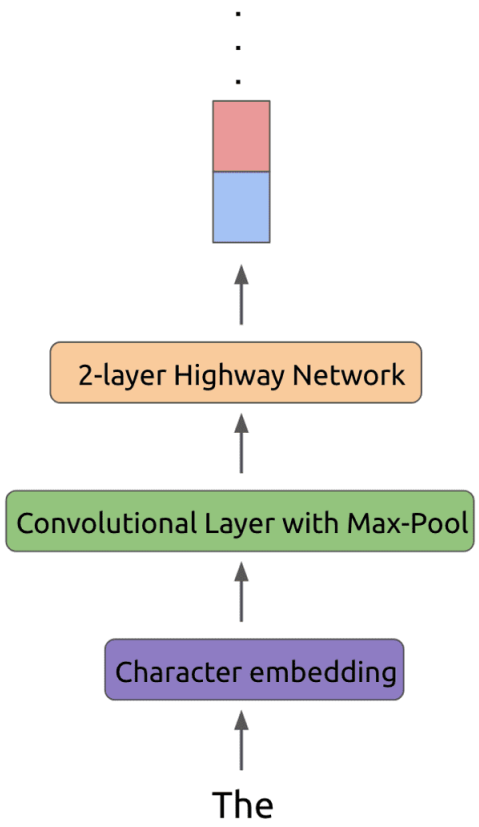


Figure 4.7: Pipeline of transformations applied for each token [13]

could miss. In addition, using character embeddings ensures that we can form a valid representation even for out-of-vocabulary words, which is of great interest.

Assume that we are looking at the k^{th} word in our input. Using our trained 2-layer language model, we take the word representation x_k as well as the bidirectional hidden layer representations $h_{1,k}$ and $h_{2,k}$ and combine them into a new weighted task representation. The visual representation is shown in Figure 4.8).

To be more concrete about the mathematical details, the function f described performs the following operation on word k of the input as characterized by the Eq 4.1, where s_i represent softmax-normalized weights on the hidden representations from the language model and γ_k represents a task-specific scaling factor.

$$ELMo_k^{task} = \gamma_k \cdot (s_0^{task} \cdot x_k + s_1^{task} \cdot h_{1,k} + s_2^{task} \cdot h_{2,k}) \tag{4.1}$$

The ELMo language model is trained on a sizable dataset: the 1B Word Benchmark. In addition, the language model really is large-scale with the LSTM layers containing 4096 units and the input embedding transform using 2048 convolutional filters.

4.3.2. BERT

BERT, which stands for Bidirectional Encoder Representations from Transformers, is based on Transformers, a deep learning model in which every output element is connected to every input element, and the weightings between them are dynamically calculated based upon their connection, a process known in NLP as attention.

BERT's key technical innovation is applying the bidirectional training of Trans-

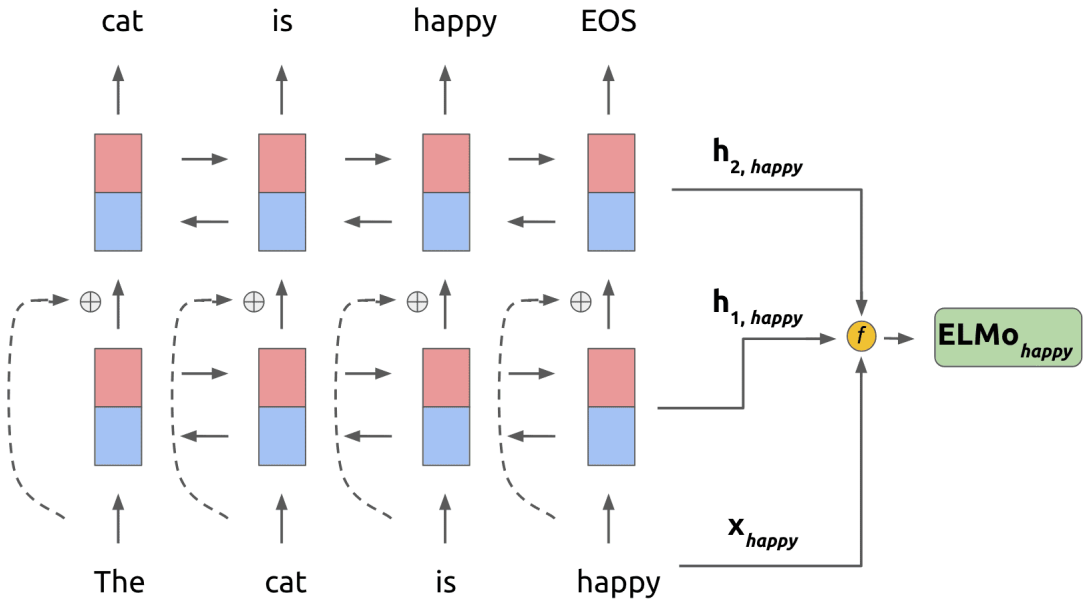


Figure 4.8: Example of combining the bidirectional hidden representations and word representation for "happy" to get an ELMo-specific representation [13]

former to language modelling. This is in contrast to previous efforts which looked at a text sequence either from left to right or combined left-to-right and right-to-left training. The paper's results [5] show that a language model which is bidirectionally trained can have a deeper sense of language context and flow than single-direction language models. In the paper, the researchers detail a novel technique named Masked LM (MLM) which allows bidirectional training in models in which it was previously impossible.

4.3.2.1 Transformers

Transformers work by leveraging attention, a powerful deep-learning algorithm, first seen in computer vision models.

Machine Learning models need to learn how to pay attention only to the things that matter and not waste computational resources processing irrelevant information. Transformers create differential weights signaling which words in a sentence are the most critical to further process.

A transformer does this by successively processing an input through a stack of transformer layers, usually called the encoder. If necessary, another stack of transformer layers, the decoder, can be used to predict a target output (Figure 4.9). Transformers are uniquely suited for unsupervised learning because they can efficiently process millions of data points.

4.3.2.2 Masked LM (MLM)

The idea here is "simple": Randomly mask out 15% of the words in the input meaning that we replace them with a [MASK] token. The model then attempts to predict the original value of the masked words, based on the context provided by the other, non-masked, words in the sequence (see Figure 4.10). In technical terms, the prediction of the output words requires:

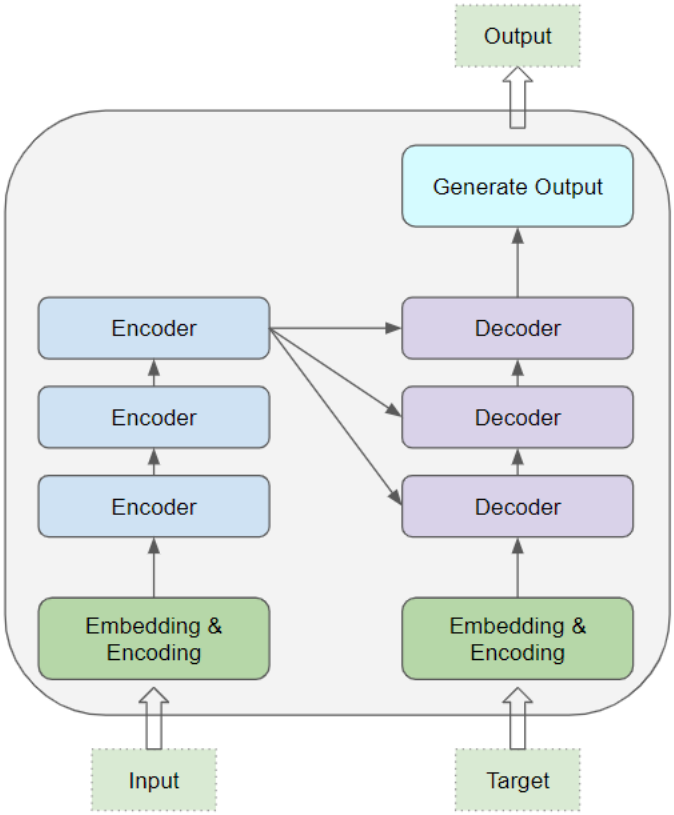


Figure 4.9: Transformer model structure [14]

1. Adding a classification layer on top of the encoder output.
2. Multiplying the output vectors by the embedding matrix, transforming them into the vocabulary dimension.
3. Calculating the probability of each word in the vocabulary with softmax.

The BERT loss function takes into consideration only the prediction of the masked values and ignores the prediction of the non-masked words. As a consequence, the model converges slower than directional models, a characteristic which is offset by its increased context awareness.

4.3.2.3 Next Sentence Prediction (NSP)

In order to understand relationship between two sentences, BERT training process also uses next sentence prediction. During training, 50% of the inputs are a pair in which the second sentence is the subsequent sentence in the original document, while in the other 50% a random sentence from the corpus is chosen as the second sentence. The assumption is that the random sentence will be disconnected from the first sentence.

To help the model distinguish between the two sentences in training, the input is processed in the following way before entering the model (see also Figure 4.11):

1. A [CLS] token is inserted at the beginning of the first sentence and a [SEP] token is inserted at the end of each sentence.
2. A sentence embedding indicating Sentence A or Sentence B is added to each token. Sentence embeddings are similar in concept to token embeddings with a vocabulary of 2.

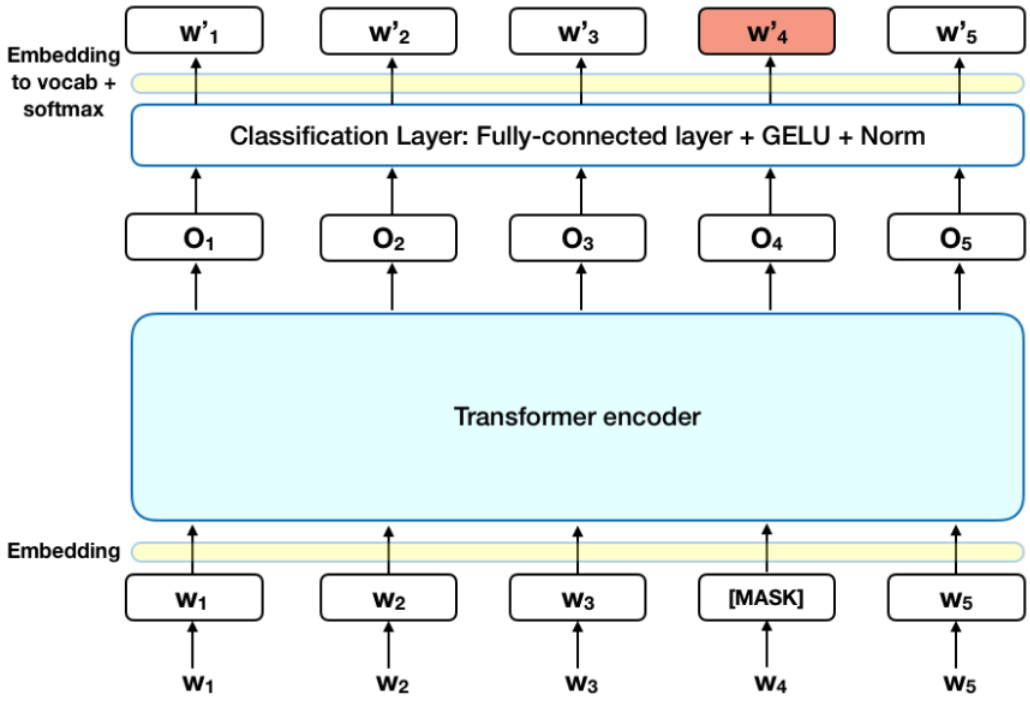


Figure 4.10: Masked Language Model architecture [15]

3. A positional embedding is added to each token to indicate its position in the sequence.

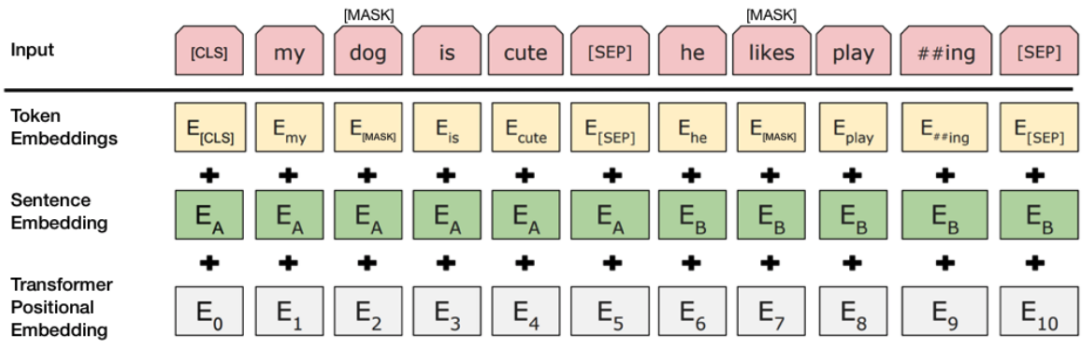


Figure 4.11: Next Sentence Prediction Model architecture [15]

To predict if the second sentence is indeed connected to the first, the following steps are performed:

1. The entire input sequence goes through the Transformer model.
2. The output of the [CLS] token is transformed into a 2×1 shaped vector, using a simple classification layer (learned matrices of weights and biases).
3. Calculating the probability of IsNextSequence with softmax.

When training the BERT model, Masked LM and Next Sentence Prediction are trained together, with the goal of minimizing the combined loss function of the two strategies.

4.4. Agglomerative Clustering

As we have seen in Chapter 3 in machine learning, unsupervised learning is a machine learning model that infers the data pattern without any guidance or label. Agglomerative Clustering or bottom-up clustering essentially started from an individual cluster where each data point is considered as an individual cluster, also called leaf, then every cluster calculates their distance with each other. The two clusters with the shortest distance with each other would merge creating what we called node. Newly formed clusters once again calculating the member of their cluster distance with another cluster outside of their cluster. The process is repeated until all the data points assigned to one cluster called root. The result is a tree-based representation of the objects called dendrogram (see Figure 4.12).

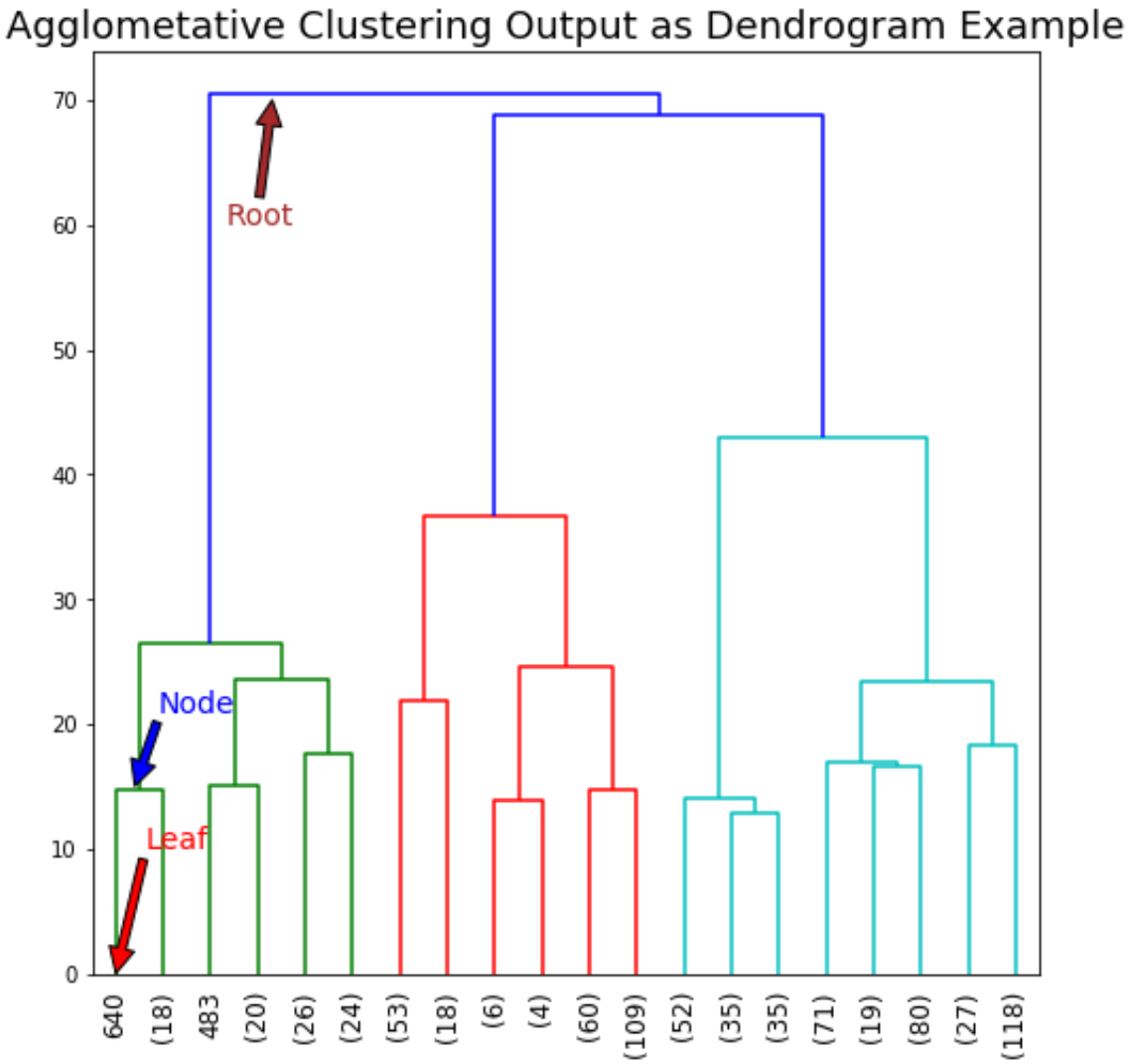


Figure 4.12: Agglomerative Clustering dendrogram example [16]

Agglomerative Clustering does not present any exact number of how our data should be clustered. It is up to us to decide where is the cut-off point.

In Chapter 3 we mentioned about proximity measures and how they are used to measure the similarity between two sample by computing the distance between them. This is a vital step for the clustering algorithm and the type of distance metric we choose

to use influences the results we obtain. But aside from the metric we use to compute the distance, another important part is the criterion used to merge clusters. This is called linkage and it can be specified in 4 different ways as follows:

1. Ward linkage (Figure 4.16): distance between clusters is the sum of squared differences within all clusters (Eq. 4.5).
2. Average linkage (Figure 4.15): distance between clusters is the average distance between each point in one cluster to every point in other cluster (Eq. 4.4).
3. Complete linkage (Figure 4.14): distance between two clusters is the longest distance between two points in each cluster (Eq. 4.3).
4. Single linkage (Figure 4.13): distance between two clusters is the shortest distance between two points in each cluster (Eq. 4.2).

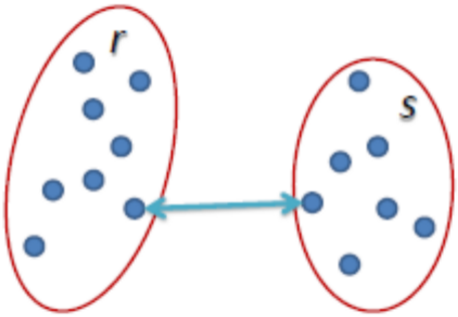


Figure 4.13: Single Linkage [17]

$$L(r,s) = min(D(x_{ri}, x_{sj})) \tag{4.2}$$

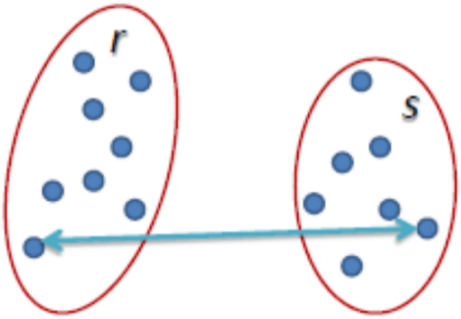


Figure 4.14: Complete Linkage [17]

$$L(r,s) = max(D(x_{ri}, x_{sj})) \tag{4.3}$$

$$L(r,s) = \frac{1}{n_r n_s} \sum_{i=1}^{n_r} \sum_{j=1}^{n_s} D(x_{ri}, x_{sj}) \tag{4.4}$$

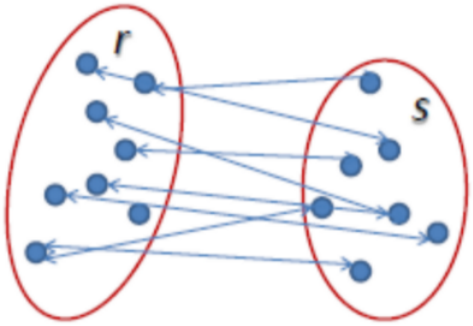


Figure 4.15: Average Linkage [17]

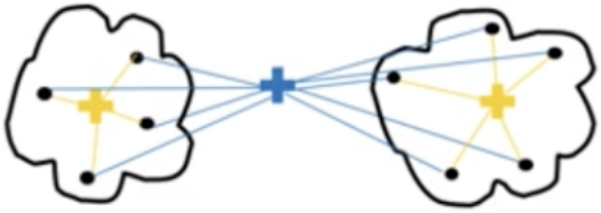


Figure 4.16: Ward Linkage [17]

$$L(r,s) = \sum_{x \in r \cup s} D(x, \mu_{r \cup s})^2 \tag{4.5}$$

Chapter 5. Detailed Design and Implementation

In the previous Chapter 4 we were presented the theoretical concepts on which the implementation is based. Over the course of this chapter, we will dive deeper into the implementation details and we will go through the steps required to achieve our goal.

5.1. Data Preprocessing

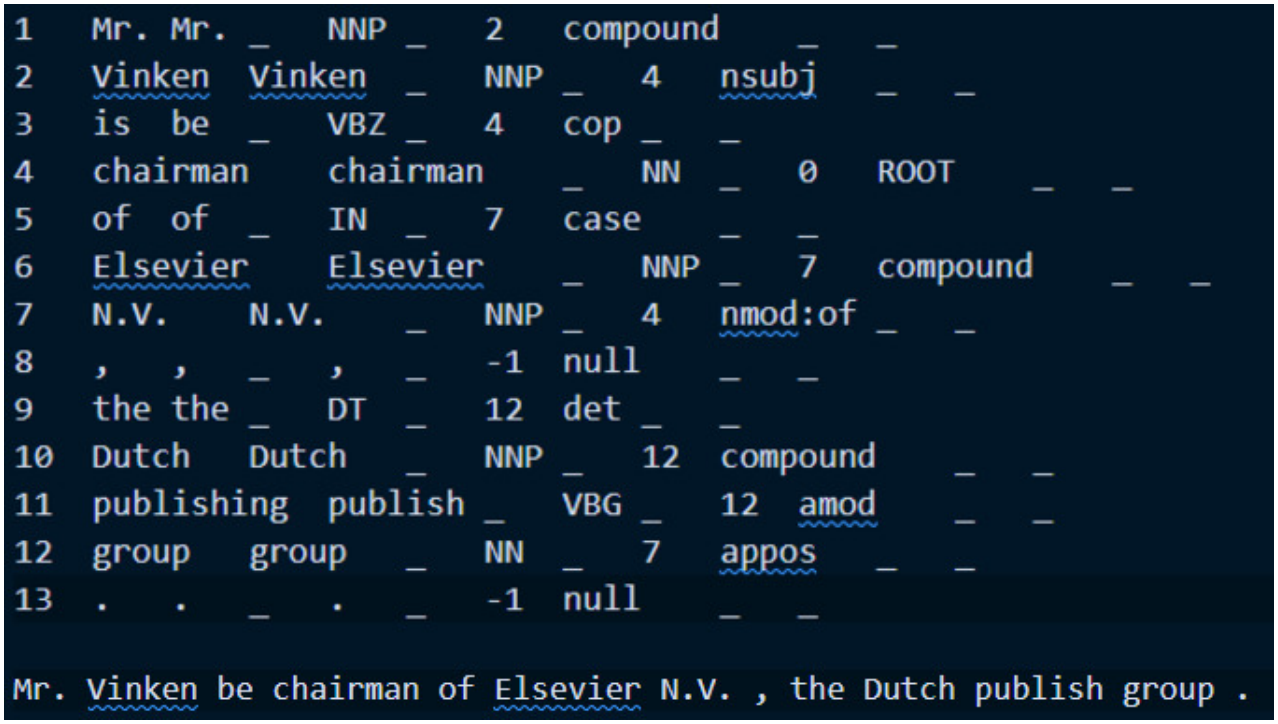
The goal of this step is to bring the data provided by the organizers to form in which we can easily work with later. As mentioned previously, we are working with a subset of the Treebank 3.0 dataset which is build based on 1989 Wall Street Journal. We are given the data in CoNLL-U format contained in a text file. The file "pos-gold-dep-auto.conll.txt" found in the "data" folder contains the sentences layed out in the CoNNL-U format along with an unique identifier.

We first want to split this file in multiple files such that each file contains a single sample (sentence) and the name of the file will be the actual id of the sentence. Therefore, in the folder "processed_data/parsed_conll" we will find multiple text files with the name stating with a "#" and 8 more digits each containing the CoNLL-U format of a single sentence. This is done by the function `split_conll_file()` which receives the path to the file containing the dataset and the path to the directory which will contained the resulting per sentence files.

Then we want to extract the actual sentence from the CoNLL-U format in order to later be able to feed it to the models for generating the embeddings. But to do so, we need some kind of tool to work with CoNNL-U formats and we have found an open source library called conllu [1] which implements a CoNLL-U parser. Now we can use the parser to obtain the initial sentence by concatenating the FORMs together. But usually when working with machine learning algorithm you want the data to be as standardized as possible as this way the results tend to be better. Hence, a better way of reconstructing the sentence is by using the LEMMAs such that we lemmatize the words from the sentence. Lemmatization is the process of grouping together the inflected forms of a word so they can be analysed as a single item, identified by the word's lemma. This whole process is realized by two functions `parse_conlls()` and `conll_to_sentence()` where the former internally calls the latter. The function `parse_conlls()` receives the path to the directory which contains the split sentence in CoNLL-U format and the path to the directory which will contained the resulting per sentence files. Function `conll_to_sentence()` simply builds the sentence from the CoNLL-U format (using the above mentioned library to do the parsing) using the LEMMA token field.

Finally, in Figure 5.1 we can the result of the data preprocessing phase exemplified on the sample with id "#20001002".

[1] available at `https://pypi.org/project/conllu/`



```
1   Mr. Mr. _   NNP _   2   compound    _   _
2   Vinken  Vinken  _   NNP _   4   nsubj   _   _
3   is  be  _   VBZ _   4   cop _   _
4   chairman    chairman    _   NN  _   0   ROOT    _   _
5   of  of  _   IN  _   7   case    _   _
6   Elsevier    Elsevier    _   NNP _   7   compound    _   _
7   N.V.    N.V.    _   NNP _   4   nmod:of _   _
8   ,   ,   _   ,   _   -1  null    _   _
9   the the _   DT  _   12  det _   _
10  Dutch   Dutch   _   NNP _   12  compound    _   _
11  publishing  publish _   VBG _   12  amod    _   _
12  group   group   _   NN  _   7   appos   _   _
13  .   .   _   .   _   -1  null    _   _

Mr. Vinken be chairman of Elsevier N.V. , the Dutch publish group .
```

Figure 5.1: The CoNLL-U format is contained in the file /processed_data/parsed_conll/#20001002.txt while the sentence below it is contained in the file /processed_data/parsed_sentences/#20001002.txt

5.2. Generating Embeddings

Having the data processed and ready to be worked on, the next step is generate numerical representations of it. This is required because the clustering algorithm works with numerical vectors. We want to generate embeddings for each of the sentence and also for target words which are the verbs along with their arguments. In order to obtain the best results, we experiment with multiple embedding models namely Word2Vec, ELMo and BERT. The idea is to generate representations of the data with each individual model and then use combinations of there representations to see which will yield the best results.

5.2.1. BERT

As mentioned in Chapter 4, BERT is a deep learning model used for generating contextualized word embeddings. The model is available in two architectures: the base version which has 12 layers of encoders resulting in a vector of size 768 and the large version which has 24 encoding layers and the output vector of size 1024. We chose to work with the base version as it shown to offer very good performance and is also pretty fast. Nikolay Arefyev solution is also based on this version of the BERT model as described in his paper [18].

The model is available at TensorFlow Hub and it comes in two versions each with its preprocessing model. The first one is bert cased [2] and the second one is the bert

[2]encoder available at `https://tfhub.dev/tensorflow/bert_en_cased_L-12_H-768_A-12/4`, preprocessing available at `https://tfhub.dev/tensorflow/bert_en_cased_preprocess/3`

uncased [3]. The preprocessing model is used to preprocess plain text inputs into the input format expected by BERT resulting in tokenized word pieces.

The output of the model is a dictionary of output vectors of size 768. From this dictionary we identified two keys. The "pooled_output" which is the very last output of the model and the "encode_outputs" which is a list containing the outputs of each encoder layer of the model (since there are 12 layers in the based version this is also the length of the list). The idea here is that the output at different layers might represent better different features of the input. For this reason, when generating the embeddings for the context (sentences) we used the "pooled_output" as well as the output from layers 3, 6 and 9.

For generating the embedding for the context we have two functions `generate_context_embeddings_pooled_output()` and `generate_context_embeddings_by_layer()`. The sentences are read from "/processed_data/parsed_sentences/" directory and loaded into a dataframe (using the functions `task1_to_df()`, `task2_2_to_df()`, `task2_1_to_df()`) which is then passed to this functions. We want to generate embeddings for verbs and their arguments also so for this we have the function `generate_word_embeddings()` which works similar to the two previously mentioned.

In order to run the model and generate the embeddings one has to use the function `get_bert_embeddings()`) and pass the text data to it. The model generates the embeddings by running in batches and concatenating their results. The machine used was set to run on CUDA (graphics card NVIDIA GeFroce MX150) and the batch size used was set to 50. In case of a machine better equiped, the batch size can be increase which will greatly improve the running time.

The vector embeddings are written into "embeddings/ber_cased/" and "embeddings/ber_uncased/" respectively folders as numpy arrays with the name being the id in case of the sentences and in case of the verbs and their arguments the actual word is used.

5.2.2. ELMo

ELMo model is also a model used for generating contextualized word embeddings using character-based word representations. The model is available at TensorFlow Hub [4] from where it can be downloaded (Eugénio Ribeiro offered a solution based on this model described in his paper [19]). The output of the model is a dictionary containing the embedding vectors. We can choose between the output at the two different lstm layers (keys "lstm_outputs1" and "lstm_outputs2"), the weighted sum of the 3 layers (key "elmo") or a fixed mean-pooling of all contextualized word representations (key "default"). In this case, we chose to work only with the "default" output which contains embedding vectors of size 1024.

The script follows the same pattern as we have seen in case of BERT. There are two functions for generating the context and word embeddings: `generate_context_embeddings()`, `generate_word_embeddings()`. Both functions internally make a call to the function `get_elmo_default_embeddings()` which in the same way as `get_bert_embeddings()`) runs the ELMo model in batches of the data. The ELMo model seems to be more complex than BERT and thus in order to deal with the memory overflow issues we had to scale

[3] encoder available at `https://tfhub.dev/tensorflow/bert_en_uncased_L-12_H-768_A-12/4`, preprocessing available at `https://tfhub.dev/tensorflow/bert_en_uncased_preprocess/3`

[4] `https://tfhub.dev/google/elmo/3`

down the batch size to only 10.

In order to preserve the same folder structure to ease the further work, the vector embeddings are written into "embeddings/elmo/" folder using the same naming convention as mentioned earlier.

5.2.3. Word2Vec

Unlike BERT and ELMo models, Word2Vec generates static word representations meaning that the representations do not take into consideration the context in which the words appear. The model is made available by Google [5]. It is pre-trained on Google News corpus containing 3 billion running words. As the representations are generated statically, the model works rather like a dictionary where the key is the word and the value is its vector representation of size 300. Saba Anwar obtained his best results using the Google News Word2Vec as he explained in his paper [20].

We chose to embed only the verbs and their arguments with this model, although it can be possible to generate embedding for a sentence by computing the mean vector of all its word representations. Therefore, we identify only the function `generate_word_embeddings()` which internally makes a call to the function `get_word2vec_embeddings()`.

The function `get_word2vec_embeddings()` is responsible for working with the Word2Vec model to obtain the embeddings for the target data received as parameter. As we mentioned earlier, the model works like a dictionary so we expect the job to be as simple as "return Word2Vec[word]", but that is not entirely true. The model can handle only single words mostly and there are few exceptions for some compound words. Considering that we need to embed phrasal verbs such as "buy back" or "join forces", we have to come up with a strategy to deal with them. For this reason, we first check if the word is contained in the dictionary as it is, if so we use that representation. Otherwise, we split the original word into the words composing it and check the words in order to see if we have the representation of any. The first word that is found in the keyset will determine the representation of the whole original word. In case both of these steps fail then we provide a default embedding vector filled with zeroes.

As for the other embedding methods, the output vectors will be stored in the "embeddings/word2vec/" folder using the same naming convention as mentioned earlier.

5.3. Extracting Features

For solving the first task where verbs have to be clustered in frame type groups it seems to be enough to make use only of the context and verb embeddings as they tend to contain most of the necessary information needed by the clustering algorithm to make good decisions. But for the remaining two tasks the arguments are part of the context and therefore the context embedding does not offer much in this regard. We need to consider more in-depth relations between verbs and their arguments in order to improve the decision making of the clustering algorithm.

In Chapter 4 we have gone over the CoNLL-U format and explained the annotations it may contain. Some of these are now of great interest to us as they comprise morphological and syntactical information. By making use of these annotations we expect to see great improvements in the results for tasks B.1 and B.2. In our case, the dataset

[5] `https://code.google.com/archive/p/word2vec/`

contains annotations for the XPOS and DEPREL fields. In what follows we will describe how we took advantage of them.

5.3.0.1 XPOS

These tags mark language-specific part-of-speech categories. There are a total of 35 different labels used for this field. Since we want to obtain an embedding for each label such that they are represented numerical and we can also distinguish between them it is necessary to categorize them. This process implies creating vectors of size equal to the number of categories which in our case is 35. Further on, each vector has the value one on a single position in the vector and the rest are set to 0. The position of the one in vector identifies the label in the label's list. This logic is implemented in function `generate_xpos_embeddings()` which parses through the dataset and identifies all unique labels of the XPOS field form the CoNLL-U format and then creates the categorical vectors. These vectors are then saved in the folder "embeddings/xpos/" as numpy arrays each being named after the label it represents (see Figure 5.2).

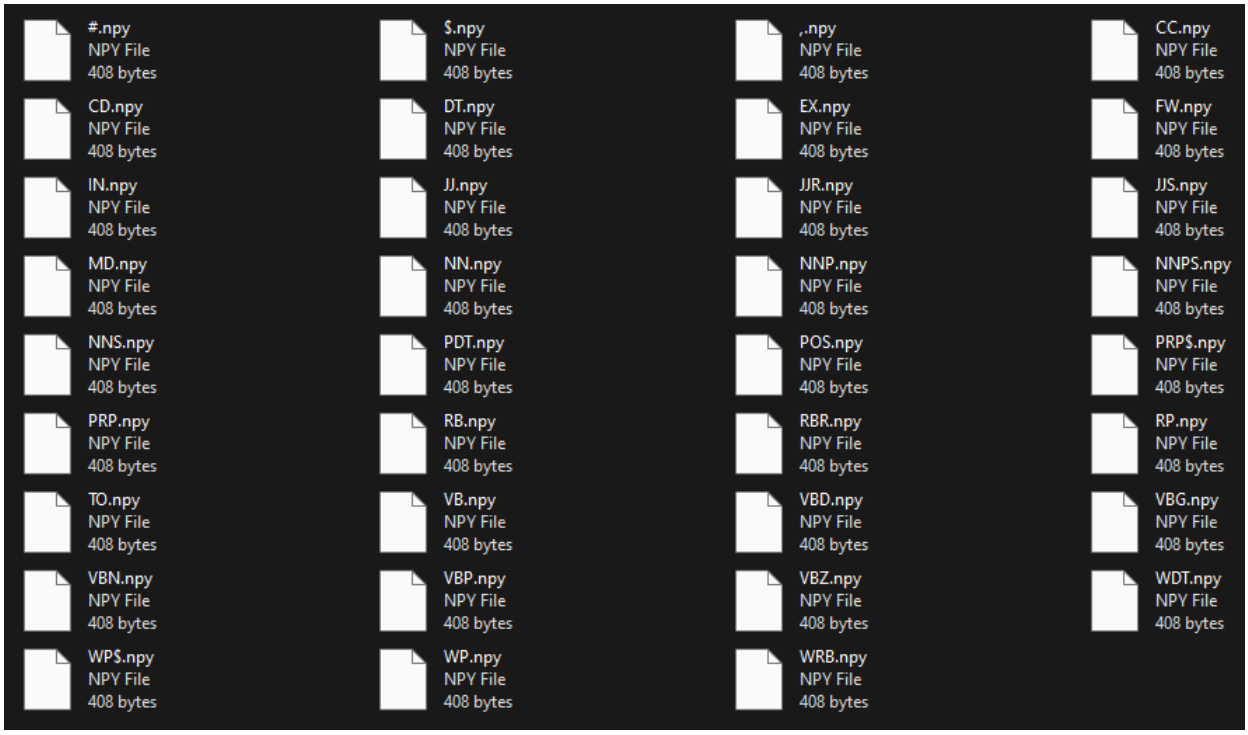


Figure 5.2: The resulting vectors from the folder "embeddings/xpos/"

5.3.0.2 DEPREL

These tags refer to the Universal dependency relation which are relations between two words in a sentence with one word being the governor and the other being the dependent of the relation. We identified a total of 91 different labels for this field and we applied the same process of categorization as presented for the XPOS. But since we are working with just a subset of the dataset which includes just 4650 sentences, the distribution of the target words over the 91 DEPREL tags is pretty sparse. Because of that we are interested to see if we can find more general tag such that to reduce the total

number of tags. And luckily for us this is possible because some of the categories are particular case of a more general one. For example if we consider the nominal modifier there are 3 tags realted to it:

1. nmod: nominal modifier
2. nmod:poss: possessive nominal modifier
3. nmod:tmod: temporal modifier

Hence, we can reduce these 3 categories to just a general one namely "nmod". By applying this principle over all the 91 tags we reach a final number of just 35 tags. We apply the categorization process to these 35 tags too. The function responsible for this is `generate_deprel_embeddings` and the generated vector representations are stored in the folders "embeddings/deprel/extended/" (for the 91 tags) and "embeddings/deprel/primary/" (for the 35 tags) with the same described naming convention (see Figure 5.3).

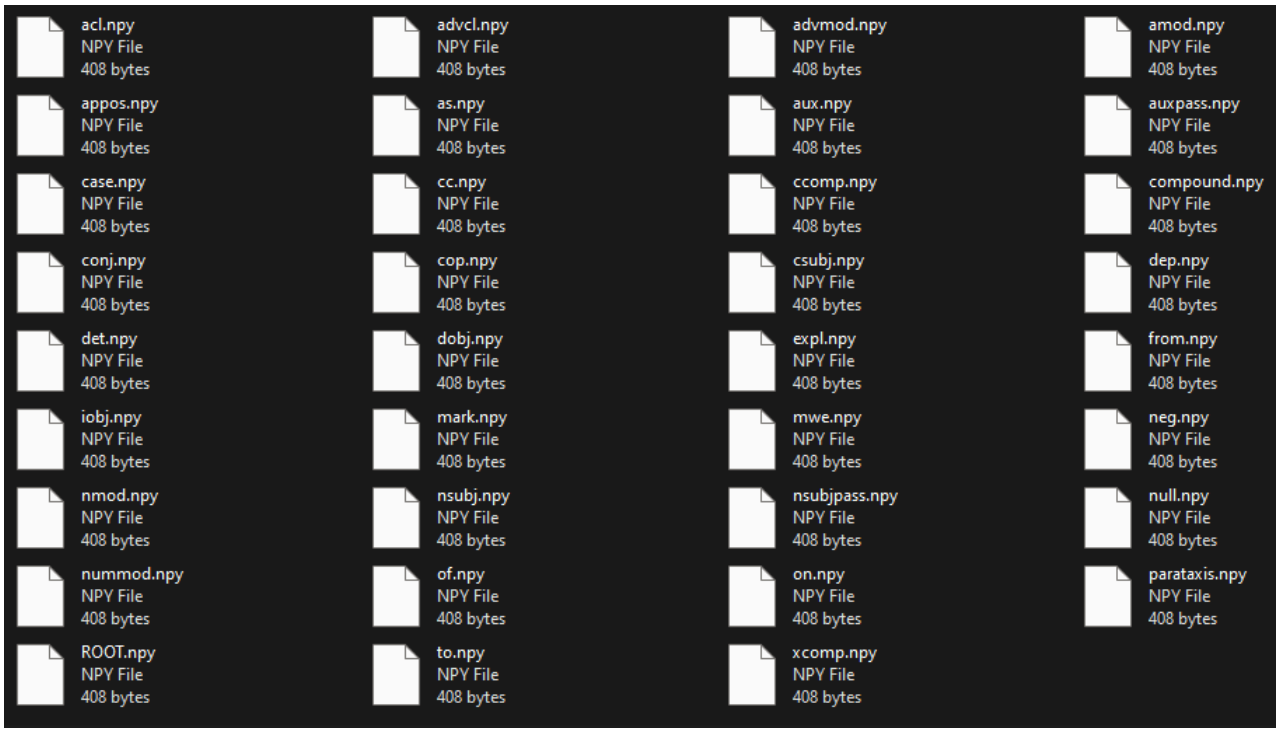


Figure 5.3: The resulting vectors from the folder "embeddings/deprel/primary/"

5.4. Generating Vectors

Up until this point we have generated embeddings for the context and verbs as well as their arguments and also create numerical representation of features based on XPOS and DEPREL field from the CoNLL-U format. At this step we want to combine these vector representations into just a single vector which will then be passed to the clustering algorithm.

We want to experiment with different combinations of features during vectorization in order to see which will perform better. The way these features are combined is by concatenating their individual representation such that the resulting vector size is equal

to the sum of sizes of the vector representations used. The process of creating the vectors differs slightly for each task, therefore we will go over them one by one.

Task A requires the grouping of verbs to frame type clusters. For this task the vectors are created based only on the context and verb embeddings. We considered the context embeddings generated by all three models and the vectors are simply just that or along with the context the verbs are also considered. In this later case, the vector is obtained by horizontally concatenating the vector representations of the two. We made all the combinations possible meaning that the context embeddings are paired with the verb embeddings generated by all the mentioned models. In total there are 3 vectors based only on context and another $2 * 3$ vectors based on context and verbs. The logic for creating the vectors for Task A is implemented in function `get_vectors_task_1()`. After creation, the vectors are also saved in the folder "vectors/task1/" as numpy arrays.

Next, we take a look at task B.2 in which our goal is to cluster arguments of verbs to generic roles. At this stage the process of building the vectors is getting a bit more complex. At the base, the vectors are build upon the word representations of the arguments exclusively or combined with the context embeddings. After these based vectors are created there is the option to add other features to the representation. Two of these features were previously describe namely XPOS and DEPREL, but along these there are two more which were mentioned by Saba Anwar in his paper explaining his approach [20]. One of them is a simple flag which states if the target word is before or after the verb in the sentence order of the words (i.e. 1 if it is before and 0 otherwise). The other one is represented by a number which the position of the word in the sentence order of the words but relative to the number of arguments in that sentence (i.e. if there are 3 arguments then the value for this feature will be one of 1,2,3 where 1 means that word appears in sentence before the other two arguments). These features are added dynamically by the function `concat_features()` which is called internally by both functions `get_vectors_task_2_1()` and `get_vectors_task_2_2()` after the base of the vector was built. In the same way the generated vectors are saved in the folder "vectors/task2_2/" as numpy arrays.

Finally for Task B.1 we aim to cluster the arguments of verbs to frame-specific slots. The vectorizing process is pretty much the same as for the Task B.2 with only one exception. We add another feature which represents the frame associated to the sample in Task A. The frame is represented by a vector obtained through the same categorization process described before applied on the frames predicted at the first task. This is handled by the same function which handles the other features too namely `concat_features()`. The resulting vectors are saved in the folder "vectors/task2_1/" as numpy arrays.

To have a better understanding of how the vectors are created we can visualize the structure in Figure 5.4. It is worth mentioning that the features part may or may not be present depending on the task (the features are considered only for Task B.1 and Task B.2). Also not all the features must be present at the same time. In the same way the base of the vectors might contain only one, two or all three of the blocks (note that the word embedding block refers to the the embeddings of the arguments of the verbs specific to Task B.1 and Task B.2). The concatenation order is also one to one to the order of blocks in the diagram.

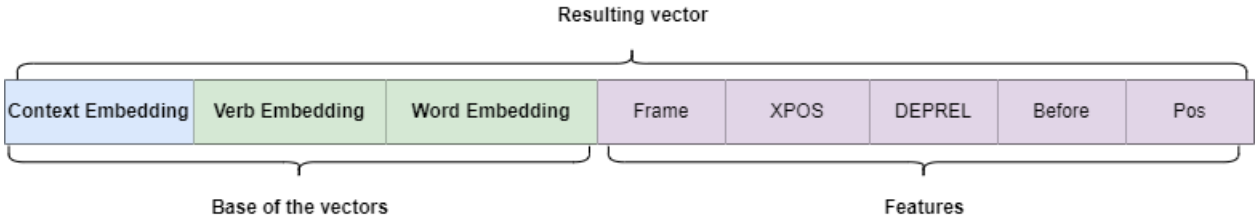


Figure 5.4: The layout of the vectors

5.5. Submission

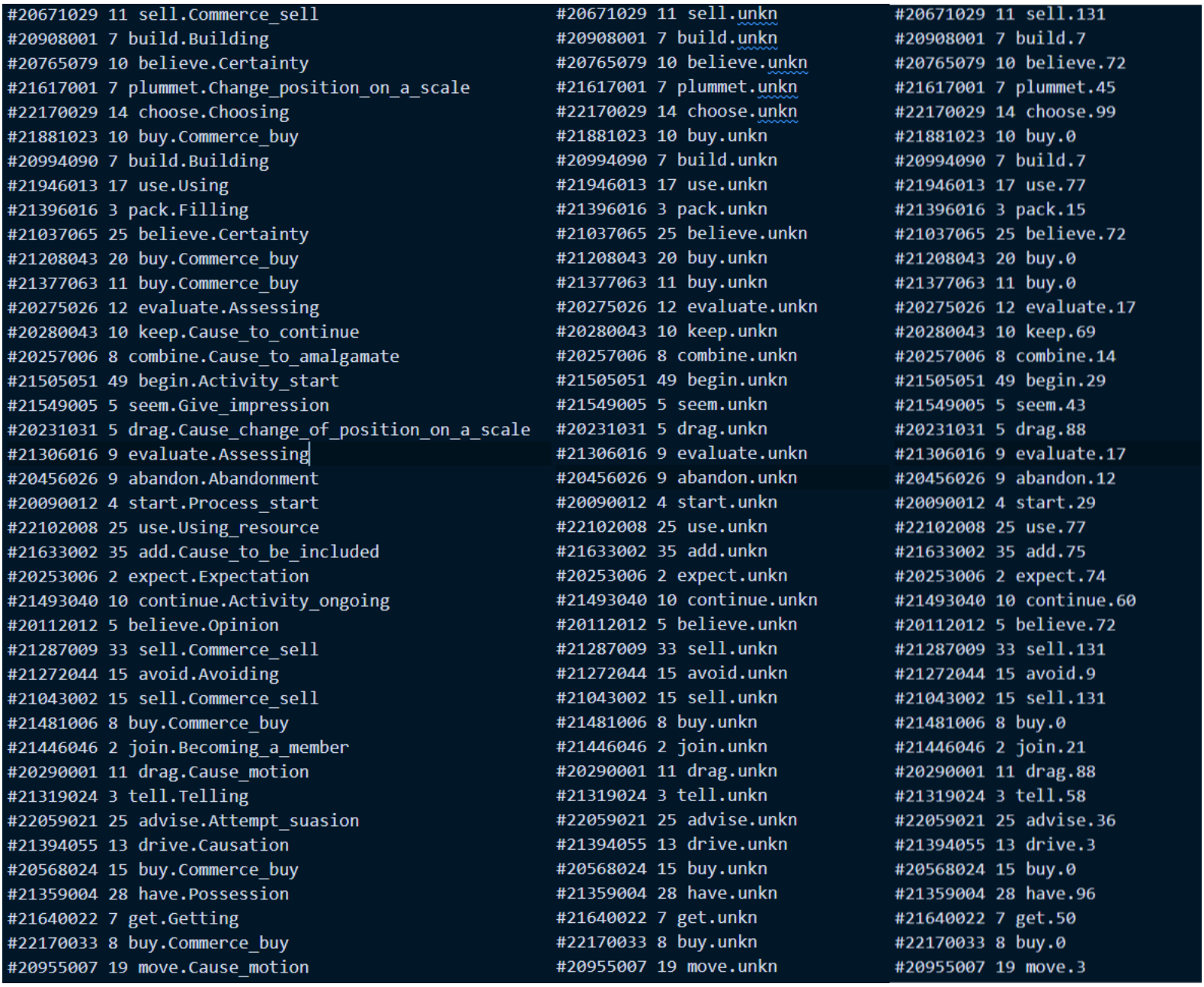
This is the final stage of the pipeline. There are a few things that happen here, first we receive the vectors created at the previous stage. Then we run the clustering algorithm on them and obtain the predicted clusters. The next thing to do is to create the submission file by filling in the prediction clusters. Lastly, we run the scorer program on the just created submission file and gather the results.

All these steps are implemented by a single function. But as there are some differences between the formats of the results for each task we decided to create a function to serve for each of the tasks. Therefore, we distinguish between 4 functions: `generate_submission_task1()`, `generate_submission_task2_2()`, `generate_submission_task2_2()`, `generate_submission_task2_2_per_frame()` where the last two address the same task B.1 but with a slightly different approach which we will discuss later. We will go now over the things that are in common for all these function and in the end we will emphasize on the differences.

We start off by initializing the clustering model. We choose to go with the agglomerative clustering model offered by the sklearn library. The api let's us set the number of clusters, affinity and linkage at the initialization of the clustering object which then can be used to "fit_predict()" on the vectors. The result of calling this function on the vectors is an array which assigns to each vector sample a label in the range 0 to $number\_of\_clusters$ (specified at the initialization) which identifies the cluster it belongs to. For Task A we also added the option of normalizing the vectors using another function from the sklearn library for which we set the norm to be "l2".

Next, we have to generate the submission file. The input and the output file follow the same format rules which are specific to each of the three tasks. For Task A each line starts with the id of the sentence followed by the index of the verb in the sentence. Then in the end we have the verb and the frame type separated by a dot. For the gold annotated samples we have the actual name of the frame (these are used just by the scoring program). These annotations are replaced with the label "UKN" and become the input for our system. Finally, in the submission file the "UKN" labels are replaced with numbers denoting the cluster that sample belongs to (see Figure 5.5). For the task B.1 and B.2 the format changes a bit by adding the verb arguments too. The beginning of the lines is identical to the one described for Task A but to this we add the arguments in the form of word followed by its index in the sentence followed by the role in case of Task B.2 or slot in case of the Task B.1. For example if we consider the same first sample from Figure 5.5 for task B.1 we will have "#20671029 11 sell.Commerce_sell fund-:-2-:-Seller bond-:-14-:-Goods" and for Task B.2 "#20671029 11 sell.na fund-:-2-:-Agent bond-:-14-:-Theme". Notice that for Task B.2 the frame type is replaced by "na" as it does not impact the clusterization based on the roles. The

generation of the submission files is handled individually per task by the helper functions: `task1_df_to_file()`, `task2_2_df_to_file()` and `task2_1_df_to_file()`.



#20671029 11 sell.Commerce_sell	#20671029 11 sell.unkn	#20671029 11 sell.131
#20908001 7 build.Building	#20908001 7 build.unkn	#20908001 7 build.7
#20765079 10 believe.Certainty	#20765079 10 believe.unkn	#20765079 10 believe.72
#21617001 7 plummet.Change_position_on_a_scale	#21617001 7 plummet.unkn	#21617001 7 plummet.45
#22170029 14 choose.Choosing	#22170029 14 choose.unkn	#22170029 14 choose.99
#21881023 10 buy.Commerce_buy	#21881023 10 buy.unkn	#21881023 10 buy.0
#20994090 7 build.Building	#20994090 7 build.unkn	#20994090 7 build.7
#21946013 17 use.Using	#21946013 17 use.unkn	#21946013 17 use.77
#21396016 3 pack.Filling	#21396016 3 pack.unkn	#21396016 3 pack.15
#21037065 25 believe.Certainty	#21037065 25 believe.unkn	#21037065 25 believe.72
#21208043 20 buy.Commerce_buy	#21208043 20 buy.unkn	#21208043 20 buy.0
#21377063 11 buy.Commerce_buy	#21377063 11 buy.unkn	#21377063 11 buy.0
#20275026 12 evaluate.Assessing	#20275026 12 evaluate.unkn	#20275026 12 evaluate.17
#20280043 10 keep.Cause_to_continue	#20280043 10 keep.unkn	#20280043 10 keep.69
#20257006 8 combine.Cause_to_amalgamate	#20257006 8 combine.unkn	#20257006 8 combine.14
#21505051 49 begin.Activity_start	#21505051 49 begin.unkn	#21505051 49 begin.29
#21549005 5 seem.Give_impression	#21549005 5 seem.unkn	#21549005 5 seem.43
#20231031 5 drag.Cause_change_of_position_on_a_scale	#20231031 5 drag.unkn	#20231031 5 drag.88
#21306016 9 evaluate.Assessing	#21306016 9 evaluate.unkn	#21306016 9 evaluate.17
#20456026 9 abandon.Abandonment	#20456026 9 abandon.unkn	#20456026 9 abandon.12
#20090012 4 start.Process_start	#20090012 4 start.unkn	#20090012 4 start.29
#22102008 25 use.Using_resource	#22102008 25 use.unkn	#22102008 25 use.77
#21633002 35 add.Cause_to_be_included	#21633002 35 add.unkn	#21633002 35 add.75
#20253006 2 expect.Expectation	#20253006 2 expect.unkn	#20253006 2 expect.74
#21493040 10 continue.Activity_ongoing	#21493040 10 continue.unkn	#21493040 10 continue.60
#20112012 5 believe.Opinion	#20112012 5 believe.unkn	#20112012 5 believe.72
#21287009 33 sell.Commerce_sell	#21287009 33 sell.unkn	#21287009 33 sell.131
#21272044 15 avoid.Avoiding	#21272044 15 avoid.unkn	#21272044 15 avoid.9
#21043002 15 sell.Commerce_sell	#21043002 15 sell.unkn	#21043002 15 sell.131
#21481006 8 buy.Commerce_buy	#21481006 8 buy.unkn	#21481006 8 buy.0
#21446046 2 join.Becoming_a_member	#21446046 2 join.unkn	#21446046 2 join.21
#20290001 11 drag.Cause_motion	#20290001 11 drag.unkn	#20290001 11 drag.88
#21319024 3 tell.Telling	#21319024 3 tell.unkn	#21319024 3 tell.58
#22059021 25 advise.Attempt_suasion	#22059021 25 advise.unkn	#22059021 25 advise.36
#21394055 13 drive.Causation	#21394055 13 drive.unkn	#21394055 13 drive.3
#20568024 15 buy.Commerce_buy	#20568024 15 buy.unkn	#20568024 15 buy.0
#21359004 28 have.Possession	#21359004 28 have.unkn	#21359004 28 have.96
#21640022 7 get.Getting	#21640022 7 get.unkn	#21640022 7 get.50
#22170033 8 buy.Commerce_buy	#22170033 8 buy.unkn	#22170033 8 buy.0
#20955007 19 move.Cause_motion	#20955007 19 move.unkn	#20955007 19 move.3

Figure 5.5: Example of the gold (left), test (middle) and result (right) files

Lastly we are going to talk about the scorer program. It is provided by the organizers in order to systematically evaluate the participants systems such that the obtained results could be compared to one another. The scorer is implemented in java and it comes as a java executable file which requires the machine to be equipped with java runtime 1.8 in order to work. In order to use the scorer for Task A we type the following command in the console ”java -cp EvaluationCodesSemEval2019Task2.jar semeval.run.Task1 path-to-gold-file path-to-submission-file -verbose” where the last argument ”-verbose” is optional (tells the scorer to print baselines). Similarly, for Task B.1 the command is ”java -cp EvaluationCodesSemEval2019Task2.jar semeval.run.Task21 path-to-gold-file path-to-

submission-file" and for Task B.2 "java -cp EvaluationCodesSemEval2019Task2.jar semeval.run.Task22 path-to-gold-file path-to-submission-file". In Figure 5.6 we can observe the format of the output of the scorer program for every of the three tasks.

```
Task1...
['elmo_context_elmo_word', 155, 'euclidean', 'complete', True, None]
----------------------------------------Evaluation----------------------------------------
       #goldInstance   #sysInstance    #goldClusterNum #sysClusterNum  purity  inversePurity   puIpuf1 BCubed-PrecisioB
Cubed-Recall    BCubed-f1
your-submission 4620    4620    149     155     78.9    77.19   78.03   70.94   70.76   70.85
Task2_1...
['features_only', ['_frame', '_before'], 315, 'euclidean', 'complete', None]
----------------------------------------Evaluation----------------------------------------
       #goldInstance   #sysInstance    #goldClusterNum #sysClusterNum  purity  inversePurity   puIpuf1 BCubed-PrecisioB
Cubed-Recall    BCubed-f1
your-submission 9510    9510    436     315     62.75   70.19   66.26   51.63   61.51   56.14
Task2_2...
['elmo_context_elmo_word', ['_xpos'], 5, 'euclidean', 'ward', None]
----------------------------------------Evaluation----------------------------------------
       #goldInstance   #sysInstance    #goldClusterNum #sysClusterNum  purity  inversePurity   puIpuf1 BCubed-PrecisioB
Cubed-Recall    BCubed-f1
your-submission 9466    9466    32      5       56.02   67.04   61.04   35.49   58.43   44.16
```

Figure 5.6: Example of running the scorer program for the three tasks

The reason behind the fact that there are two functions for generating the submission for task B.1 is that one of them applies the clustering algorithm over all the arguments (`generate_submission_task2_2()`) and the other applies the algorithm over arguments belonging to the same frame (`generate_submission_task2_2_per_frame()`). This is because unlike Task B.2 where arguments are clustered in generic role types, here in Task B.1 the arguments are grouped into slots which are specific to the frame. In other words, two frames can have a different number of slots and the slots from one frame are independent of the ones from the other frame. For this reason, we expect that clustering the arguments per frame will yield better results as it makes much more sense to do like so.

5.6. Hyperparameter Optimization

At this stage of the implementation we have all the necessary logic implemented. We have generated the vector representations for the contexts and words, created vector representations for features extracted from the CoNLL-U format, implemented the logic for combining the embeddings with the features to generate vectors and lastly we made it possible to apply agglomerative clustering on these vectors and generate the submission files which in the end are evaluated using the scorer program. But we are still not finished yet. What remains to be done is to experiment with all the possible embeddings we have generated to see which one better models our data. More than this, we have mentioned when we talked about the clustering algorithm that it receives three parameters namely

the number of clusters, the affinity and the linkage. These are called hyperparameters and we need to find the best configuration for them as well. This process becomes unfeasible to be done manually if we consider the huge number of combinations for these hyperparameters and ways of generating the vectors there are. Therefore, we automatized this process in code by writing the functions present in the file "generate_multiple_submissions.py". In this file we first defined a list per task in which we stored all the embedding names for that task. The embedding name dictates how the vectors are created.

For Task A the embedding names are the following:

- bert_cased_context_pooled_output
- bert_cased_context_layer
- bert_uncased_context_pooled_output
- bert_uncased_context_layer
- elmo_context
- bert_cased_context_pooled_output_bert_cased_word
- bert_cased_context_pooled_output_elmo_word
- bert_cased_context_pooled_output_word2vec_word
- bert_cased_context_layer_bert_cased_word
- bert_cased_context_layer_elmo_word
- bert_cased_context_layer_word2vec_word
- bert_uncased_context_pooled_output_bert_uncased_word
- bert_uncased_context_pooled_output_elmo_word
- bert_uncased_context_pooled_output_word2vec_word
- bert_uncased_context_layer_bert_uncased_word
- bert_uncased_context_layer_elmo_word
- bert_uncased_context_layer_word2vec_word
- elmo_context_elmo_word
- elmo_context_bert_cased_word
- elmo_context_bert_uncased_word
- elmo_context_word2vec_word

For Task B.2 the embedding names are:

- bert_cased_context_pooled_output_elmo_word
- bert_cased_context_pooled_output_word2vec_word
- bert_cased_context_layer_elmo_word
- bert_cased_context_layer_word2vec_word
- bert_uncased_context_pooled_output_elmo_word
- bert_uncased_context_pooled_output_word2vec_word
- bert_uncased_context_layer_elmo_word
- bert_uncased_context_layer_word2vec_word
- elmo_context_elmo_word
- elmo_context_word2vec_word
- bert_cased_word
- bert_uncased_word
- elmo_word

- features_only

And for Task B.1 the embedding names are:

- bert_cased_context_layer_elmo_word
- bert_uncased_context_layer_elmo_word
- elmo_context_elmo_word
- bert_cased_context_layer_elmo_verb_elmo_word
- bert_uncased_context_layer_elmo_verb_elmo_word
- elmo_context_elmo_verb_elmo_word
- bert_cased_word
- bert_uncased_word
- elmo_word
- features_only

The features that we used in case of the Task B.1 and B.2 are the following (note that it could be none of them, only one of them or a combination of any of them):

- XPOS
- DEPREL
- Before
- Pos
- Frame

As for the hyperparameters, the number of clusters varies in relation to the task but as a landmark we used the number specified by the scorer program which for Task A is 149, for Task B.1 436 and for Task B.2 32. The affinity was set to one of the following:

- Euclidean
- Manhattan
- Cosine

The linkage parameter also takes one of the values:

- Single
- Complete
- Average
- Ward (this type of linkage works only with Euclidean affinity)

After running the scorer for all the combinations we obtained for Task A 4662 results, for Task B.1 5522 and for Task B.2 1462. We will go over the analysis of these results in the next chapter.

5.7. Evaluation Metrics

In this section we will go over the evaluation metrics implemented by the scorer program. There are many different metrics used to asses the quality of a clustering, but we will focus only on two of them being also the ones used by the scorer program. Nevertheless, we present in Figure 5.7 a short comparison between 5 evaluation metrics. Now we are going to dive deeper into metrics based on set matching and BCubed metrics.

5.7.1. Metrics Based On Set Matching

The metrics which fall in this category are probably the most popular measures for cluster evaluation namely Purity, Inverse Purity and their harmonic mean. Purity focuses on the frequency of the most common category into each cluster while Inverse Purity focuses on the cluster with maximum recall for each category. In order to obtain a more robust metric we can combine the concepts of Purity and Inverse Purity resulting in the F measure.

If we consider C to be the set of clusters, L the set of categories and N the total number of clustered items we can define the Purity as in Eq. 5.1.

$$\text{Purity} = \sum_i \frac{|C_i|}{N} \max_j \text{Precision}(C_i, L_j) \tag{5.1}$$

Where the precision of a cluster C_i for a given category L_j is defined by Eq. 5.2

$$\text{Precision}(C_i, L_j) = \frac{|C_i \cap L_j|}{|C_i|} \tag{5.2}$$

Given the same notations we can define Inverse Purity by Eq. 5.3.

$$\text{Inverse Purity} = \sum_i \frac{|L_i|}{N} \max_j \text{Precision}(L_i, C_j) \tag{5.3}$$

Lastly we define the combined F measure by Eq. 5.4

$$\text{F} = \sum_i \frac{|L_i|}{N} \max_j F(L_i, C_j) \tag{5.4}$$

Where $F(L_i, C_j)$ is given by Eq. 5.5 and $\text{Recall}(L_i, C_j)$ by Eq. 5.6

$$F(L_i, C_j) = \frac{2 \times \text{Recall}(L_i, C_j) \times \text{Precision}(L_i, C_j)}{\text{Recall}(L_i, C_j) + \text{Precision}(L_i, C_j)} \tag{5.5}$$

$$\text{Recall}(L, C) = \text{Precision}(C, L) \tag{5.6}$$

Considering the first example (Cluster homogenity leftside) from Figure 5.7 the result obtained for Purity is given by Eq. 5.7 and for Inverse Purity by Eq. 5.8.

$$\text{Purity} = \frac{4 \cdot \frac{4}{4} + 3 \cdot \frac{2}{3} + 7 \cdot \frac{4}{7}}{14} = 0.71 \tag{5.7}$$

$$\text{Inverse Purity} = \frac{5 \cdot \frac{4}{5} + 6 \cdot \frac{4}{6} + 1 \cdot \frac{1}{1} + 1 \cdot \frac{1}{1} + 1 \cdot \frac{1}{1}}{14} = 0.78 \tag{5.8}$$

The downside of these metrics is that even the more robust one of them, the F metric still fails to satisfy two of the four formal constraints as we can observe in Figure 5.7.

5.7.2. BCubed Metrics

BCubed precision and recall metrics satisfy all of the four formal constraints when combined together. The way they work is by decomposing the evaluation process estimating the precision and recall associated to each item in the distribution. Therefore, the item precision represents how many items in the same cluster belong to its category while the item recall represents how many items from its category appear in its cluster.

If we consider the functions $L(e)$ (denotes the category of item e) and $C(e)$ (denotes the cluster of item e) we can define the correctness of the relation between two items e and e' form the distribution as in Eq. 5.9 (meaning that two items are correctly related when they share a category if and only if they appear in the same cluster).

$$\text{Correctness}(e, e') = \begin{cases} 1 \text{ iff } L(e) = L(e') \longleftrightarrow C(e) = C(e') \\ 0 \text{ otherwise} \end{cases} \tag{5.9}$$

With that in mind we can now define the precision (Eq. 5.10) and recall (Eq. 5.11).

$$\text{Precision BCubed} = \text{Avg}_e[\text{Avg}_{e'.C(e)=C(e')}[\text{Correctness}(e, e')]] \tag{5.10}$$

$$\text{Recall BCubed} = \text{Avg}_e[\text{Avg}_{e'.L(e)=L(e')}[\text{Correctness}(e, e')]] \tag{5.11}$$

Now we can define the metric F which combines the two metrics (precision and recall) by the Eq. 5.12 (fixing $\alpha = 0.5$ we obtain the harmonic mean of P and R).

$$F(R, P) = \frac{1}{\alpha(\frac{1}{P}) + (1 - \alpha)(\frac{1}{R})} \tag{5.12}$$

Considering the first example (Cluster homogenity leftside) from Figure 5.7 the result obtained for BCubed Precision is given by Eq. 5.13 and for BCubed Recall by Eq. 5.14.

$$\text{Precision BCubed} = \frac{(4 \cdot \frac{4}{4}) + (\frac{1}{3} + 2 \cdot \frac{2}{3}) + (3 \cdot \frac{1}{7} + 4 \cdot \frac{4}{7})}{14} = 0.59 \tag{5.13}$$

$$\text{Recall BCubed} = \frac{(4 \cdot \frac{4}{5}) + (\frac{1}{5} + 2 \cdot \frac{2}{6}) + (3 \cdot \frac{1}{1} + 4 \cdot \frac{4}{6})}{14} = 0.69 \tag{5.14}$$

Finally, the results of the testcases from Figure 5.7 prove the ability of the BCubedF metric to satisfy all the formal constraints, thus making it the most robust metric for measuring the quality of a clustering.

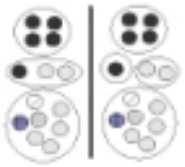
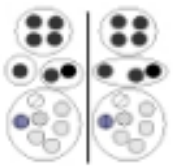
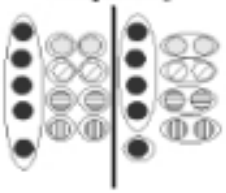
	Cluster homogeneity			Cluster completeness			Rag Bag			Cluster size vs. quantity		
												
Metrics based on set matching												
Purity	**0.71**	**0.78**	√	0.78	0.78	×	0.55	0.55	×	1	1	×
Inv. Purity	0.78	0.78	×	0.78	0.78	×	1	1	×	**0.69**	**0.92**	√
F-measure	0.63	0.63	×	0.62	0.62	×	0.61	0.61	×	**0.79**	**0.96**	√
	OK			**FAIL**			**FAIL**			**OK**		
Metrics based on counting pairs												
Rand	**0.68**	**0.7**	√	**0.68**	**0.7**	√	0.72	0.72	×	0.95	0.95	×
Jaccard	**0.31**	**0.32**	√	**0.31**	**0.35**	√	0.37	0.37	×	0.71	0.71	×
F&M	**0.47**	**0.49**	√	**0.47**	**0.52**	√	0.61	0.61	×	0.84	0.84	×
	OK			**OK**			**FAIL**			**FAIL**		
Metrics based on Entropy												
-Entropy	**-1.03**	**-0.8**	√	-0.83	-0.83	×	-1.29	-1.29	×	0	0	×
-Class Entr.	-0.65	-0.65	×	**-0.69**	**-0.49**	√	0	0	×	**-0.61**	**-0.28**	√
Mutual Inf.	**0.84**	**1.03**	√	1	1	×	0.99	0.99	×	2.19	2.19	×
-VI	**-1.68**	**-1.48**	√	**-1.52**	**-1.32**	√	-1.28	-1.28	×	**-0.61**	**-0.27**	√
	OK			**OK**			**FAIL**			**OK**		
Metrics based on editing distance												
Edit dist.	(steps) 7	7	×	**7**	**6**	√	6	6	×	**9**	**6**	√
	FAIL			**OK**			**FAIL**			**OK**		
BCubed metrics												
Precision	**0.59**	**0.69**	√	0.62	0.62	×	**0.52**	**0.64**	√	1	1	×
Recall	0.69	0.69	×	**0.71**	**0.75**	√	1	1	×	**0.64**	**0.81**	√
F(BCubed)	**0.63**	**0.69**	√	**0.66**	**0.67**	√	**0.68**	**0.78**	√	**0.78**	**0.89**	√
	OK			**OK**			**OK**			**OK**		

Figure 5.7: Comparison of different evaluation metrics applied on different examples to showcase their ability to satisfy the four basic formal constraints [21]

Chapter 6. Testing and Validation

In what follows we will explain the metrics used to evaluate the results as well as the way in which this is done. Then we will take a closer look to the results we obtained during our experimenting phase and lastly we will put these results in perspective with the results obtained by other participants.

6.1. Evaluation

The performance metrics are the same for each of the three tasks and represent measures for evaluating text clustering techniques. They include the classic measures of Purity (PU), Inverse Purity (IPU) and their harmonic mean (PIF) along with the harmonic mean for BCubed precision and recall (i.e BCP, BCR and BCF, respectively). The goal of these measures is to reflect a notion of similarity between the distribution of unsupervised labels and that of the gold reference labels. To be more precise, they define the notion of consistency and completeness of automatically generated clusters based on the evaluation data. But as each of the methods measures consistency and completeness in its own way, using just one of them might result in lack of sufficient information needed for a clear understanding and analysis of the system performance. Despite this, the organizers have agreed on the BCF measure to be the single metric for system ranking because of its satisfactory behaviour in certain situations.

The evaluation is performed by the scorer program provided by the organizers of the competition. As explained in Chapter 5 the scorer comes as a java executable file which can be run by providing the path to gold annotated file and the path to the generated submission (example of the files is shown in Figure 5.5) and as output it prints the values for the metrics presented earlier (in Figure 5.6 for Task A the value of BCF is 70.85).

The organizers also developed baseline systems to have a threshold for comparing the results. They used baseline of random, all-in-one-cluster ($AIN1$) and one-cluster-per-instance ($1CPI$). Even more, they have adapted one of the baselines used for WSI task called the most frequent sense. this was done by introducing the one-cluster-per-head ($1CPH$) baseline for Task A and one-cluster-per-syntactic-category ($1CPG$) for verb argument clustering in Task B.2. For Task B.1 the baseline was built as $1CPGH$ for labeling verbs with their lemmas (similar to $1CPH$) and the frame elements with grammatical relation to their heads (as in $1CPG$). In the test data most of the verbs frequently instantiate one particular frame and rarely other ones. Same goes for the roles where a particular role frequently is filled by words that have a particular grammatical relation to its governing verb (i.e. most subjects of most verb forms receive the "agent" label in their sub-categorization frame or agent-like element in their Frame Semantics representations). This generates a long-tailed distribution of the frequency of the test data and thus make the $1CPH$ and $1CPG$ baselines and their combinations for Task B.1

quite hard to beat. It is also mentioned the employment of one unsupervised and one supervised system baseline, both being trained out-of-the-box with no additional tuning.

6.2. Results

We will now present the results obtained by our system. As stated before, during the testing phase we experimented with multiple different combinations of embeddings generated by the three models used BERT, ELMo and Word2Vec. We also used agglomerative clustering which also bring to the table three hyperparameters for us to optimize namely number_of_clusters, affinity and linkage. We will now go over each of the task and analyze the results to point out which methods yielded the best performance.

6.2.1. Task A

In Task A we are required to group verbs to frame type clusters. The frame type is evoked by the appearance of a verb in a given context. Therefore, it is quite obvious that the vectors should be based on the context embeddings or a combination of both context and verb embeddings.

After analyzing the results we have made a ranking based on the BCF score of the embedding methods. This ranking is displayed in Table 6.1 (column "puIpuf1" represents the PIF metric which denotes the harmonic mean of the Purity and Inverse Purity metrics) where for each embedding type we show the best score obtained with any of the hyperparameters values. The ranking confirms us that vectors containing both context and verb embeddings outperform by far methods based only on context embeddings. The embedding names of BERT which contain the "layer" in the name refer to a group of three types of embeddings generated from layers 3, 6 and 9. Out of these the best performance were given by the embeddings from layer 3 which even outperformed the "pooled_output" layer. In terms of word embeddings of the verbs it seems that the ELMo model does the best job generating them.

Following in the Table 6.2 we display the best performing embeddings along with the combination of the hyperparameters used. In all cases the results benefit from normalizing the vectors before running the clustering algorithm on them. As for the linkage is appears that complete takes the lead along with the Euclidean and Manhattan distance metrics.

6.2.2. Task B.2

For Task B.2 the goal is to cluster the arguments of verbs to generic roles. For this task, the clustering to frames done in Task A is of no use. Being generic roles they don't really depend on the verb and thus using the verb embedding won't bring much to the table. Considering these, our approach for this task is similar with the one from Task A but with the difference that now we replaced the verb embeddings with the embeddings of the arguments. We have also added the use of the features mentioned in Chapter 5.

We analyzed the results in the same way we did for Task A and so we made a ranking of the best scores obtained per embedding type. The results are presented in Table 6.3.

To better visualize the methods that yield the best results we display the embedding name along with the features used and the selection of the hyperparameters values

Table 6.1: Performance in relation with the embedding types Task A

Embedding Name	puIpuf1 (PIF)	BCubed-f1 (BCF)
elmo_context_elmo_word	78.03	70.85
bert_cased_context_layer_elmo_word	77.72	70.28
bert_uncased_context_layer_elmo_word	77.68	70.00
bert_uncased_context_pooled_output_elmo_word	76.92	68.21
bert_cased_context_pooled_output_elmo_word	75.57	67.28
bert_cased_context_pooled_output_word2vec_word	61.88	49.54
bert_cased_context_pooled_output_bert_cased_word	57.07	44.18
bert_cased_context_layer_word2vec_word	55.10	41.96
bert_cased_context_layer_bert_cased_word	55.66	40.89
bert_uncased_context_layer_word2vec_word	51.30	37.36
elmo_context_bert_cased_word	50.48	34.62
elmo_context_word2vec_word	48.14	33.74
bert_uncased_context_layer_bert_uncased_word	47.65	33.30
elmo_context_bert_uncased_word	43.78	27.92
bert_uncased_context_pooled_output_bert_uncased_word	39.56	26.61
bert_uncased_context_pooled_output_word2vec_word	36.13	22.94
bert_cased_context_layer	34.52	19.04
elmo_context	32.29	17.94
bert_uncased_context_layer	32.04	17.26
bert_uncased_context_pooled_output	28.88	14.15
bert_cased_context_pooled_output	29.55	14.12

in Table 6.4. We notice how for this task ward linkage along with Euclidean affinity dominates the top. Also the best results are obtained with quite a low number of cluster of 5 given that in the gold annotations there are 32.

6.2.3. Task B.1

Task B.1 asks for the arguments of the verb to be clustered to frame-specific slots. Since in this case the slot is dependent of the frame evoked by the verb we will make use of the labels predicted in Task A. As for the embeddings we used the same as for Task B.2 with the addition of an embedding containing the context, the verb and the argument. The features also remain the same but we also consider the frame labels generated in Task A.

After analyzing the results we again made the ranking of the embeddings which is displayed in Table 6.5. As it turns out, the best results are obtained using embedding of just the features (i.e. frame, before, pos, XPOS, DEPREL). At first it would seem that this method is by far the best leaving a big gap between it and the other methods. But when we take a closer look to the actual labels predicted we understand that all it is doing is to give the same one label to all the arguments before the verb and another to all that are after the verb and does this for each frame. This is because it only takes advantage of the frame feature along with the Before one. Even though the performance is high, the predicted frames are trivial and with no use in the overall context of the problem.

Now to take a better look of the best performance methods we extracted them in the Table 6.6. As for Task B.2 the combination of ward linkage and Euclidean affinity

Table 6.2: The methods that yield the best results Task A

Embedding	Affinity	Linkage	Number of clusters	Normalized	Layer	BCubed-F1
Elmo Context + Elmo Word	Euclidean	Complete	155	True	Default	70.85
Bert Cased Context + Elmo Word	Manhattan	Complete	160	True	3	70.28
Bert Uncased Context + Elmo Word	Euclidean	Complete	155	True	3	70.00

Table 6.3: Performance in relation with the embedding types Task B.2

Embedding Name	puIpuf1 (PIF)	BCubed-f1 (BCF)
elmo_context_elmo_word	61.04	44.16
bert_cased_context_layer_elmo_word	61.06	44.15
bert_uncased_context_layer_elmo_word	60.98	43.98
bert_uncased_context_pooled_output_elmo_word	60.69	43.83
bert_cased_context_pooled_output_elmo_word	60.65	43.82
bert_cased_context_layer_word2vec_word	52.84	38.33
elmo_context_word2vec_word	51.41	37.88
bert_uncased_context_layer_word2vec_word	51.08	36.66
bert_cased_context_pooled_output_word2vec_word	50.30	34.83
bert_uncased_context_pooled_output_word2vec_word	47.63	32.62

dominates the top.

Up to this point we have made the clustering over all the arguments. This made sense for Task B.2 but in the context of Task B.1 it would seem that a better idea is to cluster the arguments in groups by frame. With this in mind we used the same setup of embeddings but we made the clustering dependent on the frames. Te best result was achieved using only the embedding of the argument words generated by bert uncased model. The features used were DEPREL, Before and Pos along with the ward linkage and Euclidean affinity. The score obtained was 44.63 (BCF) which is below what we obtained previously. We expected this strategy to yield the best results and we still believe that it should. One of the reasons for the worse results could be the fact that the dataset used contains a relatively small number of samples which results in a sparse distribution of the arguments per frame.

6.3. Comparative Analysis Of The Results

In the previous section we presented and analyzed the results obtained through the system we developed. But this might not be enough to fully understand how good or bad the results are. Therefore, we are going to put our results in perspective with the baseline offered by the organizers and also with the results obtained by other participants. For this, we considered the results from three of the participants namely: Nikolay Arefyev [18], Saba Anwar [20] and Eugénio Ribeiro [19]. We display the results for each of the tasks in a separate table as follows: Task A - Table 6.7, Task B.1 - Table 6.8 and lastly Task B.2 - Table 6.9. One thing to be mentioned is that the results for Task B.1 and B.2

Table 6.4: The methods that yield the best results Task B.2

Embedding	Features	Affinity	Linkage	Number of clusters	Layer	BCubed-F1
Elmo Context + Elmo Word	Xpos	Euclidean	Ward	5	Default	44.16
Bert Cased Context + Elmo Word	Xpos Before Pos	Euclidean	Ward	5	3	44.15
Bert Uncased Context + Elmo Word	-	Euclidean	Ward	5	3	43.98
Elmo Context + Elmo Word	Deprel	Euclidean	Ward	5	Default	43.88
Bert Uncased Context + Elmo Word	Deprel Before Pos	Euclidean	Ward	5	Pooled Output	43.83
Bert Uncased Context + Elmo Word	Xpos Before Pos	Euclidean	Ward	5	Pooled Output	43.82

Table 6.5: Performance in relation with the embedding types Task B.1

Embedding Name	puIpuf1 (PIF)	BCubed-f1 (BCF)
features_only	66.26	56.14
bert_cased_context_layer_elmo_word	57.03	46.59
elmo_word	57.12	46.59
bert_uncased_context_layer_elmo_word	57.01	46.45
elmo_context_elmo_word	56.83	46.36
elmo_context_elmo_verb_elmo_word	52.94	43.44
bert_cased_context_layer_elmo_verb_elmo_word	54.51	43.10
bert_uncased_context_layer_elmo_verb_elmo_word	53.65	43.07
bert_uncased_word	49.59	41.59
bert_cased_word	49.89	41.37

by Arefyev have been obtained using a semi-supervised method which doesn't comply with the requirements of the competition (a fully unsupervised method is imposed) and thus the results are shown just as a reference.

Table 6.6: The methods that yield the best results Task B.1

Embedding	Features	Affinity	Linkage	Number of clusters	Layer	BCubed-F1
Elmo Word	Frame Xpos	Euclidean	Ward	5	Default	46.59
Bert Cased Context + Elmo Word	Frame Before Pos	Euclidean	Ward	5	3	46.59
Bert Uncased Context + Elmo Word	Frame Xpos Before Pos	Euclidean	Ward	5	3	46.45
Elmo Context + Elmo Word	Frame Deprel	Euclidean	Ward	5	Default	46.36

Table 6.7: Task A Results Comparison

System	Approach	BCubed-F1
Arefyev et al.	BERT tunned with Hearst-like patterns + Agglomerative Clustering	70.70
Anwar et al.	Word2Vec Context and Word embedding + Agglomerative Clustering	68.10
Ribeiro et al.	Elmo embeddings + Chinese Whispers	65.32
Our	Elmo context and verb embedding + Agglomerative Clustering	70.85
Baseline	-	65.35

Table 6.8: Task B1 Results Comparison

System	Approach	BCubed-F1
Arefyev et al.	Combined predicted labels from Task A (frames) and Task B2 (frame elements)	~~63.12~~
Anwar et al.	Combined predicted labels from Task A (frames) and Task B2 (frame elements)	49.49
Ribeiro et al.	ELMo embedding of the arguments + Chinese Whispers	42.75
Our	Predicted labels from Task A + other features + Agglomerative Clustering	46.59
Baseline	-	45.79

Table 6.9: Task B2 Results Comparison

System	Approach	BCubed-F1
Arefyev et al.	Logistic regression trained on BERT representations and several handcrafted features	~~64.09~~
Anwar et al.	Word2Vec Context and Word embedding + other features + Agglomerative Clustering	42.1
Ribeiro et al.	ELMo embedding of the arguments + Chinese Whispers	45.65
Our	ELMo embeddings of the context and argument + other features + Agglomerative Clustering	44.16
Baseline	-	39.03

Chapter 7. User's manual

7.1. System Requirements

The software requirements are the following:

- Python 3
- Python libraries specified in requirements file (run command "pip install -r requirements.txt")
- CUDA installed and configured (if GPU available)

There are no restrictions about the operating system as all the software necessary is available for any OS.

As far as the hardware requirements go there are the following:

- recommended 8GB RAM
- 30 GB free space on the disk (to accommodate the scripts, models and all the generated files)
- preferably a dedicated GPU (greatly improves the runtime of generating the embeddings)

7.2. User Guide

First thing we have to do is to make sure we have all the necessary files in the right structure. The initial structure of the root folder is shown in Figure 7.1. Now we will go over each of them to explain what it contains.

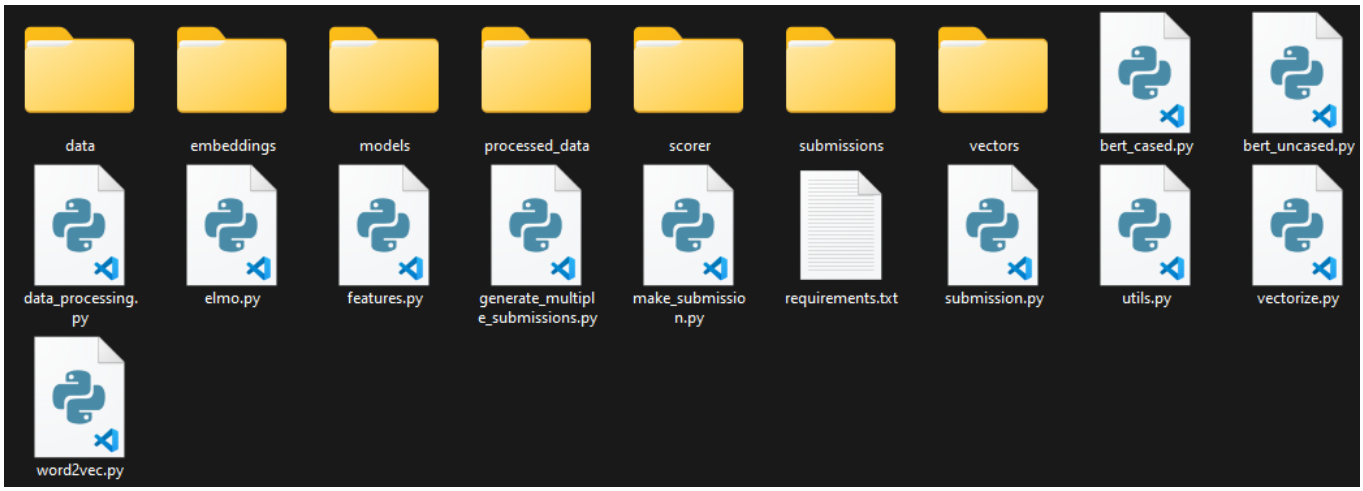


Figure 7.1: Initial structure of the root folder

- **data** This folder contains the data we are working with. The dataset can be found in folder ”dep-stx” while the other three folders ”dev”, ”gold” and ”test” will store the data files for the task.
- **embeddings** In this folder will be stored all the generated embeddings. There are 6 subfolder namely ”bert_cased”, ”bert_uncased”, ”elmo”, ”word2vec”, ”deprel” and ”xpos”. Each of the first 4 subfolder mentioned should contain 3 more subfolders ”context”, ”word” and ”verb” with the exception that ”word2vec” does not contain the ”context” subforlder.
- **models** This folder contains all the models used by the system. There are two bert models with their preprocess models another elmo model and lastry the word2vec model (GoogleNews).
- **processed_data** This folder will store preprocessed data in the two containg subfolder ”parsed_conll” and ”parsed_sentences”.
- **scorer** In this folder we will find the scorer program as ”.jar” file.
- **submissions** This folder will store the generated submission files for each of the tasks in the subfolders ”task_1”, ”task_2_1” and ”task_2_2”.
- **vectors** This folder will store the generated vectors for each of the tasks in the subfolders ”task1”, ”task2_1” and ”task2_2”.

Having that cleared out of the way we can move on and run the scripts for generating the necessary data. The first step is to run the preprocessing on the data with the following command.

```
python data_processing.py
python3 data_processing.py
```

When it finished we can check to see that the folder ”processed_data” has been populated. In both folders ”parsed_conll” and ”parsed_sentences” there should be 49,208 text files with the id as their name (i.e. ”#20001001.txt”).

Next thing to be done is to generate the embedding using each of the models. We will start with the bert_cased model. The following command should be run.

```
python bert_cased.py
python3 bert_cased.py
```

In the same way we should run the command to generate the embedding using the bert_uncased model.

```
python bert_uncased.py
python3 bert_uncased.py
```

Next we will generate the embeddings using the elmo model by running the next command.

```
python elmo.py
python3 elmo.py
```

Generating the embeddings is a very computational intensive task and thus it takes a good amount of time depending on the machine hardware resources (a good GPU can really make the difference here). While the embeddings are being generated the user is prompted with the current batch that is being worked on. After these commands are run the use can check the "context", "verb" and "word" folders from the folders of each embedding to see the generated files (context - 3346 files, verbs - 273 files, words - 3944 files).

We are left with generating the embedding using the word2vec model and this is done by running the following command.

```
python word2vec.py
python3 word2vec.py
```

Similarly, in the subfolders "verb" and "word" from the folder "word2vec" we should find 273 respectively 3944 generated files.

Next up we have to generate the embeddings for the features and we are doing this by running the next command.

```
python features.py
python3 features.py
```

As a result we should find in the "xpos" folder 35 generated files and in the subfolders "extended" and "primary" from the folder "deprel" we should find 91 respectively 35 generated files.

At this point we have all the necessary files to make predictions and then evaluate them with the scorer program. In order to run the system to generate and evaluate submissions we need to run the following command.

```
python make_submission.py task1_method_index task2_1_method_index
task2_2_method_index
python3 make_submission.py task1_method_index task2_1_method_index
task2_2_method_index
```

The different methods for each of the task are presented in Table 7.1 (Task A), Table 7.2 (Task B.1) and Table 7.3 (Task B.2). All these three tables contain the column "index" whose values map one to one to the actual method indexes expected as arguments by the program (i.e. Task A valid indexes are from 0-3, Task B.1 0-6 and Task B.2 0-5). The result of the scorer will be prompted for each of the three tasks. After running the command we can see the generated vectors in the "vectors" folder as well as the generated submission files in the "submissions" folder. A table visualization of the data will be generate as excel files in the folder "processed_data" for each individual task.

Table 7.1: Task A Methods

Index	Embedding Name	No_Clusters	Affinity	Linkage	Normalized	Layer
0	elmo_context_elmo_word	155	Euclidean	complete	True	Default
1	bert_cased_context_layer_elmo_word	160	Manhattan	complete	True	3
2	bert_cased_context_layer_elmo_word	165	Manhattan	complete	True	3
3	bert_uncased_context_layer_elmo_word	155	Euclidean	complete	True	3

Table 7.2: Task B1 Methods

Index	Embedding Name	Features	No_Clusters	Affinity	Linkage	Layer
0	features_only	frame before	315	Euclidean	complete	None
1	features_only	frame before pos	310	Euclidean	complete	None
2	features_only	frame deprel before	2	Euclidean	ward	None
3	elmo_word	frame xpos	5	Euclidean	ward	None
4	bert_cased_context_layer_elmo_word	frame before pos	5	Euclidean	ward	3
5	bert_uncased_context_layer_elmo_word	frame xpos before pos	5	Euclidean	ward	3
6	elmo_context_elmo_word	frame deprel	5	Euclidean	ward	None

Table 7.3: Task B2 Methods

Index	Embedding Name	Features	No_Clusters	Affinity	Linkage	Layer
0	elmo_context_elmo_word	xpos	5	Euclidean	ward	Default
1	bert_cased_context_layer_elmo_word	xpos before pos	5	Euclidean	ward	3
2	bert_uncased_context_layer_elmo_word	-	5	Euclidean	ward	3
3	elmo_context_elmo_word	deprel	5	Euclidean	ward	Default
4	bert_uncased_context_pooled_output_elmo_word	deprel before pos	5	Euclidean	ward	Pooled_output
5	bert_cased_context_pooled_output_elmo_word	xpos before pos	5	Euclidean	ward	Pooled_output

Chapter 8. Conclusions

8.1. Problem Description

In this research project we aimed to come up with a solution for the problem proposed in Task 2 from SemEval 2019 competition. The problem is decomposed into 3 tasks:

- **Task A** grouping verbs to frame type clusters.
- **Task B1** clustering arguments of verbs to frame-specific slots.
- **Task B2** clustering arguments of verbs to generic roles.

The goal is to develop a system to solve the tasks in a fully unsupervised way.

8.2. Proposed Solution

We proposed a system which addresses all the 3 tasks. To comply with the requirement of a fully unsupervised method, we chose to use for clustering the Agglomerative clustering algorithm and as for the data we only used the information extracted from the CoNLL-U format (the dataset is provided by the organizers in this format). For generating the word embeddings we used three models namely BERT, ELMo and Word2Vec.

8.3. Method Evaluation

In order to asses the system's performance we used the scorer program made available by the organizers. the metrics considered are the following:

- Purity, inverse-Purity and their harmonic mean.
- the harmonic mean of BCubed's Precision and Recall.

The latter being the one based on which the ranking of the system is done. For each of the task the organizers used an out-of-the-box to generate the baseline performance. In the end, we compared our results with the baselines as well as with the results obtain by other participants.

8.4. Contributions and Achievements

To summarize my contributions to the design and implementation of the system I will list them below:

- Analysis and preprocessing of the corpus available for the competition.

- Generate the embeddings for the context, verb and verb arguments using the models BERT, ELMo and Word2Vec.
- Selecting, extracting and embedding features from the CoNLL-U format of the dataset.
- Creation of the vectors fed to the clustering algorithm.
- Analysis of the different embedding methods.
- Optimization of the clustering algorithm hyperparameters.

As far as the achievements go, the system managed to obtain the best score for Task A while still obtaining competitive results which are well above baseline for the remaining two tasks.

8.5. Further Development

We have talked about our approach for Task B.1 of performing the clustering per frame slots and mentioned that we didn't manage to obtain better results. We consider this to be a result of the small size of the dataset that we are working with (being just a fraction of the whole Treebank 3.0). Thus, increasing the size of the dataset could result in a considerable performance leap.

Another strategy we can consider is to make use of a small set of annotated data. Even though this would result in a semisupervised approach to the problem (contrary to what the competition intended) it might see far better results. In fact this was actually proven by the system developed by Arefyev [18]. If we think that we only need a small amount of annotated data this can be feasible even for a large language corpus.

Lastly, we might be interested to see how the system would perform in case of other languages used. For this it will require to have a similar dataset in the same CoNLL-U format in order to incorporate morphological and syntactical annotations used for different features. It would also require the language models (BERT, ELMo and Word2Vec) to be trained beforehand on sample of the new language. With all this in mind, similar performance might be achieved for other languages as well.

Bibliography

[1] B. QasemiZadeh, M. R. L. Petruck, R. Stodden, L. Kallmeyer, and M. Candito, "SemEval-2019 task 2: Unsupervised lexical frame induction," in *Proceedings of the 13th International Workshop on Semantic Evaluation*. Minneapolis, Minnesota, USA: Association for Computational Linguistics, Jun. 2019, pp. 16–30. [Online]. Available: https://aclanthology.org/S19-2003

[2] K. Kipper, H. Dang, and M. Palmer, "Class-based construction of a verb lexicon," 01 2000, pp. 691–696.

[3] J. Ruppenhofer, M. Ellsworth, M. R. L. Petruck, C. R. Johnson, C. F. Baker, and J. Scheffczyk, "Framenet ii: Extended theory and practice," https://framenet.icsi.berkeley.edu/fndrupal/the_book, 2016.

[4] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations," in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*. New Orleans, Louisiana: Association for Computational Linguistics, Jun. 2018, pp. 2227–2237. [Online]. Available: https://aclanthology.org/N18-1202

[5] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2018. [Online]. Available: https://arxiv.org/abs/1810.04805

[6] M. Dutta, "Word2vec for word embeddings," https://www.analyticsvidhya.com/blog/2021/07/word2vec-for-word-embeddings-a-beginners-guide/, 2021.

[7] A. Wolfewicz, "Deep learning vs. machine learning," https://levity.ai/blog/difference-machine-learning-deep-learning, 2022.

[8] S. Mishra, "Unsupervised learning and data clustering," https://towardsdatascience.com/unsupervised-learning-and-data-clustering-eeecb78b422a, 2017.

[9] A. Amrami and Y. Goldberg, "Towards better substitution-based word sense induction," 2019. [Online]. Available: https://arxiv.org/abs/1905.12598

[10] M. P. Marcus, B. Santorini, and M. A. Marcinkiewicz, "Building a large annotated corpus of English: The Penn Treebank," *Computational Linguistics*, vol. 19, no. 2, pp. 313–330, 1993. [Online]. Available: https://aclanthology.org/J93-2004

[11] S. Buchholz and E. Marsi, "Conll-x shared task on multilingual dependency parsing," 01 2006.

[12] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013. [Online]. Available: https://arxiv.org/abs/1301.3781

[13] M. Eric, "Deep contextualized word representations with elmo," https://www.mihaileric.com/posts/deep-contextualized-word-representations-elmo/, 2018.

[14] K. Doshi, "Transformers explained visually (part 1): Overview of functionality," https://towardsdatascience.com/transformers-explained-visually-part-1-overview-of-functionality-95a6dd460452, 2020.

[15] R. Horev, "Bert explained: State of the art language model for nlp," https://towardsdatascience.com/bert-explained-state-of-the-art-language-model-for-nlp-f8b21a9b6270, 2018.

[16] C. Y. Wijaya, "Breaking down the agglomerative clustering process," https://towardsdatascience.com/breaking-down-the-agglomerative-clustering-process-1c367f74c7c2, 2019.

[17] C. Maklin, "Hierarchical agglomerative clustering algorithm," https://towardsdatascience.com/machine-learning-algorithms-part-12-hierarchical-agglomerative-clusterin 2018.

[18] N. Arefyev, B. Sheludko, A. Davletov, D. Kharchev, A. Nevidomsky, and A. Panchenko, "Neural GRANNy at SemEval-2019 task 2: A combined approach for better modeling of semantic relationships in semantic frame induction," in *Proceedings of the 13th International Workshop on Semantic Evaluation*. Minneapolis, Minnesota, USA: Association for Computational Linguistics, Jun. 2019, pp. 31–38. [Online]. Available: https://aclanthology.org/S19-2004

[19] E. Ribeiro, V. Mendonça, R. Ribeiro, D. Martins de Matos, A. Sardinha, A. L. Santos, and L. Coheur, "L2F/INESC-ID at SemEval-2019 task 2: Unsupervised lexical semantic frame induction using contextualized word representations," in *Proceedings of the 13th International Workshop on Semantic Evaluation*. Minneapolis, Minnesota, USA: Association for Computational Linguistics, Jun. 2019, pp. 130–136. [Online]. Available: https://aclanthology.org/S19-2019

[20] S. Anwar, D. Ustalov, N. Arefyev, S. P. Ponzetto, C. Biemann, and A. Panchenko, "HHMM at SemEval-2019 task 2: Unsupervised frame induction using contextualized word embeddings," in *Proceedings of the 13th International Workshop on Semantic Evaluation*. Minneapolis, Minnesota, USA: Association for Computational Linguistics, Jun. 2019, pp. 125–129. [Online]. Available: https://aclanthology.org/S19-2018

[21] E. Amigó, J. Gonzalo, J. Artiles, and M. Verdejo, "Amigó e, gonzalo j, artiles j et ala comparison of extrinsic clustering evaluation metrics based on formal constraints. inform retriev 12:461-486," *Information Retrieval*, vol. 12, pp. 461–486, 10 2009.