This document is part of EXCITEMENT project,
funded by the 7th Framework Programme of the European Commission through grant agreement no.: 287923.

EXploring Customer Interaction via Textual EntailMENT

Deliverable 6.2: Textual inference components development, III cycle – Draft!

Authors: Luisa Bentivogli, Kathrin Eichler, Aleksandra Gabryszak, Lili Kotlerman, Vivi Nastase, Tae-Gil Noh

Dissemination Level: Confidential

Date: December 17, 2014

Grant agreement no.	287923	
Project acronym	EXCITEMENT	
Project full title	Exploring Customer Interaction via Textual entailMENT	
Funding scheme	STREP	
Coordinator	Moshe Wasserblat (NICE)	
Start date, duration	1 January 2012, 36 months	
Distribution	Confidential	
Contractual date of delivery	31/12/2014	
Actual date of delivery	12/17/2014	
Deliverable number	6.2	
Deliverable title	Textual inference components development, III cycle	
Type	Program	
Status and version	Final	
Number of pages	82	
Contributing partners	DFKI, BIU, FBK, UHEI	
WP leader	DFKI	
Task leader	DFKI	
Authors	Luisa Bentivogli, Kathrin Eichler, Aleksandra Gabryszak, Lili Kotlerman, Vivi Nastase, Tae-Gil Noh	
EC project officer	Carola Carstens	
The partners in EXCITEMENT are:	Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI), Germany	
	NICE Systems, Israel	
	Fondazione Bruno Kessler (FBK), Italy	
	Bar-Ilan University, Israel	
	Heidelberg University, Germany	
	OMQ, Germany	
	ALMAWAVE, Italy	

For copies of reports, updates on project activities and other EXCITEMENT-related information, contact:

NICE Systems
EXCITEMENT
Nir Raz Nir.Raz@nice.com
Hapnina 8 Phone: +972 (9) 775-3013
Ra'anana, Israel Fax: +972 (9) 775-3702

Copies of reports and other material can also be accessed via http://www.excitement-project.eu

Table of Contents

1. Introduction

1.1 About this Document

Deliverable 6.2 is of type "P", i.e., a program. This document provides a description of this program. The actual source code described in this document has been made available to all project members. A zipped version of the current code and all data necessary to run it has been uploaded to the member area of the project's website in /Deliverables /Month 36/WP6/source code/. The code can also be found in the Transduction Layer github repository at https://github.com/hltfbk/Excitement-Transduction-Layer , which is currently accessible to all WP6 developers and to relevant WP7 developers. The TL code was also made accessible to WP7 as Maven artifact to facilitate the integration with the industrial systems.

1.2 Introduction to the Transduction Layer

The EXCITEMENT open platform (EOP) developed in WP4 provides the textual entailment capability to decide the entailment relation between pairs of given textual units. This entailment recognition capability itself, however, does not provide a complete solution to the needs of the industrial partners, who aim to use textual entailment for exploring customer interactions (WP7). Additional steps are required to break down the information need of the industrial partners into textual entailment problems and combine the entailment decisions returned by the EOP into a response to the information need. Therefore, we need an additional layer of services on top of the EOP to achieve inference-based exploration of customer interaction data. We call this the *Transduction Layer*.

The Transduction Layer was developed within WP6 for the following two industrial use case scenarios that have been defined within the EXCITEMENT project (see Deliverables 1.1 and 3.1b for more details on the use cases):

Use Case 1: Entailment graph creation:
In this use case, the aim is to draw an entailment graph from a set of given interactions.

Use Case 2: Interaction categorization
In this use case, the aim is to annotate matching categories on a given interaction, using entailment information.

An analysis of the companies' use case scenarios has shown that the Transduction Layer can be organized around two central steps, namely *decomposition*, i.e. converting the company's input into a set of entailment units, from which entailment decision problems can be created, and *composition*, i.e. building entailment pairs and processing entailment decisions to meet the company's information need. The decomposition part is shared by both use cases, the composition part differs.

This document is structured as follows. We first describe the main data flow for the decomposition step (shared by both use cases) and the composition step of use case 1 and use case 2, respectively (chapter 2). We then describe the core data structures (chapter 3) and the UIMA type system (chapter 4) we designed for the Transduction Layer. Chapter 5 contains interface definitions for all transduction layer modules: the interfaces for the core modules and the top level interfaces defined specifically for the industrial partners (WP7). This is followed by a chapter describing the implementation, including one or more implementations for each defined module (chapter 6). The document ends with a chapter on evaluation results (chapter 7).

1.3 Related Terminology

In this document, we will use the following terminology:

- *RTE*: Recognizing textual entailment
- *Use Case 1 / Use Case 2*: This refers to the two use cases introduced in the previous section.
- *EOP*: This refers to the EXCITEMENT open platform providing the RTE functionality.
- *EDA*: This refers to an entailment decision algorithm provided by the EOP, i.e. the part of the open platform that returns an entailment decision for a given text pair.
- *LAP*: This refers to an linguistic analysis pipeline provided by the EOP, i.e. the part of the open platform that creates a JCas object with linguistic annotations.
- *Entailment graph*: An entailment graph orders text units in a structured hierarchy based on the entailment relations that hold between these text units.

1.4 Related Documents

- *Deliverable 1.1*: User Requirements (June 2012)
- *Deliverable 3.1b*: Specification of the Transduction Layer (Sep 2012)
- *Deliverable 6.1*: Textual inference components development (June 2013)
- *EOP specification*: Specifications and architecture for the open platform, II. cycle
- *UIMA Documentation*: UIMA Tutorial and Developers' Guides (http://uima.apache.org/d/uimaj-2.4.0/tutorials_and_users_guides.html)

1.5 Changes as compared to Deliverable 6.1

As compared to the previous deliverable of WP6, this document has changed in the following ways:

- Chapter 2:
  - Based on a request made by one of the academic partners, we decided to rename the module "CollapsedGraphGenerator" to "GraphOpti-

mizer", reflecting in the name that the module actually does more than collapsing nodes (it also decides on edges to be kept in the output graph).

- Based on a request made by one of the industrial partners, we added a module (*ConfidenceCalculator*) for pre-calculating a final confidence score per category on each node of the entailment graph and for adding this information to the graph.

- Chapter 3:
  - We removed details of the graph structures. Detailed information of methods, attributes, etc. can be found in the JavaDoc of the respective classes.

- Chapter 5:
  - We added an interface for the *ConfidenceCalculator* module.

- Chapter 6:
  - We added top-level descriptions of new TL module implementations
  - We removed implementation details (methods, etc.). Detailed information of methods, attributes, etc. can be found in the JavaDoc of the respective classes.

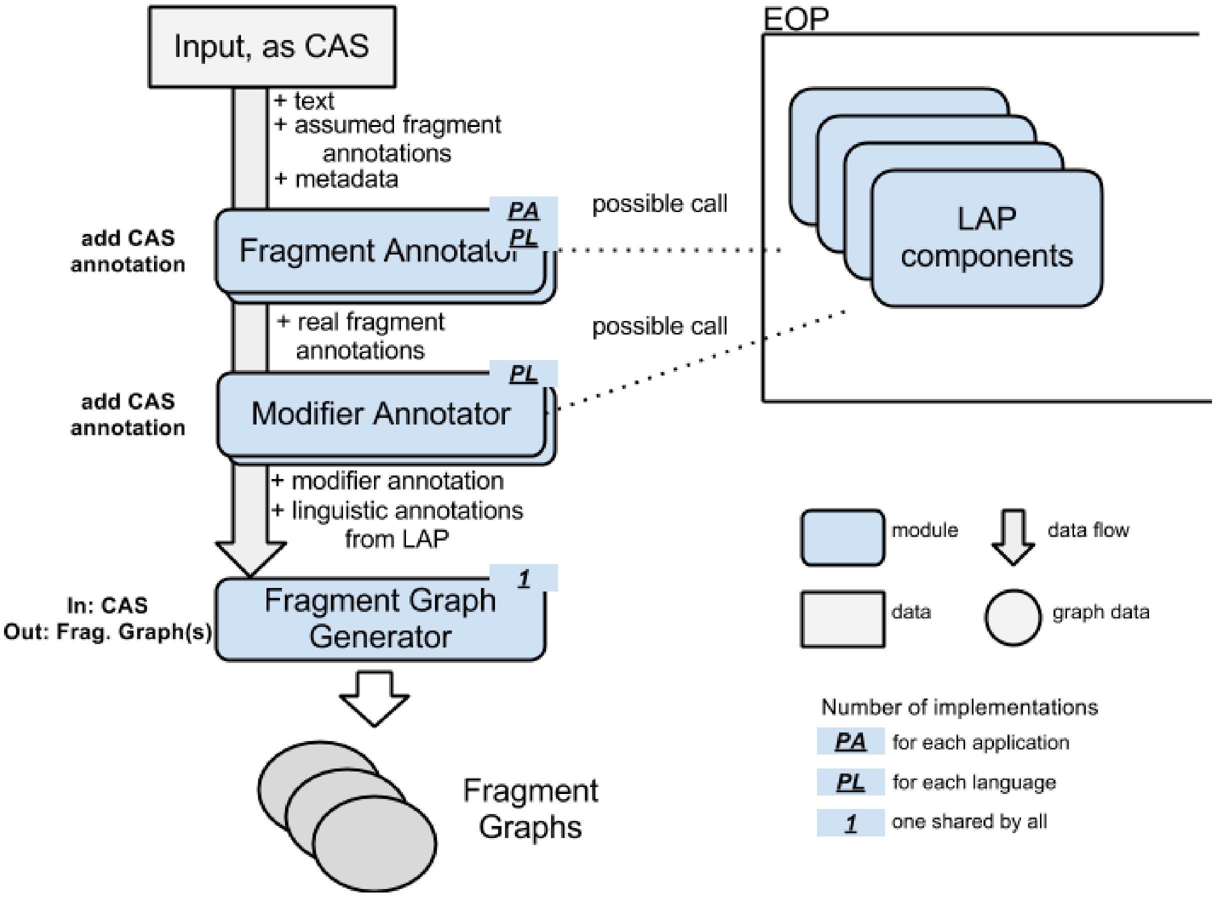
- Chapter 7:
  - We added a chapter describing the evaluations done within WP6, including the revision of industrial datasets, the evaluation of different EDA configurations on RTE-style data, and the evaluation of all TL modules.

2. Data Flow Overview

This section describes the main Transduction Layer data flows. It holds three subsections: Decomposition, Composition Use Case 1, and Composition Use Case 2.

2.1 Decomposition

The following figure outlines the data flow for the decomposition step and the modules that are part of this data flow. In the decomposition step, the input provided by the user (e.g., WP7) is processed (possibly using LAP components provided by the EOP) and turned into a set of fragments graphs. The decomposition step is relevant to both use cases: In use case 1, it is required because fragment graphs are the data structures, from which entailment graphs are built. In use case 2, it is required because an incoming email is annotated with categories by matching the fragment graphs extracted from the email against an existing entailment graph.



2.1.1 Data: Input Data

The input from the upper layer – referred to here as "input" – can be one customer interaction (in use cases 1 and 2), or a category description (in use case 2). We make the assumption that this input consists of *text* and (possibly) *assumed fragment* annotations. *text* refers to the actual (complete) text of the interaction or category description. The *assumed fragments* are annotations that span portions of the *text,* which are considered to express relevant content. One input can have any number of assumed fragment annotations, including none.

Within the Transduction Layer, the input is represented as a UIMA CAS. The *text* is given in the CAS's Sofa (Subject of Analysis), and each *assumed fragment* is given as a CAS annotation on the text. A specific type is used for this annotation.

The caller (WP7) can directly prepare this CAS. WP6 provides some helper functions that enable users to annotate fragments and modifiers (for details, refer to the following sections) without understanding the internals of CAS. If the WP7 input does not need to mark annotations (fragments or modifiers), the set of interactions can be passed in a simpler data type (String based List<Interaction>). For the actual interfaces, please see section 5.3.2.

2.1.2 Module: Fragment Annotator

A Fragment Annotator is a module that generates *determined fragment annotations*. By "determined" we mean fragment annotations determined by this module that are used in later TL steps.

There are several reasons for performing this additional fragment annotation step: (i) If there are no fragment annotations provided by the user; (ii) if the fragment annotations provided by the user are too broad, covering coordinate, subordinate or complex clauses (e.g. "The food was bad and the leg room was too small"). If no fragment annotations are provided, the module performs its own analysis of the text, and produces *determined fragment annotations*. If *assumed fragments* were given, the module iterates over them, and refines them if they are found to be complex expressions to produce the *determined fragment annotations*. The span of the determined fragment annotations may coincide with a fragment annotation provided by the user, or can cover contiguous or non-contiguous portions of the user's annotation.

The annotations produced are added to the input CAS, enriching the text's representation.

This module is application- and language-specific. The module may (need to) call an LAP, depending on the implementation. If it calls an LAP, it must consider future steps and try to minimize the need of future LAP calling.

2.1.3 Module: Modifier Annotator

After obtaining fragment annotations, the next step is to identify all modifiers within these fragments. The identified modifiers are annotated with a specific *modifier annotation type*. The words in a fragment that are not annotated as *modifier*, form the *base statement* (also called *base predicate* in WP2 terms). We simply keep one modifier annotation type, but no base statement annotation (non-modifier) type.

An implementation of this module marks all modifiers in the fragments. The module adds annotations to the given CAS, and does not generate any independent data.

We expect the modifier identification to be language specific, and thus language specific implementations of this module to be necessary.

The module may (need to) call an LAP, since detecting modifiers (probably) needs information of POS tags or more. When it calls LAP, it must consider future steps and try to minimize the need of future LAP calling.

2.1.4 Module: Fragment Graph Generator

This module consumes one CAS, and generates one or more *fragment graphs* (one fragment graph for each determined fragment).
The input CAS of this module has the following annotations at this stage.

- *[Group A]* Determined fragment annotation, modifier annotation
- *[Group B]* Linguistic annotations from LAP, Metadata from the user
- *[Others]* Assumed fragment annotation from the user

Note that the input CAS holds metadata from the interaction XML. This includes language, channel, provider, date, category, etc. To see the full list of metadata, please check the Metadata type definition (section 4.2.1). Please note that the language of the CAS is a special metadata and stored in the CAS itself. It can be accessed using a*JCas.setDocumentLanguage()* and *aJCas.getDocumentLanguage()*, respectively.

The fragment graphs are built based on the fragment and modifier annotations (group A). The fragment graph corresponding to a fragment is built by producing as nodes all combinations of base statement and modifiers, with entailment relation between nodes based on subsumption of the corresponding sets of covered modifiers. The annotations from group B are used to provide additional information that is stored in each node of a fragment graph, to be available in successive annotation steps. Other annotations (like assumed fragmentation) are not used.

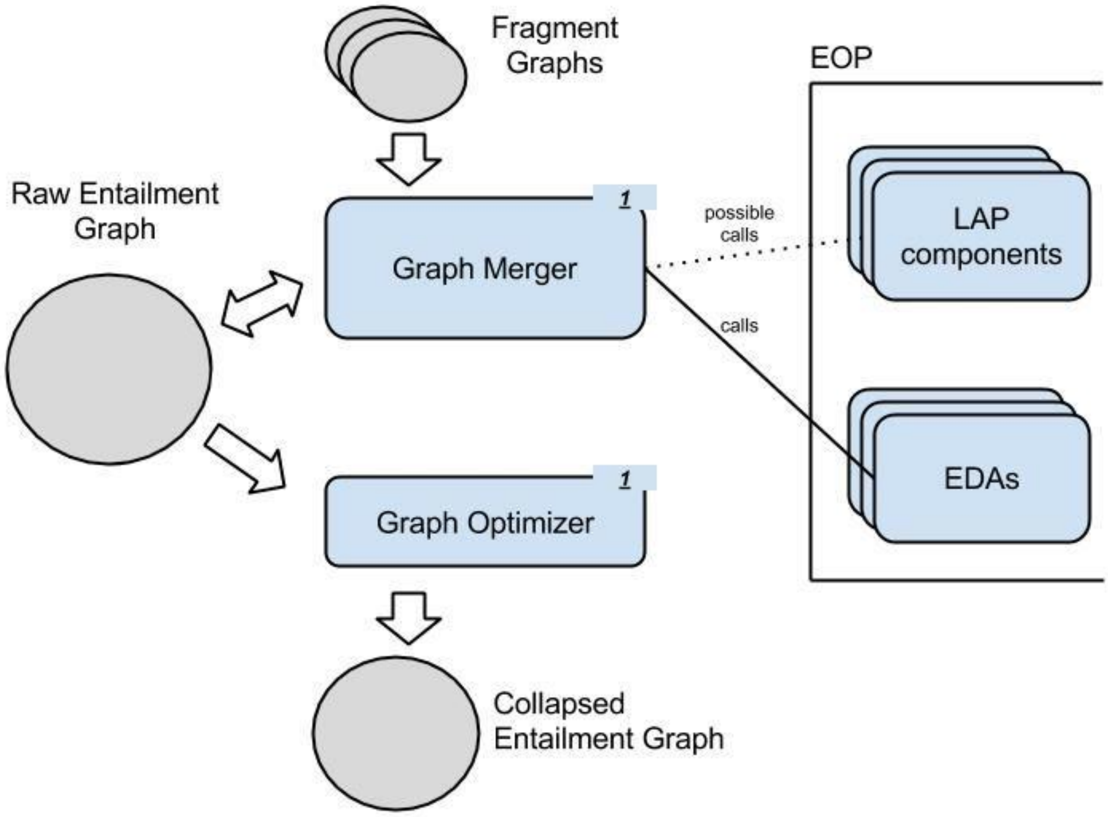
Each fragment graph is represented as a specifically designed Java object. This representation is detailed in section 3.3.1. As the input CAS may contain more than one determined fragment annotations, the output of this module is a set of fragment graphs (one per determined fragment).

When the CAS is consumed and the associated fragment graph(s) are built, one cycle of the decomposition flow is finished.

2.2 Composition Use Case 1

The following figure outlines the data flow of the composition step for use case 1 (entailment graph building) and the modules that work for this data flow. In this composition step, the fragment graphs created from the input data in the decomposition step are merged into a raw entailment graph (possibly calling LAP components and EDAs provided by the EOP) and collapsed to the final output (a collapsed graph).

For details about the Java objects representing the different graph types, see section 3.3.



2.2.1 Module: Graph Merger

This module builds or extends a raw entailment graph (also referred to as *raw graph*), by merging fragment graphs. It receives as input a raw graph (possibly empty), and a set of fragment graphs that are gradually added to the input raw graph. The result of this processing is a bigger, richer, version of the input raw entailment graph. For one instance of an industrial application there exists only one raw entailment graph, which grows with each run of the Graph Merger module.

Each raw graph is represented through a specially designed Java object. For this, see section 3.3.2.

To merge fragment graphs into the raw graph, the Graph Merger module uses the entailment decision capability of the EXCITEMENT open platform (EOP). For more information about the EOP, please see the EOP specification.

We expect this module to be application-independent. This means that the unique module implementation must provide a sound way of choosing the most fitting entailment decision algorithm (EDA) from the EOP.

To produce input data for the chosen EDA in an efficient manner (avoiding unnecessary LAP calls and creation of CAS objects), this module should try to reuse as much of the LAP annotations already attached to the nodes of the raw graph as possible.

2.2.2 Module: Graph Optimizer

This module trims an input raw graph (by selecting edges, creating equivalence classes, etc.) in order to produce a special version of the entailment graph – we call it *collapsed* graph – that is useful for the application scenario.

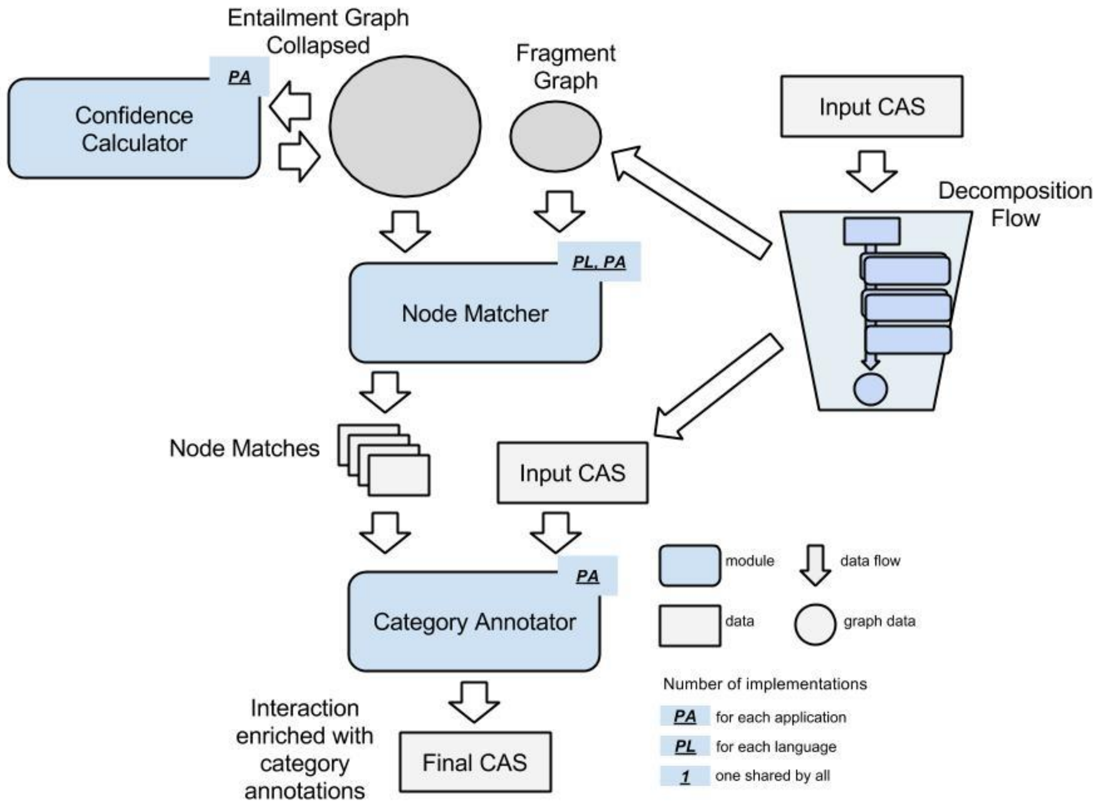
The raw graph essentially represents all Transduction Layer's knowledge about the entailment relations between its nodes. The purpose of the optimization procedure is to make final decisions on whether an entailment relation holds or not between the nodes, resolve transitivity violations and compress paraphrasing statements into equivalence class nodes.

Each collapsed graph is represented through a specially designed Java object. For this, see section 3.3.3.

The module is self-contained – it transforms the input raw graph into a collapsed graph without relying on external modules (such as the EOP) or data. A confidence threshold may be provided as an additional input parameter (e.g., by an industrial system), to customize the resulting collapsed graph by filtering entailment relations from the input raw graph based on their strength.

2.3 Composition Use Case 2

The following figure outlines the data flow of the composition step for use case 2 (category annotation), and the modules used for this data flow. In this composition step, the fragment graphs created from an incoming email are matched against an existing entailment graph. Extracting and combining category information from the matching graph nodes, the incoming email is then enriched with matching categories and their associated confidence scores.



2.3.1 Module: Confidence Calculator

This module reads category confidence scores stored in a collapsed graph, combines them to a final score per category per node and adds this information to the graph. It takes as input a collapsed graph containing category confidence scores and adds the combined confidence scores as additional information to the input graph.

This module was added following a request by OMQ. Precalculating the confidence scores, we make the actual matching step more efficient and avoid redundancy in the calculation of combined confidence scores. As a result, the confidence calculation in the *CategoryAnnotator* module was simplified to combining the final scores of different matching nodes. Unlike in the transduction layer prototype, the input fragment graph is now compared to the collapsed (not the raw) entailment graph.

This module is application-specific, as it depends on the algorithm used for combining category confidence scores to a single score. Thus, several implementations may be necessary. It does not need calls on external modules (like LAP or EOP), or stored data, other than the collapsed graph itself.

2.3.2 Module: Node Matcher

This module matches a given fragment graph *F* against a given collapsed entailment graph *C*. It returns a set of *node matches*, where each node match holds a node *M* (one of the nodes in *F*) associated to a set of *per node scores*. Each per node score is a tuple <*E*,*p*>, where *E* denotes a node in *C* and *p* denotes the confidence of *M* matching *E*. Node matches and per node scores are represented through specially designed Java objects. For details, refer to sections 5.2.4.1.3.1 and 5.2.4.1.3.2

The module aims for a fast (search-engine like) matching to produce results in near-real-time.

We expect this module to be language- and application-specific, thus several implementations may be necessary. It does not need calls on external modules (like LAP or EOP), or stored data, other than the fragment graph and the raw graph itself.

2.3.3 Module: Category Annotator

This module adds category annotation to a given input CAS. In addition to the input CAS, it takes as input the output of the Node Matcher module, i.e. a set of node matches for a particular fragment. From these node matches, it extracts the category confidence scores (computed using the Confidence Calculator module), combines the scores if needed, and adds the category confidence scores as new annotation to the fragment in the input CAS.

This module is application-specific.

There are no external dependencies expected for this module.

3. Core Data Structures

This chapter introduces the core data structures used in the Transduction Layer (TL): First, the data structure *Interaction*, which holds the input provided by the user, and, second, the graph data structures required for building entailment graphs.

For more details about the implementation of these data structures, please check the Java documentation in the code.

3.1 Interaction

3.1.1 class Interaction (eu.excitementproject.tl.structure)

This section describes the Interaction class, which represents one un-annotated interaction text and its metadata. Note that this data structure is a "boundary" data structure that is designed to get external input and translate it to the input CAS data type. The data type itself is not the main target of processing: actual processing like annotations and fragment graph building is always happening on the input CAS level.

3.2 Introduction to the Three Graphs

As we have seen in section 3, we have three conceptually different, graph-based data representations. One is *fragment graph*, which is a graph built based on the modifiers identified in the fragment. Another graph is the raw entailment graph (or *raw graph)*: this is the main entailment graph that is being kept and worked with in WP6. Major operations like adding edges and required EDA calling (for entailment judgment) is all done with this graph. Finally, the last graph is the so-called *collapsed graph* (the trimmed graph). This graph can be automatically generated from the raw graph (via a module).

To implement the graph structures we use the JGraphT library, which provides a rich and flexible inventory of graph types, as well as visualization functionalities. The JGraphT library (https://github.com/jgrapht/jgrapht) offers implementations for directed and undirected, weighted and unweighted simple- and multi-graphs.

3.3 Graph Data Structure in Detail

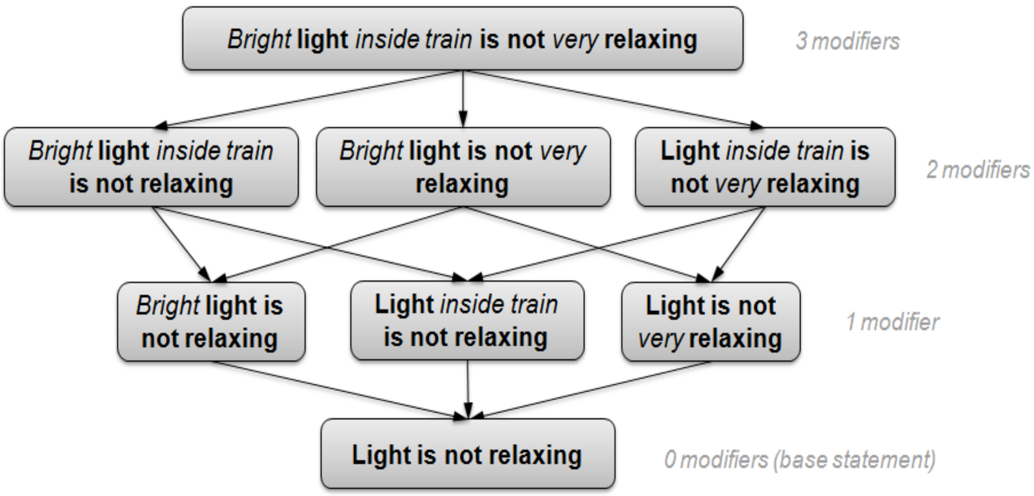
The choice for representing the three graph types as described in the following sections was driven by the structural and functional requirements for each of them.

3.3.1 Fragment Graph

The fragment graphs, with their corresponding class *FragmentGraph*, are simple graph structures. Their nodes contain much information, but structurally they are simple directed graphs. A directed edge – representing the entailment relation – connects the node corresponding to a text fragment T with a node corresponding to the text fragment T minus one of T's modifiers.

We assume a text fragment to be composed of a base statement (BS) plus a number of modifiers (M). A node of this graph corresponds to BS [+ M_1 ... M_k]. We assume a textual entailment (TE) relation (i.e. an edge in the graph) between every two statements (S_i, S_j) that differ only by one modifier: $S_i = S_j + M_x$ => S_i *entails* S_j.

An example of a fragment graph is presented below. Nodes hold text fragments with modifiers shown in italics.



3.3.1.1 class FragmentGraph (eu.excitementproject.tl.structures.fragmentgraph)

This class extends the DefaultDirectedWeightedGraph class, because the graph is directed and we might decide to have the edges weighted. Currently they are not. Please refer to the JavaDoc for the DefaultDirectedWeightedGraph class for information about inherited methods:

http://jgrapht.org/javadoc/org/jgrapht/graph/DefaultDirectedWeightedGraph.html

The nodes of the *FragmentGraph* are *EntailmentUnitMention*-s, and the edges are *FragmentGraphEdge*-s.

3.3.1.2 class EntailmentUnitMention (eu.excitementproject.tl.structures.fragmentgraph)

An *EntailmentUnitMention* refers to a piece of text (fragment or subfragment in WP2 terminology) occurring within a particular input text, i.e., it is associated to exactly one document. *EntailmentUnitMention*-s referring to the same text are grouped within a single *EntailmentUnit* (for details, see section 3.3.2.2). A vertex in the fragment graph is an *EntailmentUnitMention*. It consists of a base statement + (optionally) a number of modifiers.

3.3.1.3 class FragmentGraphEdge (eu.excitementproject.tl.structures.fragmentgraph)

This is the edge class for the FragmentGraph. For now, edges are directed, with default weight. This class extends DefaultEdge, documented here:

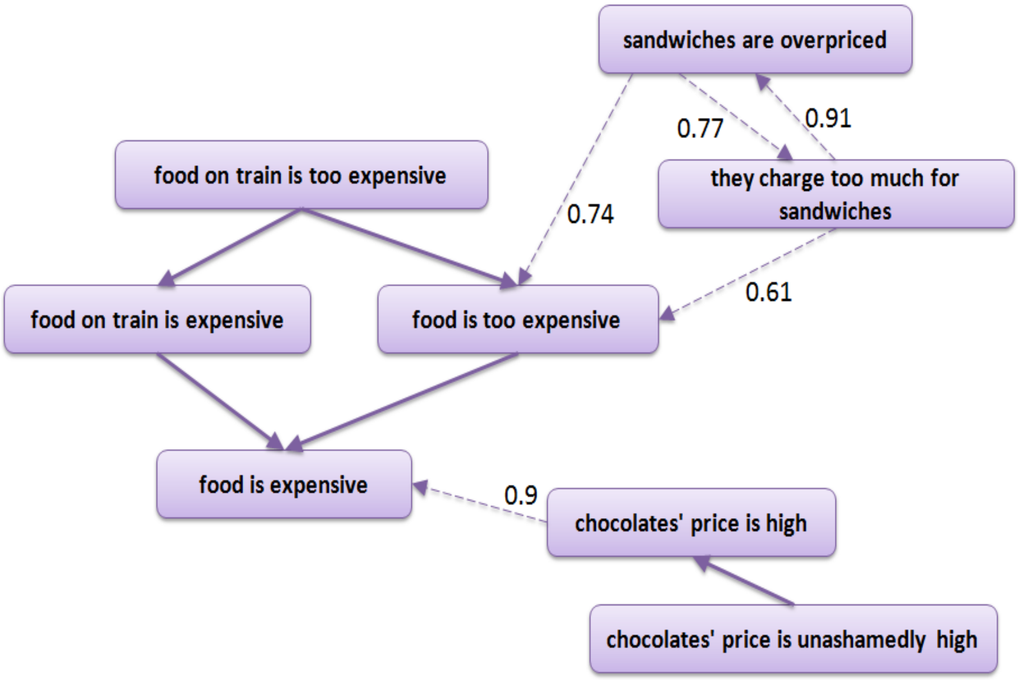
http://jgrapht.org/javadoc/org/jgrapht/graph/DefaultEdge.html

This class has the same attributes and methods as the class DefaultEdge: *source*, *target*, *weight* attributes; methods to obtain the source, target and weight information, and the expected constructor.

3.3.2 Raw Graph

The raw graphs, with their corresponding class *EntailmentGraphRaw*, are directed multigraphs (have multiple directed edges between the same pair of nodes) obtained by merging fragment graphs. The choice to represent this structure as a multigraph is motivated by the possibility that the entailment decision between a pair of nodes could be obtained from different entailment decision algorithms (EDAs). The different edges are combined when building a collapsed entailment graph, after all fragments graphs were merged into one raw graph.

An example of a raw graph is presented below. Dashed edges represent EDA entailment decisions with confidence scores, while solid edges correspond to edges copied from fragment graphs. For clarity reasons, in this example we only show decisions from a single EDA and do not show edges that correspond to “no entailment” decisions of the EDA.



The nodes of a raw graph are entailment units (*EntailmentUnit*), which cover a set of entailment unit mentions that express the same text fragment.

The edges of a raw graph are entailment relations (*EntailmentRelation*), which hold the information about whether the entailment relation holds between the edge's source and target nodes.

3.3.2.1 class EntailmentGraphRaw (eu.excitementproject.tl.structures.rawgraph)

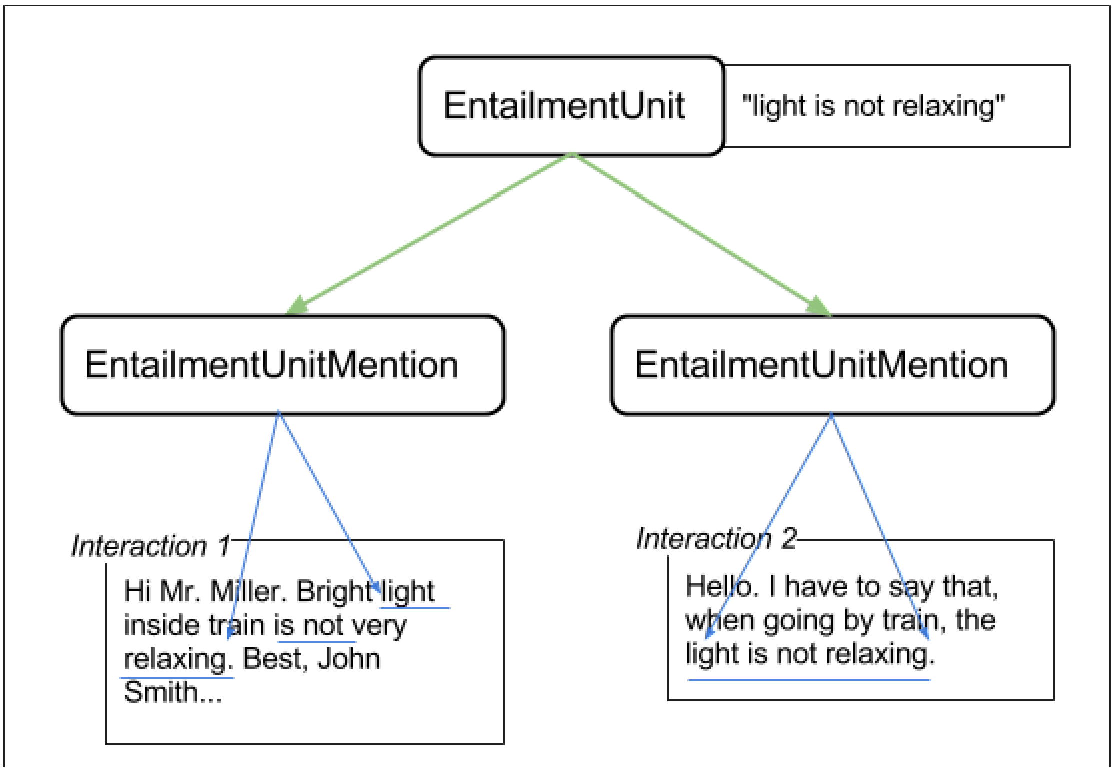
This class contains the graph structure for the raw graph. We call it *EntailmentGraphRaw*. This graph grows by adding to it *FragmentGraph*-s by "merging", which is done through the *GraphMerger* interface. The nodes are entailment units (*EntailmentUnit*), and the edges (*EntailmentRelation*) are generated based on decisions from the EDA. As such there can be several edges between the same two nodes, each corresponding to one EDA result. Edges can also be added based on graph structure or prior knowledge (e.g.by copying edges contained in fragment graphs). This graph extends DirectedMultigraph, to allow for multiple directed edges between the same two nodes. The JavaDoc for the DirectedMultigraph for information about inherited methods can be found here:

http://jgrapht.org/javadoc/org/jgrapht/graph/DirectedMultigraph.html

The class *EntailmentGraphRaw* contains methods for building, traversing, querying and saving the graph, as well as various auxiliary methods.

3.3.2.2 class EntailmentUnit (eu.excitementproject.tl.structures.rawgraph)

An *EntailmentUnit* refers to a piece of text that can occur in one or more input texts. *EntailmentUnit*-s form the nodes of the raw entailment graph. Each such node covers *EntailmentUnitMention*-s that represent the same text. The relationship between entailment unit mentions and entailment units is illustrated in the following figure.



EntailmentUnit texts are unique, i.e. two different *EntailmentUnit* objects within the same raw graph should never hold the same text.

3.3.2.3 class EntailmentRelation (eu.excitementproject.tl.structures.rawgraph)

This is the edge type for the raw graph (*EntailmentGraphRaw*). The edge "value" is a textual entailment decision (*eu.excitementproject.eop.common.TEDecision*) obtained from the EDA. The TEDecision object also stores (among other things) a decision label (*eu.excitementproject.eop.common.DecisionLabel*). For details on the *TEDecision* and *DecisionLabel* data types, see the EOP specification.

The class extends DefaultEdge:

http://jgrapht.org/javadoc/org/jgrapht/graph/DefaultEdge.html

3.3.3 Collapsed Graph

A collapsed graph, implemented in class *EntailmentGraphCollapsed*, is a simple directed graph, obtained from the raw graph by optimizing it based on the entailment decisions collected in the raw graph. This includes collapsing nodes corresponding to equivalent text fragments (from the point of view of the entailment relation), and multiple edges between the same pair of nodes, as well as resolving conflicts (transitivity violations) resulting from automatic entailment decisions.

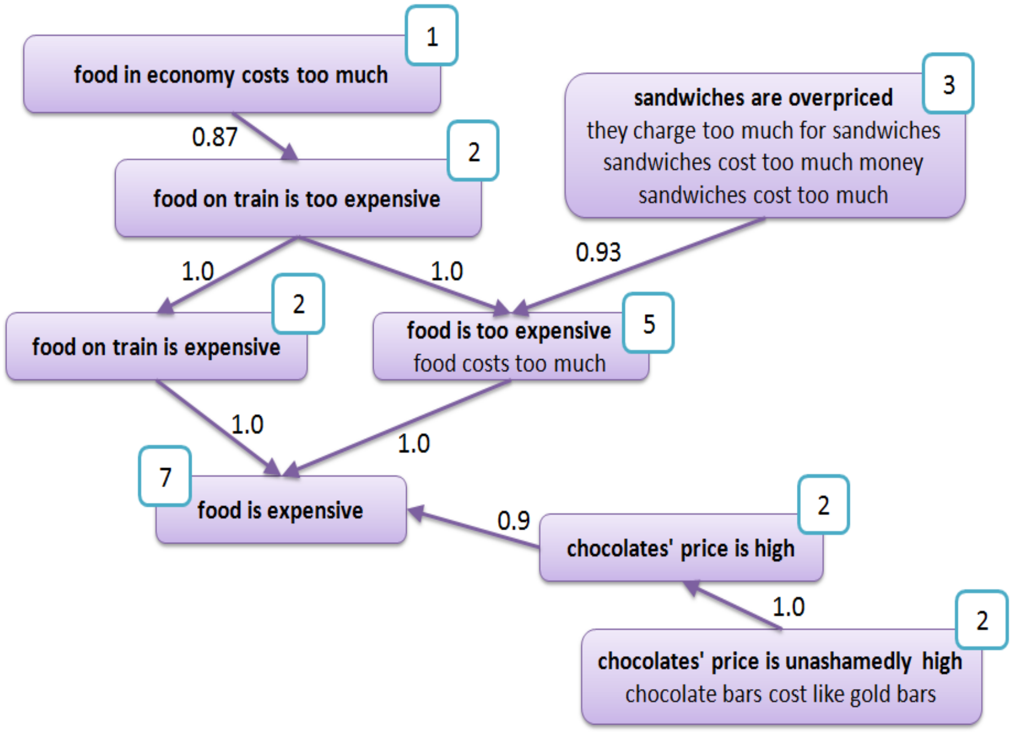
By definition, a collapsed graph is a transitive graph. The nodes of the collapsed graph are equivalence classes (*EquivalenceClass*), which cover semantically equivalent text fragments (entailment units). The edges of the collapsed graph are collapsed entailment relations (*EntailmentRelationCollapsed*).

3.3.3.1 class EntailmentGraphCollapsed (eu.excitementproject.tl.structures.collapsedgraph)

The structure of the collapsed graph is simpler than that of the raw graph:

- There is no need for multiple edges between the same pair of nodes: such multiple edges are collapsed to form a single edge with the final entailment decision. The presence of an edge between two nodes (source and target) means that there is an entailment relation between the two nodes in the direction source -> target.
- There are no cycles: entailment paths, which form cycles, are collapsed to form equivalence class (paraphrase) nodes.
- Edges and nodes contain less information: the information needed for internal purposes of building the graph is excluded.

The example below presents a simple collapsed graph. Numbers attached to the nodes are counters of occurrences.



This graph is built from the raw graph, by collapsing multiple edges between the same pair of vertices into one edge, and grouping entailment units into equivalence classes. This process is performed by the *Graph Optimizer* module.

Unlike the raw graph, this is no longer a multigraph, but a simple directed graph. It extends DefaultDirectedWeightedGraph. For inherited methods see the JavaDoc:

http://jgrapht.org/javadoc/org/jgrapht/graph/DefaultDirectedWeightedGraph.html

3.3.3.2 class EquivalenceClass (eu.excitementproject.tl.structures.collapsedgraph)

The node of an collapsed entailment graph is an equivalence class. This type of node contains all text fragments that are equivalent from the point of view of textual entailment. More formally, each *EquivalenceClass* contains a set of *EntailmentUnit*-s, which were considered paraphrasing by an algorithm utilized for graph construction.

3.3.3.3 class EntailmentRelationCollapsed (eu.excitementproject.tl.structures.collapsedgraph)

This class implements the collapsed graph edges, obtained by collapsing multiple edges (decisions from different EDAs or other sources) from the raw graph into one edge. The edges are directional and represent entailment relations between the source and the target nodes (the source node entails the target node). This class extends DefaultEdge: http://jgrapht.org/javadoc/org/jgrapht/graph/DefaultEdge.html

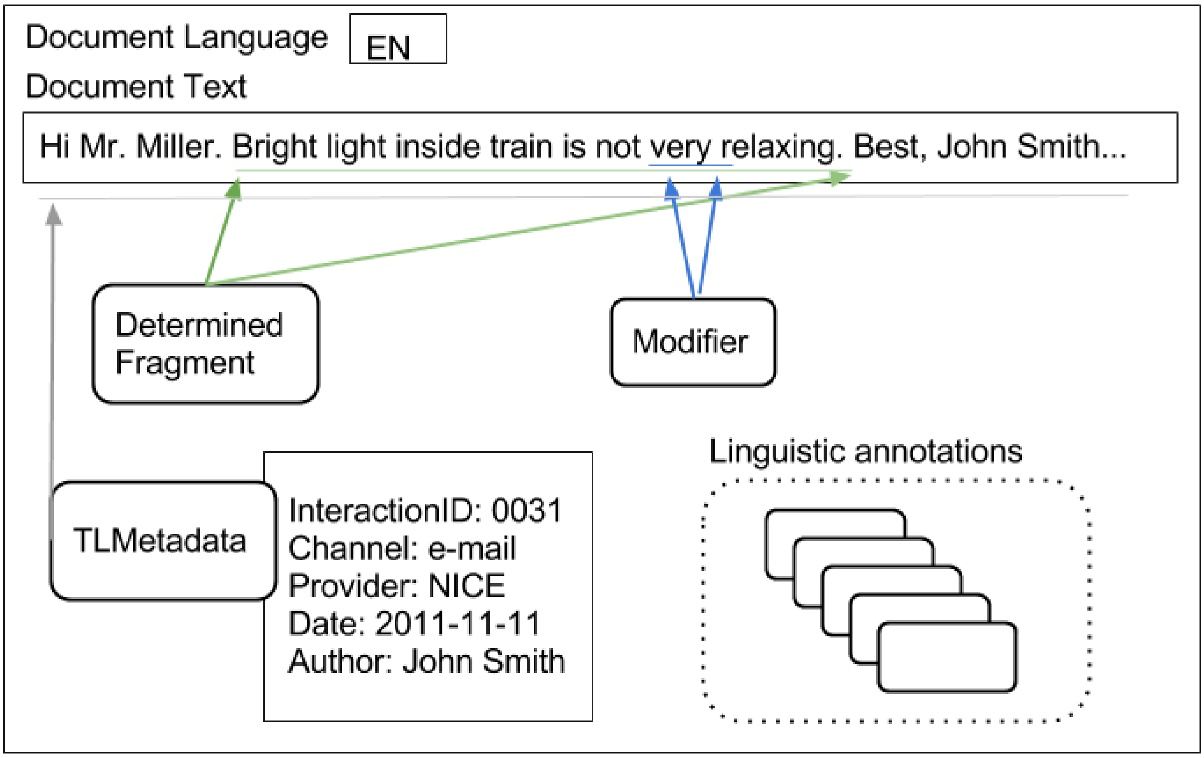
4. UIMA Type System

4.1 Introduction

There are two types of CASes used in the Transduction Layer. The first is the *input CAS*, which holds the customer interaction, fragment annotations and modifier annotations. The steps for obtaining these annotations are detailed in the decomposition data flow description.

The other type is the CAS data that is used as input for the EDA.

In this section, we present only the type systems that are newly added for the first case (input CAS). For other types (linguistic annotations for EDAs), please refer to the EOP specification. One input CAS always holds one Interaction. It can have multiple numbers of fragments and modifiers. The CAS can only have one Metadata, which includes the interaction ID. The CAS will also contain all linguistic annotations (POS, lemma, parse result, etc). The following figure shows an example of an input CAS.



Details about all types related to the input CAS are given in the following section.

4.2 Types

4.2.1 Metadata (eu.excitement.type.tl)

4.2.1.1 Supertype

- uima.tcas.Annotation

4.2.1.2 Features

The type includes: (all strings)

- interactionId
- channel
- provider
- date (string as YYYY-MM-DD)
- businessScenario
- author
- category

4.2.1.3 Description

This type description file defines the input CAS Metadata type, which records various metadata related to the Interaction and the input CAS. Note that one CAS should have only one metadata (only the first one should be considered, if more than one), and each CAS should have one metadata, even if all of its fields are null. Note that language ID is not recorded in this metadata type. It is directly recorded in CAS. Also note that all of the metadata are simply strings, and can be null if that metadata is missing.

4.2.2 FragmentAnnotation (eu.excitement.type.tl)

4.2.2.1 Super type

-uima.tcas.Annotation

4.2.2.2 Features

- begin/end (inherited): this span covers the general region, even if the fragment text is non-contiguous within the region.
- text (String): this holds the text that this fragmentation represents.

- fragParts (Array of FragmentParts type): this holds one or more FragmentsParts type in an array. Thus, it can actually map non-contiguous regions. If the fragmentation is contiguous, this array has only one item.

4.2.2.3 Description

This type annotates a *fragment*, as defined in EXCITEMENT WP6 (and WP2). This is the base type of two different fragments: AssumedFragment type and DeterminedFragment type.

Example.
0 23 44 67
The connection was slow. I was on vacation. GPRS was specially slow.
begin:0
end:67
text: The connection was slow. GPRS was specially slow.
fragParts(0): FragmentParts -begin:0 -end:23
(1): FragmentParts -begin:44 -end:67

4.2.3 FragmentPart (eu.excitement.type.tl)

4.2.3.1 Super type

-uima.tcas.Annotation

4.2.3.2 Features

(none)

4.2.3.3 Description

This is a type that is designed to represent one contiguous region of a (potentially non-contiguous) fragment. This is only used for that purpose, and does not have any additional feature.

4.2.4 AssumedFragment (eu.excitement.type.tl)

4.2.4.1 Super type

- FragmentAnnotation

4.2.4.2 Features

(none)

4.2.4.3 Description

This is a fragment annotation that is used to mark the *assumed fragment*. The WP7 application layer uses this annotation to mark what the WP7 application considers a possible fragment. This might not be accurate, and WP6 performs an additional fragment analysis.

4.2.5 DeterminedFragment (eu.excitement.type.tl)

4.2.5.1 Super type

- FragmentAnnotation

4.2.5.2 Features

(none)

4.2.5.3 Description

This is a fragmentation annotation that is "determined" by WP6 internal modules. Unlike the "assumed fragment", this is the actual fragment that will be further processed as the real fragment.

4.2.6 ModifierAnnotation (eu.excitement.type.tl)

4.2.6.1 Super type

-uima.tcas.Annotation

4.2.6.2 Features

- modifierParts (Array of ModifierPart type): this holds one or more ModifierPart-s in an array. Thus, it can actually map non-contiguous regions. If the modifier is contiguous, this array only has one item.
- dependsOn (ModifierAnnotation): If this modifier depends on some other modifier, this feature points to that modifier. This modifier depends on the modifier to which it points. (If the modifier pointed to by this feature does not exist, this modifier is not grammatical / meaningless.)

4.2.6.3 Description

This annotation type annotates a region as a "modifier", and it is used within WP6 modules to create the fragment graph nodes: each node represents a unique (and valid, from the point of view of the dependsOn relation) combination of modifiers. While this could be simple, it gets a bit complicated by "dependsOn" and "non-contiguous" regions.

See the following example:

0 24 27 31
Seats are uncomfortable as too old.

Here we have two modifiers.

ModifierAnnotation #1 "too"
-begin: 27
-end: 29
-modifierParts: (0) -begin:27 -end:29
-dependsOn: ModifierAnnotation #2

ModifierAnnotation #2 "as ... old"
-begin: 24
-end: 34
-modifierParts: (0) -begin:24 -end:25
(1) -begin:31 -end:33
-dependsOn: (null)

The above example shows two modifiers that have a dependency relation: "too" has a meaning only if "old" is present. Thus, removing only modifier #2 is not possible. It is marked in #1 that it depends on #2. (#1 is not a valid stand-alone modifier, and if #2 is removed, #1 should also be removed).

4.2.7 CategoryAnnotation (eu.excitement.type.tl)

4.2.7.1 Super type

- FragmentAnnotation

4.2.7.2 Features

- categories (array of CategoryDecision type): at least one or more category decision data associated with this fragment.

4.2.7.3 Description

This is a type designed to represent the result of use case 2 processing. It represents a fragment, but also with the data associated for category decision. The fragment annotated by this type has one or more category decision types, which annotates category id and confidence for that category.

4.2.8 CategoryDecision (eu.excitement.type.tl)

4.2.8.1 Super type

-uima.cas.Top

4.2.8.2Features

- categoriy id (String)
- confidence (Float)

4.2.8.3Description

This is the metadata used for the output of use case 2 (category annotation). This type is used in CategoryAnnotation, as an element of an array.

5. Interface Definitions for the WP6 Modules

This section describes interfaces of all Transduction Layer modules. Before looking into the interface definitions, please take a look at section 2, which details how each component listed in the section contributes to the whole process.

5.1 Interfaces of Decomposition Components

5.1.1 Fragment Annotator Module: interface *FragmentAnnotator* (eu.excitementproject.tl.decomposition.api)

5.1.1.1 General Description

As described in section 2.1.2 in the data flow chapter, an instance of *FragmentAnnotator* annotates a part of an interaction text as a coherent statement (a *fragment*). One fragment holds one statement. The goal of a fragment annotator is to identify them and correctly mark them.

The TL type system of input CAS has two types of fragment annotations. One is assumedFragment and the other is determinedFragment. An assumed fragment represents the belief of the caller who prepared the input, while a determinedFragment represents the decision of the FragmentAnnotator (an instance of this interface). The fragment annotator may use and rely on the assumed fragment annotation as evidence, or feature. But in general, it does not blindly follow it. determinedFragment annotations are the output of this module, and will be used as the "real" fragments in the downstream modules.

Thus, a Fragment Annotator consumes as its input an interaction and optionally assumed fragment annotations. The output is determined fragments, which work as the final decision of fragments.

5.1.1.2 API Methods

The interface contains one method, for adding (determined) fragment annotation to the given CAS.

- *void annotateFragments(JCas text) throws FragmentAnnotatorException*
  - @param text – JCas with text and metadata. It may additionally hold assumedFragment annotation.
  - @return – the method returns nothing. But the argument JCas text is enriched with real (determined) fragment annotations.
  - @throws (FragmentAnnotatorException) if any needed data is missing in the JCas, or if the module cannot successfully annotate the determined fragment.

5.1.1.3 Related Data Structure & Other Notes

Each fragment annotation can be non-contiguous. Check the type definition in section 4.2.2 to see how this is represented in the annotation.

The specific implementations of this module may call an LAP pipeline. The implementations of this interface can be found under the *eu.excitementproject.tl.decomposition.fragmentannotator* package. For information on the implementations of this module see section 6.1.1.

Any new implementation of this module should consider extending the *AbstractFragmentAnnotator* class. It forces the implementation to expose the LAP in the constructor. Also, any additional configurable parameters of this module implementation should be clearly exposed in the constructor.

5.1.2 Modifier Annotator Modules:

5.1.2.1 interface *ModifierAnnotator* (eu.excitementproject.tl.decomposition.api)

5.1.2.1.1 General Description

As described in section 2.1.3, a modifier annotator annotates any "non-essential" part of a statement (fragment) as a *modifier*. The term may be a bit misleading, since modifier does not necessarily refer to a syntactic modifier, but rather semantically to a term, which modifies the meaning of the predicate of the statement or the meaning of its main arguments (non-essential arguments, conditions, etc.). By using this term we follow the terminology defined in WP2. Examples of modifiers can be found in section 3.3.1, as well as in WP2 documentation.

A modifier annotator gets one input CAS, which holds the corresponding interaction with fragment annotation, and annotates modifiers found in the interaction. It uses the TL type ModifierAnnotation to do this.

5.1.2.1.2 API Methods

The interface contains one method, for adding modifier annotation to the given CAS.

- *void annotateModifiers(JCas text) throws ModifierAnnotatorException;*
    - @param text – a JCas with interaction text, metadata and determined fragment annotation(s). When successfully run, the input CAS is enriched with modifier annotation(s).
    - @return – the method returns nothing. The input CAS is directly enriched.

- @throws ModifierAnnotatorException if any of the needed data is missing in the input CAS, or when the module could not annotate the modifiers due to some failures.

5.1.2.1.3 Related Data Structure & Other Notes

Each modifier annotation can be non-continuous, and may contain dependencies among modifiers (check the type definition in section 4.2.6 for some examples.)

Specific implementations of this module may need to call an LAP pipeline. The implementations of this interface can be found under the *eu.excitementproject.tl.decomposition.-modifierannotator* package. For information on the implementations of this module see section 6.1.2.

New implementations of this module should consider extending the *AbstractModifierAnnotator* class or the *AbstractPOSBasedModifierAnnotator* class. They force the implementation to expose the LAP in the constructor. Also, any additional configurable parameters of this module implementation should be clearly exposed in the constructor.

5.1.3 Fragment Graph Generator Module: interface *FragmentGraphGenerator* (eu.excitementproject.tl.decomposition.api)

5.1.3.1 General Description

FragmentGraphGenerator is the interface between annotated document CAS objects and the Transduction Layer. The interface produces a set of *FragmentGraph*-s from an input CAS object. For each fragment (*determinedFragment*) annotated in the CAS, there will be a *FragmentGraph* object, which is further processed within the platform.

5.1.3.2 API Methods

The interface contains one method, for generating the FragmentGraph structures from the user's input:

- *Set<FragmentGraph> generateFragmentGraphs(JCas text)*:
  - @param text – the CAS object representing a document/user interaction
  - @return (Set<FragmentGraph>) – the set of FragmentGraph objects, one for each of the DeterminedFragment annotations in the text
  - @throws FragmentGraphGeneratorException when the FragmentGraph generation failed

5.1.3.3 Related Data Structure & Other Notes

The implementations of this interface can be found under the *eu.excitementproject.tl.decomposition.fragmentgraphgenerator* package.

New implementations of this module should consider extending the *AbstractFragment-GraphGenerator* class. For information on the implementations of this module see section 6.1.3.

5.2 Interfaces of Composition Components

5.2.1 Graph Merger Module: interface *GraphMerger* (eu.excitementproject.tl.composition.api)

5.2.1.1 General Description

The main goal of this module is "enriching" a raw graph (*EntailmentGraphRaw*), by merging newly mined fragment graphs. Conceptually, its input is two things. One is a set of fragment graphs, and the other is the raw graph. After a successful run, the module returns the enriched entailment graph. This module uses the entailment decision capability (EDA) of the EXCITEMENT open platform (EOP) to add fragment graphs into the entailment graph. For more details see section 2.2.1.

5.2.1.2 API Methods

EntailmentGraphRaw mergeGraphs(Set<FragmentGraph> fragmentGraphs, EntailmentGraphRaw workGraph)

The method consumes a raw graph (*EntailmentGraphRaw*) and a set of fragment graphs, which should be merged with the given raw graph. In case of success the enriched raw graph is returned. Otherwise the method throws a *GraphMergerException*.

- @param fragmentGraphs – a set of fragment graphs. If the set is empty or null, the input raw graph is returned unchanged.
- @param workGraph – the raw entailment graph that should be enriched. If this parameter is null, a new empty graph is created and merged with the given fragmentGraphs.
- @return (EntailmentGraphRaw) the given raw graph enriched by the given set of fragments.
- @throws (GraphMergerException) if the implementation cannot merge the graphs for some reason

EntailmentGraphRaw mergeGraphs(FragmentGraph fragmentGraph, EntailmentGraphRaw workGraph)

The method consumes a raw graph (*EntailmentGraphRaw*) and a single fragment graph, which should be merged with the given raw graph. In case of success the enriched raw graph is returned. Otherwise the method throws a *GraphMergerException*.

- @param fragmentGraph – the fragment graph. If this parameter is null, the input raw graph is returned unchanged.
- @param workGraph – the raw entailment graph that should be enriched. If this parameter is null, a new raw graph is created based on the given fragmentGraph.
- @return (EntailmentGraphRaw) the given raw graph enriched by the given fragment graph.
- @throws (GraphMergerException) if the implementation cannot merge the graphs for some reason

5.2.1.3 Related Data Structure & Other Notes

An implementation of this interface might need to call an LAP and is most likely to call an EDA. The needed LAP and EDA related configurations should be passed via the Constructor (thus, they are not defined in the interface). Also, any additional configurable parameters of this module implementation should be clearly exposed in the constructor.

The implementations of this interface can be found under the *eu.excitementproject.tl.composition.graphmerger* package.

For implementing this interface it is recommended to extend the *AbstractGraphMerger* class. This abstract implementation contains auxiliary methods that are expected to be common over different implementations. For information on the implementations of this module see section 6.2.1.

5.2.2 Graph Optimizer Module: interface *GraphOptimizer* (eu.excitementproject.tl.composition.api)

5.2.2.1 General Description

This module consumes a raw graph (*EntailmentGraphRaw*) and produces the *collapsed graph* or final graph (*EntailmentGraphCollapsed*). For more details see section 2.2.2.

5.2.2.2 API Methods

EntailmentGraphCollapsed optimizeGraph(EntailmentGraphRaw workGraph), EntailmentGraphCollapsed optimizeGraph (EntailmentGraphRaw rawGraph, double threshold)

These methods consume a raw graph (*EntailmentGraphRaw*) and produce the final collapsed entailment graph (*EntailmentGraphCollapsed*). In case of success the collapsed graph is returned. Otherwise the method throws a *GraphOptimizerException*.

- @param workGraph – the raw entailment graph, which should be optimized
- @param threshold –if provided, confidence threshold representing the minimum confidence for an edge from the raw graph to be kept in the collapsed graph
- @return (EntailmentGraphCollapsed) the resulting collapsed entailment graph
- @throws GraphOptimizerException if the implementation cannot convert the graph for some reason

5.2.2.3 Related Data Structure & Other Notes

We do not foresee any external EOP component dependency for this module. Yet, if any arguments or configurable values are needed, they should be exposed in the implementation constructor.

The implementations of this interface can be found under the *eu.excitementproject.tl.-composition.graphoptimizer* package.

For implementing this interface it is recommended to extend the *AbstractGraphOptimizer* class. This abstract implementation contains auxiliary methods that are expected to be common over different implementations.

For information on the implementations of this module see section 6.2.2.

5.2.3 Confidence Calculator Module: interface *ConfidenceCalculator* (eu.excitementproject.tl.composition.api)

5.2.3.1 General Description

This module reads the category information stored in the mentions associated to a collapsed graph node, combines it to a single score per category per node, and adds the scores as new information to the graph. For more details see section 2.3.1.

5.2.3.2 API Methods

The interface contains one method:

void computeCategoryConfidences(EntailmentGraphCollapsed entailmentGraph) throws ConfidenceCalculatorException
This method computes category confidence scores per node in the input graph and adds this information to the input graph.

- @param entailmentGraph – the collapsed entailment graph to which category confidence scores will be added
- @throws ConfidenceCalculatorException if the calculation fails

5.2.3.3 Related Data Structure & Other Notes

The implementations of this interface can be found under the *eu.excitementproject.tl.-composition.confidencecalculator* package.

For implementing this interface it is recommended to extend the *AbstractConfidenceCalculator* class. For information on the implementation of this module see section 6.2.3.

5.2.4 Node Matcher Modules

5.2.4.1 interfaces *NodeMatcher and NodeMatcherWithIndex* (eu.excitementproject.tl.composition.api)

5.2.4.1.1 General Description

This module matches a given *FragmentGraph* against an *EntailmentGraphCollapsed* and returns a set of *NodeMatch*-es. The *EntailmentGraphCollapsed* should be exposed in the constructor. For more details see section 2.3.1.

5.2.4.1.2 API Methods

The interface contains one method:

> *Set<NodeMatch> findMatchingNodesInGraph(FragmentGraph fragmentGraph) throws NodeMatcherException*
> This method takes a fragment graph and returns a set of node matches (matching nodes in the entailment graph associated to match confidence scores).
> - @param fragmentGraph – the fragment graph to be matched
> - @return (Set<NodeMatch>) – the set of node matches
> - @throws NodeMatcherException if the match fails

5.2.4.1.3 Related Data Structure & Other Notes

There are two related data structures: *NodeMatch* and *PerNodeScore*, which are defined in the following.

The implementations of this interface can be found under the *eu.excitementproject.tl.-composition.nodematcher* package.

For implementing this interface it is recommended to extend the *AbstractNodeMatcher* or the *AbstractNodeMatcherLucene* class. The latter one contains auxiliary methods that are expected to be common over different index-based implementations. For information on the implementation(s) of this module see section 6.2.4.

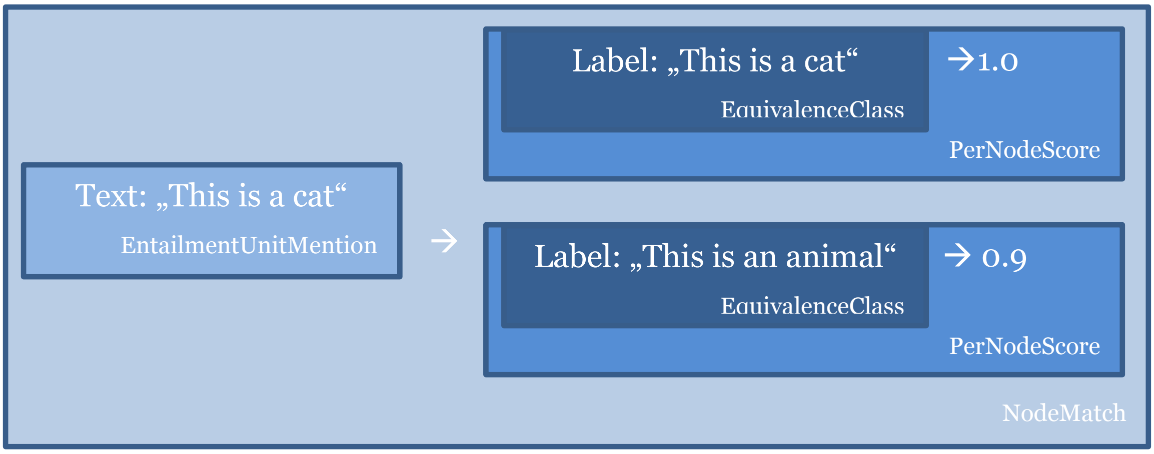
5.2.4.1.3.1 Class NodeMatch

A "node match" holds an *EntailmentUnitMention* associated to a set of *PerNodeScore*-s.

5.2.4.1.3.2 Class PerNodeScore

A "per node score" keeps matched nodes with the corresponding confidence score of the match. It is a tuple <E,C>, where E denotes an *EquivalenceClass* (a matching node in the collapsed graph) and C denotes a confidence score (a score expressing how well this node matches the entailment unit mention, to which the *PerNodeScore* object is associated).

An example illustrating the relationship between *NodeMatch* and *PerNodeScore* is given in the following:



5.2.5 Category Annotator Module: interface *CategoryAnnotator* (eu.excitementproject.tl.composition.api)

5.2.5.1 General Description

This module adds category annotation to a given input CAS, i.e. it assigns, to a particular fragment in the input CAS, a category ID together with a confidence score expressing how well this category matches the fragment. For computing this confidence score, this module makes use of the category information stored in the *NodeMatch* objects returned for the fragment, i.e. the output of the NodeMatcher module. *NodeMatch* objects hold category information within their *PerNodeScore* objects: Each *PerNodeScore* holds a matching collapsed graph node associated to categories and their confidence scores (computed using an implementation of the *ConfidenceCalculator* module). The module combines the category information of several *PerNodeScore*-s into a single score per category and fragment. For more details see section 2.3.3.

5.2.5.2 API Methods

The interface contains one method:

> *void addCategoryAnnotation(JCas cas, Set<NodeMatch> matches) throws CategoryAnnotatorException*
>
> This method takes a set of node matches and combines the category information found in the node matches to a category confidence that is then added to the input CAS.
>
> - @param cas – input CAS
> - @param matches – set of node matches
> - @return no new data, but the input CAS is annotated with category annotation
> - @throws (CategoryAnnotatorException) if category annotation fails

5.2.5.3 Related Data Structure & Other Notes

The implementations of this interface can be found under the *eu.excitementproject.tl.composition.categoryannotator* package.

For implementing this interface it is recommended to extend the *AbstractCategoryAnnotator* class. For information on the implementation of this module see section 6.2.5.

5.3 Top Level Interface Definition

5.3.1 Introduction to the Top Level

In this document, the top level is used for the main data flow runner. The Transduction Layer (TL) top level code configures and runs available components of the TL to instantiate one instance of the TL that will work for one of the WP7 use cases.

The TL top levels are designed to make the TL transparent: Once a WP7 user sets the TL and EOP up and running, WP7 code only needs to access the top level APIs to get all the results.

Note that, to the TL layer and to the users (WP7), EOP is exposed with two interfaces. One is the LAPAccess interface that exposes annotation capabilities, and the other is the EDABasic interface that enables us to make entailment decisions.

EOP accepts and uses a configuration system. EDAs are often shipped with a sophisticated and already configured configuration file. Such files must be provided at the initialization time of EDAs (also for LAPs, if LAP is complex). To customize EDAs of the EOP to a specific need (e.g., retraining, parameter tuning), one has to study the configuration file of a specific EDA.

Note that within the TL layer, we have adopted a "configuration-less" approach of providing modules. The TL layer does not keep its own configuration files. However, it is definitely not a stateless machine: it has various possible parameters (e.g., threshold variables). However, those are systemically exposed on each component's constructor. They are also exposed in the top-level constructors' arguments.

The reason for this "configuration-less" approach, in which we expose every state/parameter on the constructor level, is to reduce the amount of configuration storage. We expect that the calling system of WP7 will have its own property/configuration/state storage mechanism, differing for each industrial partner. By exposing and not defining parameters as a configuration format, we hope to remove the need of WP7 users to work with three configurations (industrial system, EOP, and then TL). We aim to be the adaptor layer, in which the storage of properties/configuration can be integrated into the industrial partners' system by exposing all TL properties transparently to the users.

The TL provides two top level APIs described in the following sections. One is for use case 1, and the other is for use case 2. Any top level module should implement one or both of the interfaces.

5.3.2 Use Case 1 Top Level API: interface *UseCaseOneRunner* (eu.excitementproject.tl.toplevel.api)

5.3.2.1 General Description

This top level interface configures and runs the TL platform for WP7's use case 1, applicable to all the scenarios.

5.3.2.2 API Methods

This use case requires building raw and collapsed entailment graphs from user interactions, and is implemented through the following methods:

- *EntailmentGraphRaw buildRawGraph (List<JCas> annotatedInteractions)*
    - @param interactions – a set of (annotated) user interactions represented as CAS objects
    - @return (EntailmentGraphRaw) – raw entailment graph of text fragments connected through entailment relations; the graph is obtained by merging the *FragmentGraph*-s corresponding to each fragment (*DeterminedFragment*) annotation in the input CAS objects.

- *EntailmentGraphRaw buildRawGraph (Set<Interaction> interactions)*
    - @param interactions – a set of user interactions represented as Interaction objects
    - @return (EntailmentGraphRaw) – raw entailment graph of text fragments connected through entailment relations; the graph is obtained by merging the *FragmentGraph*-s corresponding to each fragment (*DeterminedFragment*) annotation in the input *Interaction* objects into a raw graph, and further collapsing the raw graph based on the (optional) confidence score.

- *EntailmentGraphCollapsed buildCollapsedGraph (List<JCas> annotatedInteractions)*
- *EntailmentGraphCollapsed buildCollapsedGraph (List<JCas> annotatedInteractions, double threshold)*
    - @param interactions – a set of (annotated) user interactions represented as CAS objects
    - @param threshold – if provided, confidence threshold representing the minimum confidence for an edge from the raw graph to be kept in the collapsed graph
    - @return (EntailmentGraphCollapsed) – graph of text fragments connected through entailment relations, obtained by merging the *FragmentGraph*-s corresponding to each fragment (*DeterminedFragment*) annotation in the *Interaction* objects into a raw graph, and further optimizing the raw graph based on the (optional) confidence score.

- *EntailmentGraphCollapsed buildCollapsedGraph (Set<Interaction> interactions)*
- *EntailmentGraphCollapsed buildCollapsedGraph (Set<Interaction> interactions, double threshold)*
    - @param interactions – a set of user interactions represented as Interaction objects
    - @param threshold – if provided, confidence threshold representing the minimum confidence for an edge from the raw graph to be kept in the collapsed graph
    - @return (EntailmentGraphCollapsed) – graph of text fragments connected through entailment relations, obtained by optimizing an *EntailmentGraphRaw* based on the confidence score; the raw graph was obtained by annotating the input interactions and merging the *FragmentGraph*-s correspond-

ing to each fragment (*DeterminedFragment*) annotation in the *Interaction* objects.

- *EntailmentGraphCollapsed buildCollapsedGraph (File rawGraph)*
- *EntailmentGraphCollapsed buildCollapsedGraph (File rawGraph, double threshold)*
  - @param rawGraph – an XML file representation of a raw graph
  - @param threshold – if provided, confidence threshold representing the minimum confidence for an edge from the raw graph to be kept in the collapsed graph
  - @return (EntailmentGraphCollapsed) – graph obtained by optimizing the graph read from *rawGraph* based on the confidence score

5.3.2.3 Related Data Structure & Other Notes

The implementations of this interface can be found under the *eu.excitementproject.tl.toplevel.usecaseonerunner* package.

5.3.3 Use Case 2 Top Level API: interface *UseCaseTwoRunner* (eu.excitementproject.tl.toplevel.api)

5.3.3.1 General Description

This top level interface configures and runs available TL components to instantiate one "instance" of the transduction layer that will work for WP7's use case 2.

5.3.3.2 API Methods

- *void annotateCategories(JCas cas, EntailmentGraph* Collapsed *graph)*
  - @param cas – input CAS representing the interaction to be categorized
  - @param graph – collapsed entailment graph created for this domain
  - @return no new data, but the input CAS is enriched with category annotations computed based on category information extracted from the graph

5.3.3.3 Related Data Structure & Other Notes

The implementations of this interface can be found under the *eu.excitementproject.tl.toplevel.usecasetworunner* package.

6. Implementation of the Modules

In the previous chapters, we have provided a description of the data structures and interfaces we defined for the various transduction layer modules, including the core modules as well as the top level modules. In this chapter, we describe the final implementations of those modules, which show how the defined interfaces and data structures were used to realize the two industrial use cases. For more details about the implementation of these modules, please check the Java documentation in the code.

6.1 Implementation of Decomposition Components

6.1.1 Fragment Annotator Modules

The classes described in this section implement the interface *FragmentAnnotator*.

6.1.1.1 class SentenceAsFragmentAnnotator(eu.excitementproject.tl.decomposition.fragmentannotator)

This is a very simple fragment annotation method that annotates each sentence as a separate fragment. The component calls the given LAP or searches the input CAS to find the sentence annotation, and then annotates each sentence as one fragment.
The component calls the given LAP once, if no sentence annotation was found in the input CAS. If one or more sentences were found within the CAS, an LAP call is not performed.

6.1.1.2 Class KeywordBasedFragmentAnnotator

The *KeywordBasedFragmentAnnotator* builds fragments starting from keywords. It uses dependency relations provided by an LAP to gather the grammatical phrase that encompasses the given keyword. If no keyword was provided or no dependency relations are found even after adding grammatical annotations through the LAP, no fragments are added.

The quality of the output depends much on the quality of the dependency relations.

6.1.1.3 Class KeywordBasedFixedLengthFragmentAnnotator

As an alternative to the *KeywordBasedFragmentAnnotator* that requires a dependency parser to build a fragment, we introduced the *KeywordBasedFixedLengthFragmentAnnotator*, which builds a fragment centered on the given keyword by adding N tokens to the left and to the right, while respecting sentence boundaries (and not counting punctuation). This is a light weight alternative to the keyword-based annotator, that our experiments have shown to provide high quality fragments.

6.1.1.4 Class TokenAsFragmentAnnotator

This is a fragment annotation method that annotates each token (except punctuation) as a fragment.

The component calls the given LAP or searches the input CAS to find the token annotation, and then annotates each token as one fragment. The component calls the given LAP once, if no token annotation was found in the input CAS. If one or more tokens were found within the CAS, an LAP call is not performed.

6.1.1.5 Class TokenAsFragmentAnnotatorForGerman

This is a fragment annotation method that annotates each token (except punctuation), and, unlike the *TokenAsFragmentAnnotator*, each component of a German compound word as a fragment. As a German compound splitter is used, this fragment annotator only works for German language input. If a filter for parts of speech is passed to the constructor (via the *POSTag_DE* parameter), then only tokens are annotated, which match the filter.

As compound splitter, the GermanWordSplitter of the jWordSplitter[1] is used. We use the "strict" mode of this splitter, which only splits compounds if all components are recognized as words, e.g. "Fehlermeldung" [*error message*] is split into "Fehler" [*error*] and "Meldung" [*message*], but "gefunden" [*found*] is not split into "ge" and "funden".

The embedded dictionary of the GermanWordSplitter lacks terms from the computer domain. Because of this, we were not able to split compounds like "XML-Datei" [*XML file*], "PRC-Datei" [*PRC file*], "XLS-Scanner" [*XLS scanner*], etc. We noticed that, if these computer terms are constituents of German compounds, then they are very often connected to the other constituents by a hyphen. Therefore we added hyphen-based decompounding. It should be noticed that the GermanWordSplitter allows the expansion of the embedded dictionary or usage a self-built dictionary. This possibility might also be used to deal with the problem of decompounding computer terms, but the hyphen based decompounding was a quick solution, which served our purposes very well.

The user can specify a decomposition type in the constructor of the class via the *WordDecompositionType* parameter, which determines the type of decompounding to be applied: no decompounding (NONE), hyphen-based decompounding only (ONLY_HYPHEN), or full decompounding (NO_RESTRICTION). In the "full decompounding" mode we apply both the strict-mode decompounding of the GermanWordSplitter and the hyphen-based decompounding.

[1] http://www.danielnaber.de/jwordsplitter/

6.1.1.6 Class DependencyAsFragmentAnnotator

This is a fragment annotation method that annotates a combination of two tokens (except punctuation) linked via a dependency as a fragment. If a filter for dependency types is passed to the constructor, then only dependencies matching the filter are annotated.

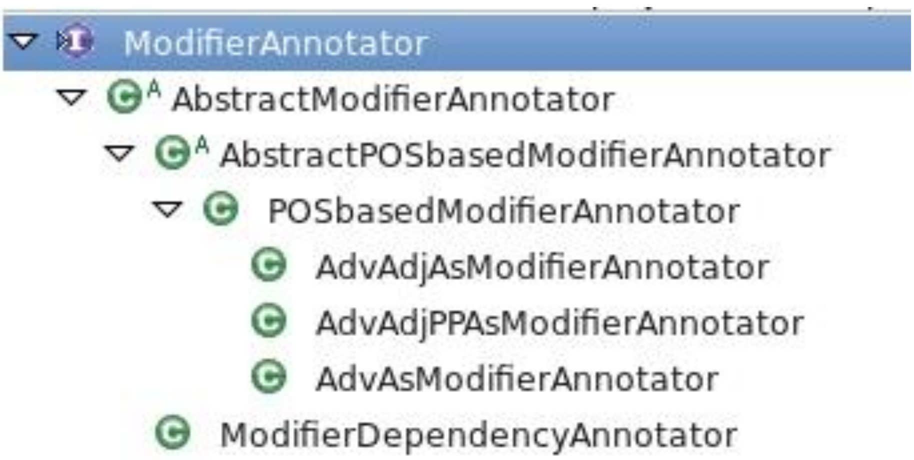
The component calls the given LAP or searches the input CAS to find the dependency annotation, and then annotates each two-token combination as one fragment. The component calls the given LAP once, if no dependency annotation was found in the input CAS. If one or more dependencies were found within the CAS, an LAP call is not performed.

6.1.1.7 Class DependencyAsFragmentAnnotatorForGerman

As the *DependencyAsFragmentAnnotator*, this is a fragment annotation method that annotates a combination of two tokens (except punctuation) linked via a dependency as a fragment. In addition to the filter for dependency types, this implementation allows the user to specify, in the constructor, filters for part of speech of governor and / or dependent (passed via the *POSTag_DE* parameter) and for particular words. If filters are passed, only dependencies matching the filters are annotated.

6.1.2 Modifier Annotator Modules

The classes in this package implement the interface *ModifierAnnotator,* which requires the method annotateModifiers(JCas aJCas). The first class in the hierarchy is AbstractModifierAnnotator which defines the attributes and methods shared by all descendants.



6.1.2.1 Class AbstractModifierAnnotator

The modifier annotators have two attributes: the Linguistic Analysis Pipeline (LAP) to add grammatical information (minimally parts of speech), and (optionally) a fragment annotator for marking modifiers only inside fragments.

Modifiers may have relations with one another, such that removing one requires the removal of all others that depend on it, as in the example: "very small amouns" -- removing "small" requires removing "very" as well ("very amounts" is not grammatical). To capture this dependency relation which is required for all modifier annotators, the *AbstractModifierAnnotator* implements three static methods:

addDependencies(JCas aJCas) – finds and adds dependencies between modifiers in all annotated fragments in the *aJCas* object.

addDependencies(JCas aJCas, Annotation a) – finds and adds dependencies between modifiers subsumed by the given annotation *a*.

addDependencies(JCas aJCas, LAPAccess lap) – finds and adds dependencies between modifiers in all annotated fragments in the *aJCas* object, after adding grammatical annotations with the given *lap*.

6.1.2.2 Class AbstractPOSbasedModifierAnnotator

AbstractPOSbasedModifierAnnotator is the ancestor of modifier annotators that rely on POS information. It adds three attributes to those defined by its parent (*AbstractModifierAnnotator*):

wantedClasses – specifies the parts-of-speech that will be considered by the annotator;

checkNegation – a boolean value that indicates whether the annotator will verify whether a potential modifier is in the scope of a negation (in which case it will not be annotated as a modifier);

modLogger – a logger for tracking behaviour

This class implements the interface method *annotateModifiers(JCas aJCas)*, and adds the following:

addModifierAnnotations(Iterator<Annotation> fragIter, Jcas aJCas) – iterates over all Annotations (specifically fragments), and adds modifiers according to the wanted parts of speech.

addModifiers(JCas aJCas, FragmentAnnotation frag, int negationPosition, Class pos) – adds modifier annotations with part-of-speech *pos* for the specified fragment *frag*, taking into account the negation position if attribute *checkNegation* is true.

addPOSclasses() – a abstract method that will be implemented by the descendants, each according the parts-of-speech it works with

isModifier(JCas aJCas, Annotation a, FragmentAnnotation frag) – verifies whether the given annotation *a* is a "proper" modifier – is not in predicative position (this is relevant particularly for adjectival modifiers).

6.1.2.3 Class POSbasedModifierAnnotator

POSbasedModifierAnnotator is a descendant of *AbstractPOSbasedModifierAnnotator*. It implements the abstract methods, and in particular implements several variations of the *addPOSClasses* method.

6.1.2.4 Classes AdvAsModifierAnnotatorand AdvAdjAsModifierAnnotator (eu.excitementproject.tl.decomposition.modifierannotator)

These classes are instances of *POSBasedModifierAnnotator*. The part-of-speech classes for each (*adv*, and *adv, adj* respectively) are specified in the constructors.

6.1.2.5 Class AdvAdjPPAsModifierAnnotator

AdvAdjPPAsModifierAnnotator is also an instance of the *POSBasedModifierAnnotator*. This one reimplements the interface method *annotateModifiers* because prepositions (PP) are treated differently than adjectives or adverbs. The new version of the interface method invokes the specially defined method *addPPmodifiers(JCas aJCas, FragmentAnnotation frag, Annotation a)* which builds the prepositional phrase starting from the preposition (a) and using dependency relations.

6.1.2.6 Class ModifierDependencyAnnotator

ModifierDependencyAnnotator implements *AbstractModifierAnnotator*, but does not actually add modifier annotations, rather it enriches the existing modifier annotations with the *dependsOn* relation (if it finds any). We found this necessary because the gold-standard annotations did not always include this information, which is however important for generating nodes for the fragment graph. The other modifier annotators add the *dependsOn* relation during the annotation process.

6.1.3 Fragment Graph Generator Modules

The classes described in this section implement the *FragmentGraphGenerator* interface, where we assume the input data is a CAS object with the structure described in chapter 4 of this document.

6.1.3.1 Class FragmentGraphGeneratorFromCAS (eu.excitementproject.tl.decomposition.fragmentgraphgenerator)

This class implements the *generateFragmentGraphs* method of the interface (described in section 5.1.3), by iterating over the *DeterminedFragment* annotations from the input CAS object.

6.1.3.2 Class FragmentGraphLiteGeneratorFromCAS (eu.excitementproject.tl.decomposition.fragmentgraphgenerator)

The *FragmentGraphLiteGeneratorFromCAS* class , unlike the *FragmentGraphGeneratorFromCAS* (implemented in the first iteration of the TL), produces "lite" fragment graphs, which have at most two nodes, one corresponding to the base statement, the other to the top statement (if the corresponding fragment has any modifiers). The reason for adding this class was that the industrial data (in particular the one from our ALMA partner) contained very long sentences, with numerous modifiers, leading to inordinately large fragment graphs – roughly, if there are n modifiers in a fragment, the corresponding fragment graph will have 2^n nodes. The class implements the *generateFragmentGraphs* method of the interface (described in section 5.1.3), by iterating over the *DeterminedFragment* annotations from the input CAS object.

6.2 Implementation of Composition Components

6.2.1 Graph Merger Modules

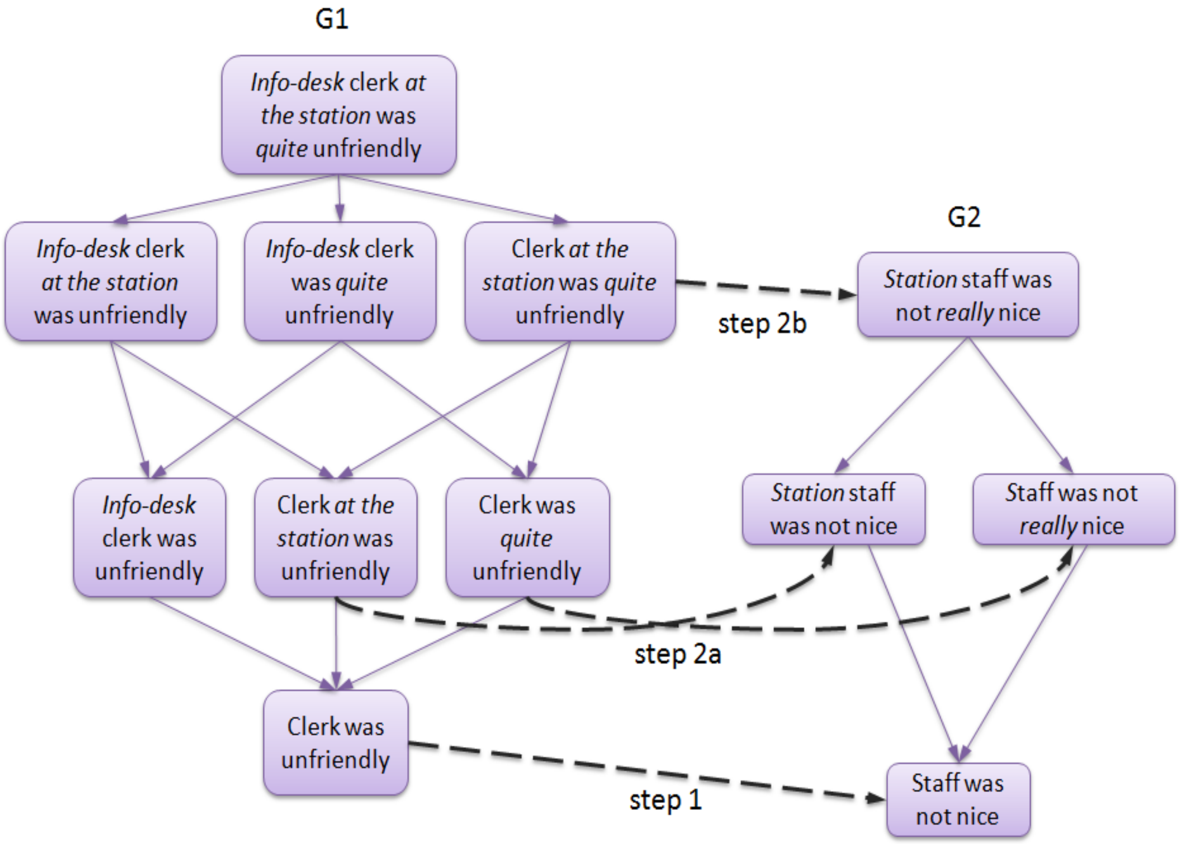
The classes described in this section implement the *GraphMerger* module.

6.2.1.1 Class StructureBasedGraphMerger (eu.excitementproject.tl.composition.graphmerger)

This implementation of the *GraphMerger* module automates the manual annotation procedure developed within WP2, which takes into consideration the structure of the input fragment graphs. According to this procedure, in order to merge two fragment graphs G1 and G2, the following steps should be performed:

1. Check for entailment (obtain EDA decision) in either direction between the base statements of the two graphs. If there is no entailment relation, the graphs should not be connected.
2. If there is entailment in one direction (let's assume G1 base statement -> G2 base statement)

   a. Check for entailment between all the pairs G1 node^1 -> G2 node^1, where nodes^1 hold 1-modifier statements, which directly entail the base statement of the corresponding graph.
   b. Induce entailment for upper-level (i=2..n) nodes in the direction G1 node^i -> G2 node^i, if each of the nodes G1 node^{i-1} directly entailed by G1 node^i ,entails a node G2 node^{i-1} directly entailed by G2 node^i.
3. If there is entailment in both directions (paraphrase), perform step 2 in both directions.



The aim of the algorithm is to minimize the number of annotations (EDA calls in our case) needed to perform the merge.
This implementation of the module performs the procedure described above to merge the new fragment graph with each of the fragment graphs that are recognized within the given raw graph (i.e. were previously merged with the raw graph), while ensuring not to call an EDA twice for the same pair of statements.

6.2.1.2 class AllPairsGraphMerger (eu.excitementproject.tl.composition.graphmerger)

This baseline implementation of the *GraphMerger* module merges fragment graphs by obtaining an entailment decision for each possible pair of nodes. The merging procedure ensures not to call an EDA twice for the same pair of statements, as well as for node pairs from the same fragment graph.

6.2.1.3 class LegacyAllPairsGraphMerger (eu.excitementproject.tl.composition.graphmerger)

This graph merger performs the merge by comparing all possible node pairs. Note that in this implementation only "entailment" edges are added during the merge, while "non-entailment" edges are not added, i.e., the absence of an edge in the merged graph should be interpreted as "no entailment".

Attention: This is now a legacy class. It produces a valid raw graph, but the graph may not be processed properly by a *GraphOptimizer*.

6.2.2 Graph Optimizer Modules

The classes described in this section implement the *GraphOptimizer* module.

6.2.2.1 Class SimpleGraphOptimizer (eu.excitementproject.tl.composition.graphoptimizer)

This baseline implementation produces a collapsed graph from a raw graph by performing the following steps:

1. Remove all non-entailing edges, as well as entailing edges with confidence below a certain threshold (specified in the generator's constructor). .
2. Recognize cycles and collapse all the nodes along each cycle's path into a single *EquivalenceClass* node. The resulting graph is a transitive graph (with no transitivity violations).

6.2.2.2 Class GlobalGraphOptimizer (eu.excitementproject.tl.composition.graphoptimizer)

This implementation applies to a given raw graph the global optimization algorithm of Berant et al. (2012)[2] , implemented within the Excitement Open Platform. The optimizer ensures that the fragment graph edges, both positive (entailing) and negative (non-entailing), are not changed by the optimization algorithm. The resulting graph is a transitive graph (with no transitivity violations).

[2] Berant, Jonathan, Ido Dagan, and Jacob Goldberger. 2012. Learning entailment relations by global graph structure optimization. Computational Linguistics, 38(1):73–111.

6.2.3 Confidence Calculator modules

6.2.3.1 Class ConfidenceCalculatorCategoricalFrequencyDistribution (eu.excitementproject.tl.composition.confidencecalculator)

This implementation of the *ConfidenceCalculator* module provides several ways of computing confidence scores per category for each node in the collapsed graph based on the frequency distribution of categories in the mentions associated to the node.

For each node, we first collect the frequency distribution on the node by retrieving the category of each of the m mentions associated to the node and storing the sum of occurrences of this category.

To compute the final confidence for each category, we provide a Naïve Bayes implementation and several TFIDF variants. The TFIDF variant to be used can be specified via a three-size character array parameter (based on SMART notation presented in Manning et al.[3], chapter 6, p.128).

The possible values are specified and explained in the following table:

Parameter	Possible values	Explanation
“term frequency” (tf)	n	Number of occurrences of the category in the node (tf)
	l	Sublinear tf-scaling: 1 + Math.log(tf)
“document frequency”	t	Math.log(N/n), where N = the total number of different categories and n = the number of different categories associated to this particular node
	n	1
“normalization”	n	None
	c	Cosine normalization

In addition, a “category boost” value can be specified. This adds the category boost value to all categories associated to mentions that appeared in the category description and thus gives more weight to fragments appearing in the description of the category.

[3] Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze, *Introduction to Information Retrieval*, Cambridge University Press. 2008.

6.2.4 Node Matcher modules

The classes described in this section implement the *NodeMatcher* module.

6.2.4.1 Class NodeMatcherLongestOnly (eu.excitementproject.tl.composition.nodematcher)

This implementation compares an input fragment graph to an input entailment graph and tries to find the longest match: It starts with the complete statement, on which the fragment graph was built (i.e., the statement containing all modifiers), and tries to find a matching node in the entailment graph. If a match is found, it returns this matching node; otherwise it tries to match the strings on the next level of the fragment graph, i.e. the statements in which one modifier is missing. Again, if a match is found, the node(s) are returned, otherwise the process continues until the base statement (in which all modifiers are removed) is reached.

If the longest match is found, then its entailed nodes can be added to the set of the matched nodes (if the parameter *bestNodeOnly* is set to false).

6.2.4.2 Class NodeMatcherLuceneSimple (eu.excitementproject.tl.composition.nodematcher)

This implementation compares an input fragment graph to an input entailment graph. It provides methods for transforming the entailment graph into a Lucene index and to convert a fragment graph into a Lucene query, which can be matched against this index. For efficiency reasons, it compares the base statement of the fragment graph only. If a matching node is found, it returns this node together with the confidence score of the match.

6.2.5 Category Annotator Modules

6.2.5.1 Class CategoryAnnotatorAllCats (eu.excitementproject.tl.composition.categoryannotator)

The *CategoryAnnotator* module adds category annotation to an input CAS based on an input set of *NodeMatch*-es. Each *NodeMatch* in the input set of *NodeMatch*-es holds exactly one *EntailmentUnitMention M* (found in the input CAS), which is associated to a set of *PerNodeScore*-s *P*. Each *PerNodeScore* in *P* refers to a tuple of an *EquivalenceClass E* (a node in a collapsed entailment graph) and a confidence score *s* denoting the confidence of *M* matching *E*.
For computing the final confidence score for a particular category, we need to combine the category confidence computed for a particular node with the confidence of the match.

In this implementation of the *CategoryAnnotator* module, final category confidence scores for a particular M are computed in two steps.

For each per node score p_y in *P*, we first compute a score per category c_x associated to this per node score by multiplying the confidence score of this category $f(c_x)$ (as computed using the ConfidenceCalculator module) with s_y, i.e., the confidence of the match:

$score(p_y,c_x) = f(c_x)*s_y$

Going through all per node scores in P, we then sum up all scores per category and divide them by the total number of per node scores in *P* to compute the final confidence score for the x^{th} category $score(c_x)$:$score(c_x) = \frac{\sum_{y=0}^{m} score(p_y,c_x)}{m}$

The final category scores are added as *CategoryAnnotation t*o the fragment in the input CAS to which *M* refers.

6.3 Implementation of Top Levels

6.3.1 Use Case 1: Class UseCaseOneRunnerPrototype (eu.excitementproject.tl.toplevel.usecaseonerunner)

The *UseCaseOneRunnerPrototype* provides an implementation of the *UseCaseOneRunner* interface. The interface's *buildRawGraph* methods produce an *EntailmentGraphRaw* object from input CAS objects or input *Interaction* objects, by producing sets of *FragmentGraph*-s for each input interaction and merging them to produce an *EntailmentGraphRaw*. The interface's *buildCollapsedGraph* methods produce an *EntailmentGraphCollapsed* object either directly from a raw graph or from input CAS objects / input *Interaction* objects, from which an *EntailmentGraphRaw* is built as described above and further optimized (merging nodes into equivalence classes, collapsing multiple edges between the same pair of nodes into one edge, etc.), to produce the final output.

- Attributes:

  To perform the transformations between the different types of graphs mentioned above, a few intermediate processing steps must be performed: fragment annotation, modifier annotation, generating fragment graphs from an input CAS object, merging of fragment graphs, and collapsing a raw graph. Each of these processing steps is encapsulated in an interface, which are attributes of the *UseCaseOneRunnerPrototype*, and are initialized by the *initInterfaces()* method. In addition to these interfaces, the class also has as attributes the EDA used in the merging step, and the LAP for producing the required input for the EDA. A summary of the attributes is presented below:

- LAPAccess lap
- EDABasic<?> eda
- FragmentAnnotator fragAnot
- ModifierAnnotator modAnot
- FragmentGraphGenerator fragGen
- GraphMerger graphMerger
- GraphOptimizer collapseGraph

- Methods:

  Besides implementing *UseCaseOneRunner*'s abstract method, the *UseCaseOneRunnerPrototype* implements the *initInterfaces()* method, which initializes the interfaces necessary for building the graphs. The method initializes a pre-specified set of interfaces:

  - FragmentAnnotator: *SentenceAsFragmentAnnotator* for adding (determined) fragment annotation of the input CAS
  - ModifierAnnotator: *AdvAsModifierAnnotator* for adding modifier annotations to each (determined) fragment
  - FragmentGraphGenerator: *FragmentGraphGeneratorFromCAS* for generating *FragmentGraph*-s for each (determined) fragment in the input CAS
  - GraphMerger: *StructureBasedGraphMerger* for merging *FragmentGraph*-s into an *EntailmentGraphRaw*
  - GraphOptimizer: *SimpleGraphOptimizer* to build an *EntailmentGraphCollapsed* from an *EntailmentGraphRaw*

6.3.2 Use Case 2: class UseCaseTwoRunnerPrototype (eu.excitementproject.tl.toplevel.usecaseonerunner)

The UseCaseTwoRunnerPrototype provides an implementation of the *UseCaseTwoRunner* interface, which has one interface method *annotateCategories(JCas cas, EntailmentGraphRaw graph)*. This method annotates categories on the given input CAS based on an input entailment graph using the following module implementations:

- *SentenceAsFragmentAnnotator*: This module adds fragment annotation to the input CAS.
- *AdvAsModifierAnnotator*: This module adds modifier annotation to the input CAS.
- *FragmentGraphGeneratorFromCAS*: This module generates fragment graphs for the annotated fragments, using the modifier annotation.
- *NodeMatcherLongestOnly*: This module compares the created fragment graphs to the input entailment graph to find matching nodes.
- *CategoryAnnotatorAllCats*: This module adds category annotation to the input CAS by combining category information from the matching nodes.

6.4 Implementation of Data Readers, and Other Utilities

6.4.1 Class CASUtils (eu.excitementproject.tl.laputils)

CASUtils is a class that contains a set of public methods. All of them are utility functions that are related to accessing CAS (JCas) data. The following summarizes the utilities provided by this static class.

- *JCas createNewInputCAS()*: this is the preferred method of generating a new JCas object.
- *void serializeToXmi(JCas aJCas, File f)*: this method serializes the JCas into CAS-standard serialization format of XMI (file extension is XMI, where the content is XML representation of the CAS data structure).
- *void deserializeFromXmi(JCas aJCas, File f)*: this method reads an XMI file and fills the content of the JCas with it.
- *annotateOneAssumedFragment(JCas aJCas, Region[] r), annotateOneDeterminedFragment(JCas aJCas, Region[] r)*: the two utility methods annotate CAS data with fragment annotation. Each method gets a list of “regions” and annotates one fragment. The first method annotates *assumedFragment*, and the second annotates *determinedFragment*.
- *annotateOneModifier(JCas aJCas, Region[] r, ModifierAnnotation dependOn), annotateOneModifier(JCas aJCas, Region[] r)*: the two utility methods annotate CAS data with a modifier annotation. The difference between the two methods is the capability to handle “dependency”, where one modifier depends on the other. See section 4.2.6 , for explanation of the dependency between modifier annotations.
- *annotateCategories(JCas aJCas, Region r, String text, Map<String, Double> decisions)*: this method annotates the region r in the CAS with category annotation based on the decisions input (as part of use case 2).
- *void dumpCAS(JCas aJCas)*: this is a method that prints details of the JCas into standard output stream. The method is provided mostly for debugging.
- *void dumpAnnotationsInCAS(JCas aJCas, int annotType)*: this method is a "smarter" version of *dumpCAS()*, which gets the type of the annotation and only prints out that type within the input CAS. Just like *dumpCAS()*, this method is also provided for mostly debugging and checking.

6.4.2 class InteractionReader (eu.excitementproject.tl.laputils)

For the moment, WP6 has one utility class that provides two readers for WP2 data. Both of them are static methods in the InteractionReader class.

- *List<Interaction> readInteractionXML(File xmlFile)*: This method reads a WP2 interaction XML file, and converts it into a list of Interaction data type objects. Note that the Interaction data type is a simple Java class that represents non-annotated (no fragment, no modifiers) interaction based on strings. Each Interaction object can be easily converted to JCas input CAS by calling one of its member methods (*fillInputCAS(JCas aJCas),* or *JCas createAndFillInputCAS*())
- *readWP2FragGraphDump(File interactionText, File graphsInXML, JCas aJCas, String languageID)*: This method reads WP2 fragment graph data. Note that WP2 fragment graph data has multiple files for one interaction. An interaction is given as a raw text file, and each fragment is given as one XML that denotes fragment graphs. To run this reader, one JCas, one XML file and one raw text file are to be provided.

7. Evaluation

This chapter describes the evaluation environment we developed for the two use cases and the results achieved on industrial datasets created within the project. The evaluation environment allows us to test and evaluate the performance of different implementations of the Transduction Layer modules, and thus to determine the Transduction Layer configuration that is expected to create highest benefit to the industrial use cases. The following matrix gives an overview of the Transduction Layer modules evaluated within WP6 and the datasets used for the evaluation per language.

	English	Italian	German
Fragment & Modifier Annotation	NICE interactions	ALMA interactions	-
Graph Merger & Optimizer	NICE graphs	ALMA graphs	-
Category Annotation	-	-	OMQ interactions

In addition to the evaluation of Transduction Layer module implementations, WP6 conducted evaluations of different EDA configurations on project data for all three languages and all EDAs provided within the EOP.

7.1 Data revision

After a first evaluation round, we realized that the datasets created within WP2 contained inconsistencies that should be fixed in order to make the results of our evaluation more meaningful. We therefore decided to spend some effort on revising and correcting the existing datasets for both use cases. This activity, being functional to the implementation of the transduction layer, was carried out under WP6, and is described in the following two sections. The revised datasets were used for evaluating the TL and can be found in the TL's github repository at https://github.com/hltfbk/Excitement-Transduction-Layer/tree/master/tl/src/main/resources/exci/

7.1.1 Public English and Italian Entailment Graph Datasets (Use case 1)

During the first evaluation phase, the analysis of the collapsed graphs resulting from WP2 final public datasets revealed non-optimal annotation. In fact, the global view allowed by collapsed graphs highlighted annotation problems that were not detected during dataset creation, which was based on "local" annotations. In order to further improve data quality for our evaluation, we decided to revise the original annotations.

WP2 public datasets included a Speech component for both English and Italian. Speech data were particularly problematic as the starting texts were automatic transcriptions of telephone calls and were thus very difficult to interpret and to be judged for entailment. Starting from the consideration that developing new and more performing speech recognition systems able to increase the quality of the transcribed text was not among Excitement objectives, we decided not to revise speech data and give priority to written data, more suitable for our current state of the art.

7.1.1.1 Data Revision Methodology

The data revision procedure was based directly on the collapsed graphs resulting from the original WP2 annotation. It included a number of steps aiming to ensure the correctness of both modifier annotation and entailment relations.

1. Text fragments composing each collapsed node were checked. First, if wrong modifiers were found, they were removed and their corresponding Fragment Graphs were modified accordingly. Then, we checked the semantic equivalence between them. Text fragments that were not equivalent to all the other texts composing the collapsed node were removed from that collapsed node. When possible, they were moved to other existing nodes, otherwise new nodes were created;
2. A new collapsed graph was automatically created according to the revised nodes;
3. Existing edges between original collapsed nodes were manually revised. New relations deriving from the creation of new nodes in step 1 were annotated from scratch.
4. A new collapsed graph was automatically re-created according to the revised/annotated edges. If inconsistencies (entailment transitivity violations) were found, edge annotation (step 3) was fixed until no inconsistencies were left;
5. WP2 official format made of not collapsed entailment graphs was recreated.

7.1.1.2 New Statistics

Table 1 below shows the new statistics about the composition of the datasets. Column 1 gives the number of clusters (subtopics) in which data were subdivided; Columns 2 and 3 present the number of original interactions and the fragment graphs derived from them. Columns 4 and 5 show the number of nodes and edges composing the entailment graphs as they result from the merging of fragment graphs. The total number of edges contained in the final graphs is further subdivided in Columns 6 and 7, which distinguish between *intra*-fragment edges, i.e. fragment graphs' edges (imported from the single fragment graphs into the final merged graph), and *inter*-fragment edges, i.e. edges connecting nodes belonging to different fragment graphs (created when merging the various fragment graphs composing each cluster).

	Clusters	Interactions	Fragment Graphs	Total Nodes	Total Edges	*Intra*-fragment edges	*Inter*-fragment
ENGLISH (Email)	29	389	457	756	7,830	402	7,428
ITALIAN (Social)	18	291	344	760	2,318	733	1,585

Table 1: Composition of the datasets

Table 2 below presents the data in terms of Textual Entailment (TE) pairs, which are created considering all the possible pairs of nodes belonging to different fragment graphs in both directions, that is: for each pair of inter-fragment nodes, two TE pairs are created. Column 1 gives the total number of inter-fragment TE pairs, Column 2 shows the number of TE pairs for which there is no entailment relation, while Column 3 presents the number of TE pairs where an entailment relation holds - corresponding to the inter-fragment edges in the graphs (see Table 1). Another interesting information from the textual entailment point of view is the subdivision between unidirectional entailment (Column 4) and bidirectional entailment (Column 5). In the first case the entailment relation holds only in one direction of the node pair (corresponding to one positive TE pair and one negative TE pair), whereas in the second case the relation holds in both directions of the node pair (corresponding to two positive TE pairs).

	Inter-Fragment TE Pairs	Non-Entailment TE Pairs	Entailment TE Pairs	Unidir Entailment	Bidir Entailment
ENGLISH (Email)	28,434	21,006	7,428	4,670	1,379 (*2)
ITALIAN (Social)	51,760	50,175	1,585	1,331	127 (*2)

Table 2: TE pairs in the datasets

7.1.1.3 Inter-Annotator Agreement

In order to assess the reliability of the revised data, we computed again Inter-Annotator Agreement (IAA) for the two main manual annotation steps, namely *(i)* the annotation of modifiers within text fragments, which is necessary to build the fragment graphs and *(ii)* the annotation of entailment relations between statements (nodes) belonging to different fragment graphs, which is required to merge the fragment graphs. To this purpose, two annotators independently annotated portions of the datasets as described in details below.

Annotation of modifiers within fragments

Agreement on modifier annotation was calculated on around the 30% of the fragments contained in each dataset, namely 130 fragments for English (6 clusters) and 94 fragments for Italian (5 clusters).

The metric used to calculate agreement was the Dice coefficient. The Dice coefficient is computed as DICE=2C/(A+B), where C is the number of common annotations, while A and B are respectively the number of annotations provided by the first and the second annotator.[4]

Two different types of agreement were calculated, to account for both complete agreement and partial agreement on the modifiers' extent.

To calculate complete agreement only perfect overlap between modifiers extent was taken into account (i.e. exact matches), whereas for partial agreement the overlap was calculated at the level of single words composing the modifiers extent. Table 3 presents the results obtained for each language, and shows a good quality of modifiers annotation.

	Fragments	Perfect overlap	Partial overlap
ENGLISH	130	0.85	0.91
ITALIAN	94	0.81	0.85

Table 3: Inter-annotator agreement for modifier annotation (Dice coefficient)

Entailment annotation for graph merging

Agreement on entailment annotation for graph merging was calculated on around 2,000 annotations for each language. These annotations correspond to the total number of entailment judgments necessary to merge the collapsed nodes composing 4 different clusters for English and 2 different clusters for Italian.

The agreement rates were calculated using the Kappa Coefficient (Scott's *Pi*), which is computed as P(A) - P(E) / 1 - P(E), where P(A) is the observed agreement, i.e. the proportion of items on which the two annotators agree, and P(E) is the estimated agreement due to chance, calculated empirically on the basis of the cumulative distribution of judgments by all raters.

According to the commonly accepted interpretation of Kappa given by Landis and Koch (1977)[5], agreement results are to be interpreted as follows:

[4] Note that the Dice coefficient has the same value of the F1 measure computed considering any of the two annotators as the reference.

- <0 No agreement
- 0.00 - 0.20 Slight agreement
- 0.21 – 0.40 Fair agreement
- 0.41 – 0.60 Moderate agreement
- 0.61 – 0.80 Substantial agreement
- 0.81 – 1.00 Almost perfect agreement

Table 4 presents the results obtained for each dataset. As we can see, both datasets are in the range of "substantial agreement", which shows the good quality of the annotated data.

	Annotations	Observed Agreement	Chance Agreement	***Kappa***
ENGLISH	2,294	0.9760	0.9028	**0.7534**
ITALIAN	1,794	0.9849	0.9348	**0.7693**

Table 4: IAA results for entailment annotation

7.1.2 Public German email dataset (Use case 2)

During the first evaluation phase, we detected a few errors and inconsistencies in the OMQ email dataset for use case 2, which we decided to correct in order to reach a better data quality for our evaluation. The corrections are described in the following. Please note that we only corrected errors that were introduced during data processing, NOT errors introduced by the authors of the emails themselves.

7.1.2.1 Lines starting with ">"-sign

A number of emails contained ">" signs at the beginning of the (original) lines of the user email, as shown in the following data sample:


```
- <text systemCategory="55" interactionId="232">
    > Lieber WAREHOUSE Helpdesk. Im User Forum gibt es hunderte von Fragen zum > Thema Asymmetrie. Imme
    Verkaufsware. Wir haben aber eben nur dieses > Material.
    <relevantText goldCategory="55" end="371" begin="233">Und so viele User beklagen, dass nach einem Import ei
       werden.</relevantText>
    Und zwar nicht mit einem > festen Abstand, sondern mit der Laenge des Prozesses ansteigend. Ich bin > siche
    Loesung aufzeigen, oder aber deutlich aussprechen, dass das Programm > hier an die Grenzen stoesst?! Mag j
    Systemen eine Loesung erarbeitet werden > kann. Aber wir alle sind eben keine Profis. Unsere Kartenlesegerä
    Haben Sie > vielen Dank fuer Ihre Hilfe. Das alles haette ich viel lieber > erzaehlt, aber eine Helpdesk Telefon
    eine gute Antwort ...
```


We decided to remove them automatically as they are not expected to be part of a user email in a real setting.

[5] J. Landis and G. Koch, "The measurement of observer agreement for categorical data", Biometrics, vol. 33 (1), pp. 159–174, 1977.

7.1.2.2 Missing white spaces between words

A number of emails contained blank spaces between the line ending and line beginning of the (original) user email, as shown in the following data sample:


```
- <text systemCategory="100" interactionId="612">
    Hallo mein Name ist Jessica Grosmann. Ich habe mir am Freitag, den 12.4.11 den SpecsManager 11 Ultimate gel
    <relevantText goldCategory="100" end="497" begin="165">Leidermusste ich feststellen das mein Konto anscheinei
       ihrer Seite registriert. Als ich dann meinen SpecsManager 11 registrieren wollte, wurde dies geblockt mit dei
       registrieren brauche.</relevantText>
    Ich würde Sie daherbitten dies richtigzustellen, dass heißt die Regiestrierung zurückzustellen oder mir denAcco
    52366-80540-26476-84177-73426 .
    <relevantText goldCategory="57" end="816" begin="719">Desweiteren habe ich noch ein Softwareproblem undzwa
    Es öffnet sich zwar ein kleines weißesFenster, aber nichts ist darauf zu sehen und im unteren linken Eck flacker
    auf 100 %. Ichhoffe, Sie können mir helfen. Für ein schnelle Antwort wäre ich sehr dankbar.Jessica Grosmann
```


We decided to remove them automatically as they are not expected to be part of a user email in a real setting.

7.1.2.3 Wrong processing of emails assigned to multiple categories

Several errors were introduced in the generation of the dataset when processing emails assigned to more than one category. One problem here was that identical emails occurred more than once in the dataset, and (in some cases) different occurrences of the same email were associated to different categories. In addition, for emails in which several “relevant texts” were marked during the manual annotation step, the second and all following occurrences of a “relevant text” in the same email were not processed correctly. The following table summarizes the different cases we detected in the original dataset and the actions we took to solve the respective problems:

# in data	Category	Relevant text (span)	Action taken
4	Same	Same	Removed additional email occurrences
3	Different	Different	Added several „relevantText“ annotations
3	Different	Same	Added several categories to „relevantText“ annotation
3	Different	Overlapping	Decided on one of the spans and annotated several categories

7.1.3 New statistics

Number of emails: 627

Number of relevant texts: 638

Number of Categories: 41 (arranged in 20 groups)

Number of words:

- in emails: 46,137
- in categories: 432
- total: 46,569

Number of Text/Hypothesis pairs: 24,553 (balanced: 1274)

Number of positive T/H pairs: 637

Number of negative T/H pairs: 23,916 (balanced: 637)

7.2 Evaluation of EDA configurations

This section describes the results of evaluating different EDA configurations on project data (for all three languages). The purpose of this analysis is to find the best configuration in terms of linguistic analysis pipeline (LAP), entailment engine (EDA) and resources, for use with the industrial data. Within WP4, each developer had performed extensive evaluation of their own EDA on "standard" entailment datasets. These datasets contained (grammatical and clean) sentences which in content and style resemble sentences in Wikipedia. This type of data is very different from industrial partner data. Industrial data are not necessarily grammatical, contain misspellings, deal with specific issues and contain less varied vocabulary.

There are several parameters for this analysis:

- preprocessing (LAP) – the entailment platform EOP implements several linguistic analysis pipelines for each language.
- entailment engine (EDA) – at the time of this study there are more than 5 EDAs, each with an adjustable configuration setting in terms of LAP, components, resources, and additional specific parameters (such as weights for example) or training (learning) algorithms.
- lexical resources – for each language we have general purpose resources (such as wordnets), but also resources developed specifically for the industrial partner, from corpora that reflect the topics represented in the partners' data.

A comprehensive analysis of all variations is not feasible. We adopt the strategy of using the base configuration for each EDA, plus one or more configuration recommended by the EDA developers based on the performance of their EDA on the general purpose datasets mentioned above.

Apart from the variations in the configurations and parameters related to the EDAs, there are also several possible experimental set-ups:

- learning variations:
  - single train / test cycle
  - cross-validation
- data preparation:
  - balanced / imbalanced
  - mixed / pure – in the course of the project we discovered that the entailment task can benefit from clustering the data by topic (such as *food*, *trains*, *internet connection, payments*). Because different clusters have no connection with each other, entailment relations could hold only between statements from the same cluster. This eliminates numerous comparisons and speeds up the process. Having the data organized in such disjoint clusters brings up the issue of data split: split the data from one cluster between training and testing (*mixed*), or include one cluster in its entirety either in the training or in the testing partition (*pure*).

Because of the size of the data (order of magnitude 10^4), we decided for a single train / test cycle. To have a clearer idea of the performance we chose to work with balanced data, where no bias exists towards one or the other of the entailment classes. With respect to mixed or pure splits, neither is a clear winner - in a real world setting both situations could occur: an interaction comes in on a topic not seen before (as in pure split, where the test data consists of not previously seen topics), while interactions on previously seen topics may continue to come in (as in mixed split, where topics encountered in the test data were encountered before). Because of this we try both these settings for the two datasets were clusters were available (ALMA and NICE).

The table below shows the data statistics for the experimental set-up chosen:

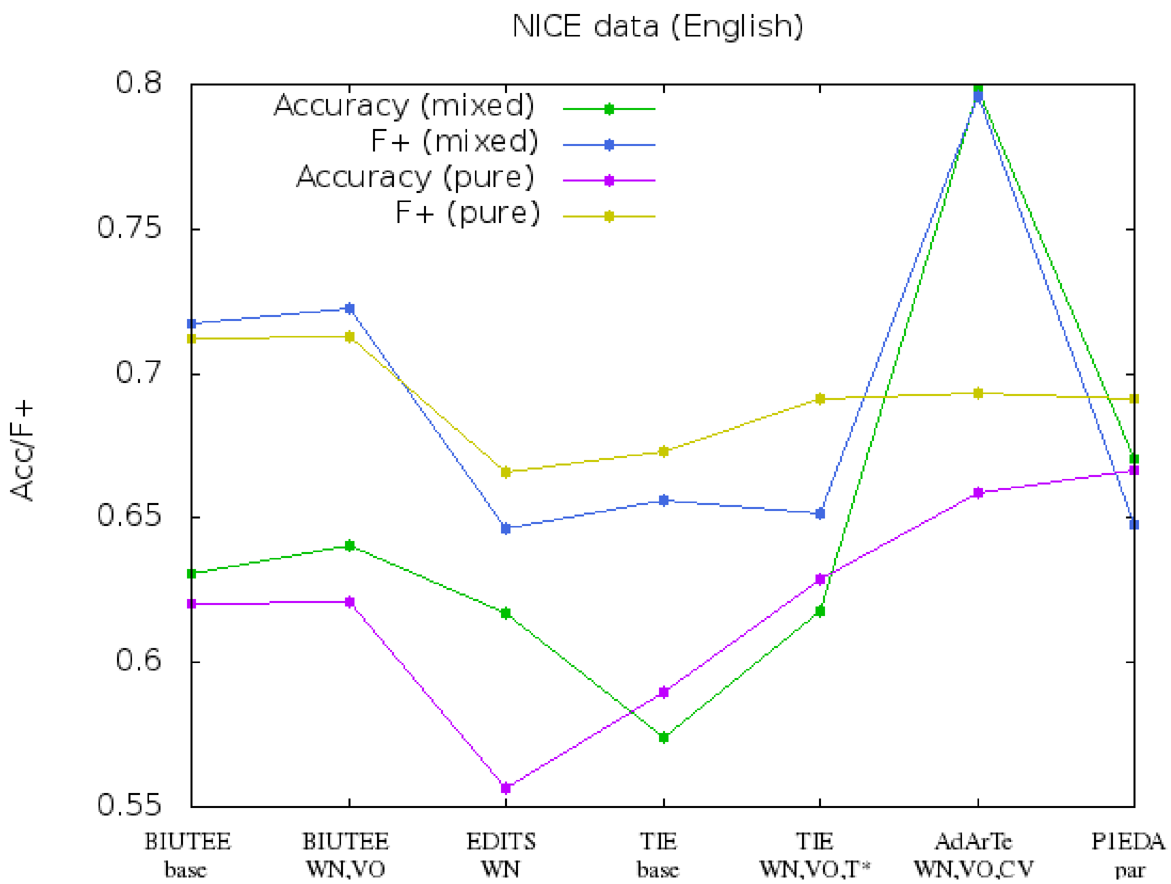
		Training (+)	Training (-)	Testing (+)	Testing (-)
ALMA	balanced/mixed	795 (45.8%)	941	790 (45.6%)	944
	balanced/pure	719 (41.4%)	1019	866 (50%)	866
	imbalanced/pure	1070 (3.1%)	33814	515 (1.5%)	34355
NICE	balanced/mixed	2620 (49.4%)	2680	2614 (49.3%)	2686
	balanced/pure	2576 (48.8%)	2708	2642 (50%)	2642

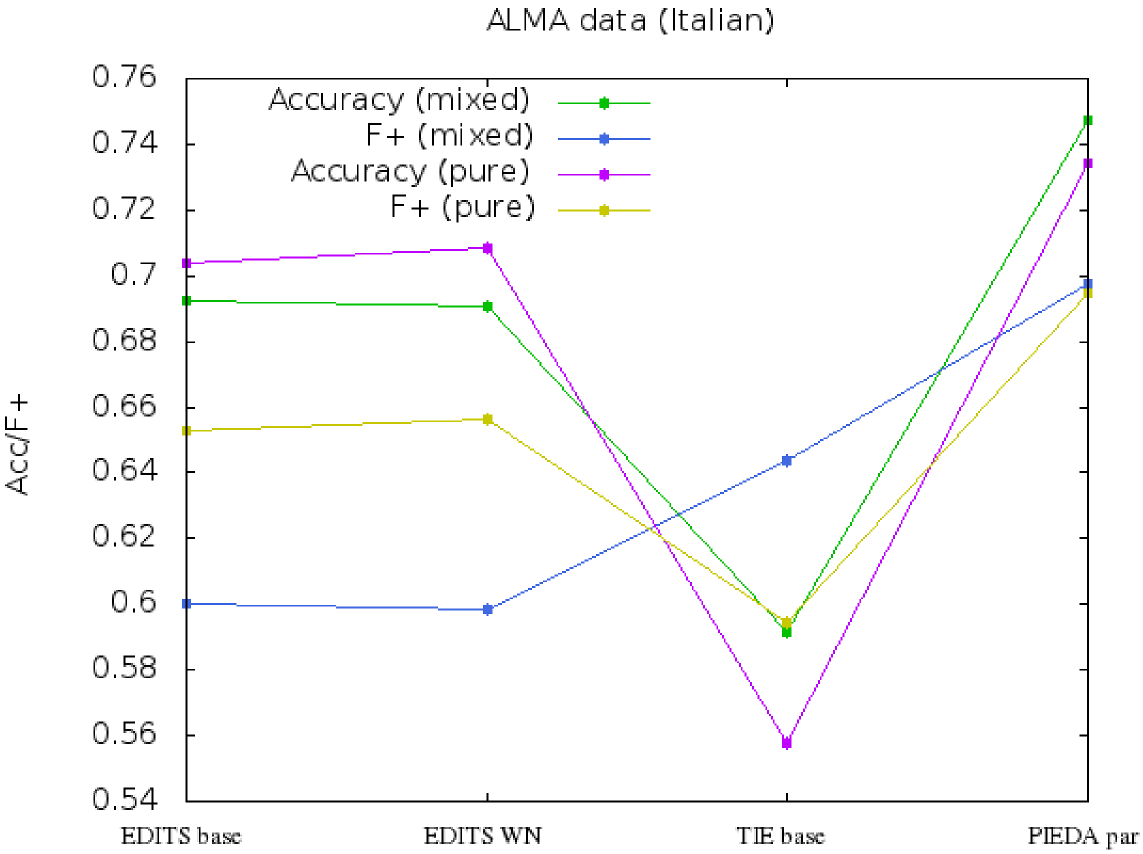
	imbalanced/mixed	2632 (25.6%)	7666	2622 (25.2%)	7674
	imbalanced/pure	3947 (38.3%)	6359	1307 (12.7%)	8989
OMQ	Balanced	319 (50%)	319	318 (50%)	318
	Imbalanced	319 (2.6%)	11958	318 (2.6%)	11958

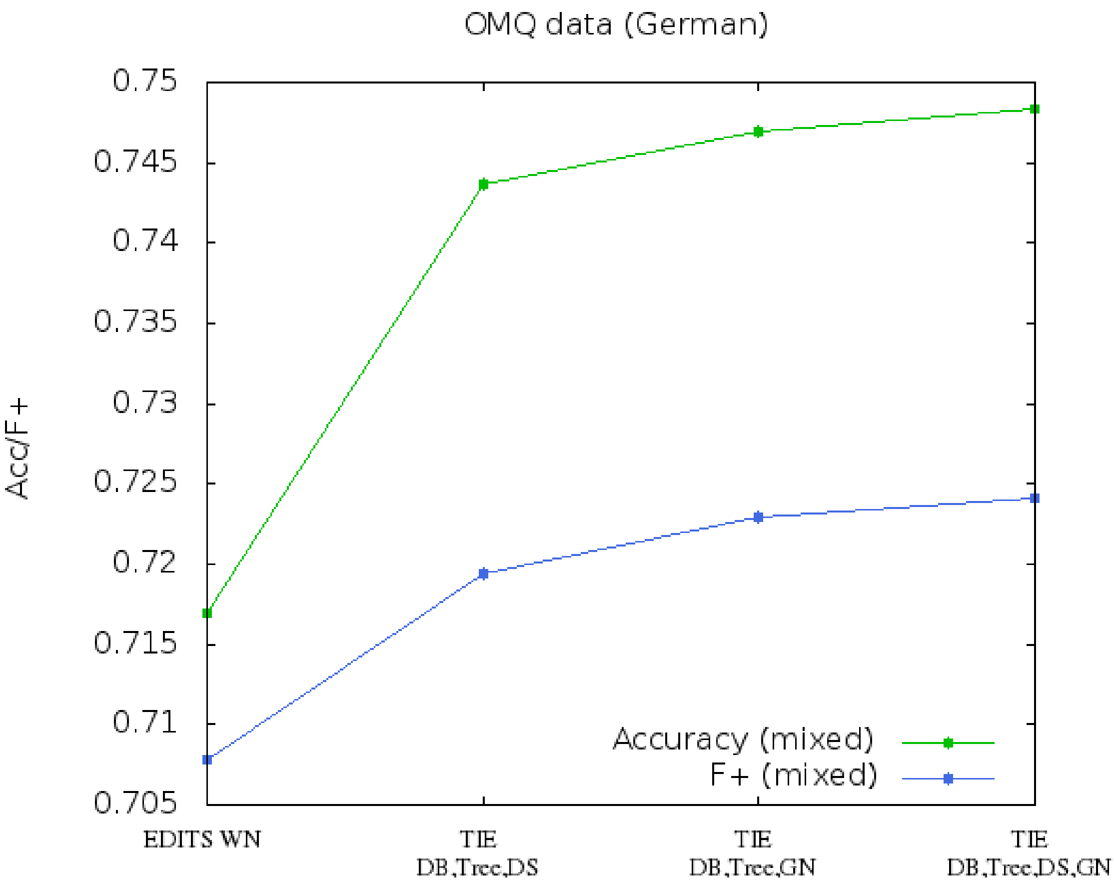
The details of the experiments performed are reported here:

https://github.com/hltfbk/Excitement-Open-Platform/wiki/Development-test-suites-and-reports

In the following figures we present a summary of these experiments, in terms of accuracy (Acc) and f-score on the positive (ENTAILMENT) class (F+). Not all EDAs could be used for all languages.




ALMA data (Italian)
Acc/F+
Accuracy (mixed)
F+ (mixed)
Accuracy (pure)
F+ (pure)
0.76
0.74
0.72
0.7
0.68
0.66
0.64
0.62
0.6
0.58
0.56
0.54
EDITS base
EDITS WN
TIE base
PIEDA par


OMQ data (German)
Acc/F+
Accuracy (mixed)
F+ (mixed)
0.75
0.745
0.74
0.735
0.73
0.725
0.72
0.715
0.71
0.705
EDITS WN
TIE DB,Tree,DS
TIE DB,Tree,GN
TIE DB,Tree,DS,GN

The results of training and testing EDAs on industrial partner data have guided us to choose an EDA set-up for use within the transduction layer for each industrial partner. The considerations for the choice were not driven only by performance, but also by complexity and computation time, and the availability of the resources required for use for commercial purposes. For our use case 1 experiments on graph buiding, we use P1EDA for Italian and English, which achieves the highest accuracy results on the "pure" split. P1EDA is an alignment based EDA that relies on automatically built and freely available paraphrase tables for English and Italian. For German, we use the best-performing TIE EDA, which uses the following resources: DB (DerivBase), Tree (dependency relation features), DS (Distributional Similarity), and GN (GermaNet).

7.3 Evaluation of Transduction Layer modules

This section describes the evaluation environment developed for the evaluation of Transduction Layer modules as well as the results achieved on the revised industrial datasets.

7.3.1 Overview of evaluation measures

WP2 Dataset	Evaluated module	Evaluation measure
Decomposition		
Annotated fragments and modifiers (NICE / ALMA)	Fragment Annotation + Modifier Annotation	$P = \frac{\#\ of\ correctly\ assigned\ fragment\ (modifier)\ tokens}{\#\ of\ assigned\ fragment\ (modifier)\ tokens}$ $R = \frac{\#\ of\ correctly\ assigned\ fragment\ (modifier)\ tokens}{\#\ of\ gold\ standard\ fragment\ (modifier)\ tokens}$
Composition (use case 1)		
Annotated graphs (NICE / ALMA)	Graph Merging + Graph Optimization	$P = \frac{\#\ of\ correctly\ added\ edges}{\#\ of\ added\ edges}$ $R = \frac{\#\ of\ correctly\ added\ edges}{\#\ of\ edges\ in\ gold\ standard\ graph}$ $Acc = \frac{\#\ of\ correct\ entailment\ decisions^1}{\#\ of\ entailment\ decisions^1\ in\ gold\ standard\ graph}$ [1]) *Including both positive and negativ decisions*
Composition (use case 2)		
Annotated emails (OMQ)	Category Annotation	$Acc = \frac{\#\ of\ correctly\ assigned\ categories}{\#\ of\ assigned\ categories}$

7.3.2 Evaluation of Decomposition modules

7.3.2.1 Fragment Annotation

The transduction layer offers three implementations for fragment annotation that are relevant for use case 1 (details are given in section 6.1.1):

- Sentences as fragments (sentence)
- Fixed-length fragments centered on (manually provided) keywords (KBFL)
- Phrases built using dependency relations, starting from (manually provided) keywords (KBDep)

Each has advantages and shortcomings – the sentence annotation will have perfect recall, but in the industrial data we noticed very long interactions without punctuation, where the relevant fragment is actually short. An example from the ALMA data illustrates this point. The fragment of interest is underlined:

> il mio **rimborso** non e' ancora arrivato e'quasi 1 anno che aspetto ed e' passato gia' 2 mesi da quando mi sono rivolta a voi senza nessun risultato cosa posso pensare di TELEFONIAX? N pratica 5C9-v3tev-4184-grazie.

One sentence could also contain more than one fragment, in which case the sentence-as-fragment annotation is too coarse, as this example from the NICE data shows:

> **Seats** need to be more comfortable and it would be better if there were staff walking around selling **snacks**.

The keyword-based annotations are more precise, but they rely on the availability of manually provided keywords, or a less than perfect automatic keyword annotator.

An overview of the performance of the three fragment annotators on the industrial partner data is included below. We compute the micro and macro Recall, Precision and F-score. For the micro evaluation we computed overall true positive, true negative, false positive and false negative scores by verifying the gold standard and automatic annotation for each token (belongs to a fragment or not). For the macro evaluation we compute similar scores for each fragment instance, and then average them for the overall measures.

The ALMA data consisted of 21060 tokens, 3039 of which belonged to one of 327 fragments (an average of 9.3 tokens per fragment).

Annotator	R (macro)	P	F-score	R (micro)	P	F

Sentence	1.0	0.2263	0.3692	1.0	0.1443	0.2522
KBFL 4	0.6753	**0.7063**	0.6905	0.6094	**0.6477**	0.628
KBFL 5	0.7331	0.6682	**0.6991**	0.6781	0.6088	0.6416
KBFL 6	**0.7755**	0.6356	**0.6986**	**0.7334**	0.5738	**0.6439**
(rel to BP)	0.7984	0.5870	0.6765	0.7641	0.4946	0.6005
KBDep	0.0999	0.9701	0.1811	0.0822	0.2014	0.1501

Apart from the comparison to the gold-standard fragments, for the best performing fragment annotator (keyword based fixed length with a window of 6 tokens before and after the keyword) we added the comparison to the base predicate, in the idea that it is more important that the automatic fragment covers the base predicate, missing modifiers are less important. These results are in the row *rel to BP* (relative to the Base Predicate).

The NICE data consisted of 12052 tokens, 3112 of which belonged to one of 440 fragments (an average of 7.1 tokens per fragment).

Annotator	R (macro)	P	F-score	R (micro)	P	F-score
Sentence	1.0	0.3536	0.5224	1.0	0.2582	0.4104
KBFL 4	0.7837	**0.7508**	0.7669	0.7043	**0.7130**	0.7086
KBFL 5	0.8349	0.7214	**0.774**	0.7696	0.6794	0.7217
KBFL 6	**0.8689**	0.6987	**0.7745**	**0.8174**	0.6521	**0.7255**
(rel to BP)	0.8784	0.6661	0.7577	0.8169	0.6004	0.6921
KB	0.2789	0.9256	0.4286	0.2766	0.6488	0.3879

The comparison to the base predicate are on the row *rel to BP* (relative to the Base Predicate).

For both datasets we notice high performance for the keyword-based fixed length fragment annotator - it balances nicely precision and recall. As it only requires token annotation, it is also the fastest and computationally simplest of all those implemented.

7.3.2.2 Modifier Annotation

In annotating modifiers, the target is to mark all words or phrases that can be removed without significantly changing the meaning of the text fragment. The assumption is that the entailment relation will hold between versions of the text fragment with sets of modifiers differing by one. The first implementation for modifier annotators marked all adverbs as modifiers, because at a first and superficial look, they do add detail to the sentence, but not of a nature that influences the overall meaning of the sentence. In the next step we added adjectives to the modifier pool, following the shallow observation that most adjectives are predicative, but do not change the meaning of the compound they are part of – a *blue book* is always a *book*. And in the last phase we added prepositional phrases, many of which add temporal or location details that can be omitted without altering the meaning of the remaining fragment, as in the example:: *I took the train* *to Rome* *in the morning.* (the "removable" modifiers are underlined) implies that *I took the train.*

This view of modifiers is of course overly simplistic, and on proper inspection the issue of modifier "removability" is a hard one. Our additional investigations into this matter have not materialized into an implementation for the transduction layer yet.

The difficulty of annotating modifiers is further increased when processing industrial data, due to spelling and grammatical errors, as the following examples illustrate. Gold standard modifiers are highlighted in blue, fragments are underlined, and keywords are in bold.

> (NICE) + Smiling and courteous staff. Clean toilets. Efficient processing. Good signage. - Ticket was expensive. Passageway within train too narrow for people to get through with luggage, limited luggage space so would not recommend if travelling with large

> (ALMA) Non ci credo! Ho aderito ad un'offerta simile di TELEFONIAX anni fa e QUANTO PROMESSO **nel'offerta** commerciale NON E' MAI STATO MANTENUTO malgrado diversi solleciti.. a Roma la chiamano "sola"..

The tables below show the evaluation in terms of micro Recall, Precision and F-score – we computed overall true positive, true negative, false positive and false negative scores by verifying the gold standard and automatic annotation for each token (belongs to a modifier or not). The numbers are based on the evaluation done on the subset of the modifiers that was revised in the data revision phase.

Because modifiers are only annotated inside fragments, we include results obtained with different fragment annotations:

- sentences-as-fragments – this annotator has 100% recall in terms of fragments, and would give the best estimation for each method's recall potential.
- KBFL6 – the best performing fragment annotator – would give us a realistic estimation of the performance of the modifier annotators in the "real world" setting
- GS – the gold standard – we used the gold standard fragment annotation to give us the best estimate of the performance of the modifier annotators in terms of precision.

We also tried to estimate the influence of negations, more specifically their scope, on the modifier candidates. The lines "adv,adj,pp,-neg" annotate adverbs, adjectives, prepositional phrases only if they do not fall in the scope of a negation. The results show that avoiding modifiers in the scope of a negation leads to an increase in precision, with a slight recall drop.

Within the ALMA data there were 538 tokens belonging to one of 181 modifiers (an average of 2.97 tokens per modifier).

Frag.	Modifier	R (micro)	P	F
Sent.	Adv	0.0824	0.0418	0.0555
	adv, adj	0.1704	0.0478	0.0747
	adv, adj, pp	**0.7266**	0.0654	0.12
	adv, adj, pp, -neg	0.6891	**0.0691**	**0.1257**
KBFL 6	Adv	0.0543	0.1229	0.0753
	adv, adj	0.1217	0.1529	0.1356
	adv, adj, pp	**0.5431**	0.2542	0.3463

	adv, adj, pp, -neg	0.5169	**0.2845**	**0.367**
GS	Adv	0.0824	0.2234	0.1204
	adv, adj	0.1704	0.2571	0.205
	adv, adj, pp	**0.7228**	0.3655	0.4855
	adv, adj, pp, -neg	0.6798	**0.4002**	**0.5038**

For NICE there were 586 tokens belonging to one of 239 modifiers (an average of 2.45 tokens per modifier).

Frag.	Modifier	R (micro)	P	F
Sent.	adv	0.1343	0.0894	0.1074
	adv, adj	0.2186	0.0595	0.0936
	adv, adj, pp	**0.7952**	0.1004	0.1782
	adv, adj, pp, -neg	0.7642	**0.1012**	**0.1787**
KBFL 6	adv	0.1205	0.2593	0.1645
	adv, adj	0.1910	0.1537	0.1704
	adv, adj, pp	**0.6747**	0.2776	0.3934
	adv, adj, pp, -neg	0.6540	**0.2846**	**0.3967**
GS	adv	0.1343	0.3362	0.1919
	adv, adj	0.2186	0.2022	0.2101
	adv, adj, pp	**0.7848**	0.3483	0.4825
	adv, adj, pp, -neg	0.7659	**0.3571**	**0.4871**

As expected, adding new types of potential modifiers (adverbs, adjectives, then prepositional phrases) leads to increases in recall. The highest increase in recall is obtained for both English and Italian data when adding prepositional phrases, indicating that a large part of the modifiers are indeed such phrases. Missed modifier tokens are caused by part-of-speech tagger errors for adverbs and adjectives, dependency parser errors, which causes us to create incomplete or partly erroneous prepositional phrases, and nominal modifiers which we did not attempt as yet to add.

The low precision scores throughout suggest that only a part of the adverbs, adjectives and prepositional phrases annotated as modifiers can be removed without losing the gist of the fragment. An interesting phenomenon can be observed for English, where adding adjectives leads to a drop in precision, compared to having only adverbs as modifiers. Inspection of the data shows that numerous adjectives are in fact the core of the base statement, as they specify the problem, e.g. *small* in the statement *"small legroom"* (adjectives in predicative position - *legroom is small* - are not marked as modifiers).

7.3.3 Evaluation of Composition modules (Use case 1)

The evaluations of the composition modules for use case 1 were performed using the datasets described above:

- NICE (English): 17 clusters in development set, 12 clusters in test set
- ALMA (Italian): 11 clusters in development set, 7 clusters in test set

Following the evaluation of EDA configurations (see section 7.2), we report the results of the best-performing P1EDA for graph construction.

Clusters from the development set were used to generate training data for the EDA. To construct the training data we used all the inter-fragment edges in the development set as positive (entailing) examples, and a similar amount of randomly selected node pairs for which there was no edge in the dataset as negative (non-entailing) examples. Such 'balanced' training set construction showed better results than using all available positive (edge) and negative (no-edge) examples in our preliminary experiments over the development set.

Then, we applied merging and optimization procedures to construct entailments graphs for each of the clusters in the test set. Gold-standard fragment graphs were used as input. We evaluated the resulting graphs and report macro-averaged results across the clusters of the test set.

7.3.3.1 Graph Merging

The table below compares the performance of the *AllPairsGraphMerger* (*All-pairs*) and the *StructureBasedGraphMerger* (*Str-based*). In this evaluation fragment graph edges were excluded from the calculation, since they are given as input. We see that *StructureBased-GraphMerger* indeed allows considerably reducing the number of EDA calls, while outperforming the *AllPairsGraphMerger*[6].

Dataset	F1		Accuracy		EDA calls	
	All-pairs	Str-based	All-pairs	Str-based	All-pairs	Str-based
NICE	0.58	**0.67**	0.72	**0.76**	7,510	2,282
ALMA	0.44	**0.51**	0.79	**0.84**	11,384	4,355

7.3.3.2 Graph Optimization

The tables below evaluate the quality of the final graphs (*Entailment graph*) obtained after merging and optimization, including the fragment graph edges. This is compared to a graph, which only includes fragment graph edges given as input (*FG edges only*).

Dataset	Graph	Str-based merger			Global optimizer		
		R	P	F1	R	P	F1
NICE	FG edges only	0.00	1.00	0.00	-	-	-
	Entailment graph	0.65	0.52	0.58	0.64	0.58	0.61
ALMA	FG edges only	0.14	1.00	0.25	-	-	-
	Entailment graph	0.78	0.46	0.58	0.48	0.80	0.60

[6] The advantage of StructureBasedGraphMerger is statistically significant for both datasets (p=0.01), according to the Wilcoxon singed-rank test.

Dataset	Graph	Str-based merger	Global optimizer
		Accuracy	Accuracy
NICE	FG edges only	0.756	-
	Entailment graph	0.764	**0.782**
ALMA	FG edges only	0.933	-
	Entailment graph	0.842	**0.941**

The results show considerable recall gain achieved by the merged graphs.

We also see that global optimization yields improved performance (significant with p=0.01 for both datasets).

7.3.4 Evaluation of Composition modules (Use case 2)

This section describes the evaluation performed for use case 2. The goal for use case 2 was to produce entailment graphs, integrate them into the email categorization procedure and evaluate their effect on categorization accuracy.

7.3.4.1 Dataset preparation

The dataset we used for the evaluation was the email data provided by OMQ. For evaluation, we split the dataset into three parts: one part for training the EDA, one part for building the graph, and the last part for testing. Based on this split, we performed a three-fold-cross-validation.

7.3.4.2 Evaluation parameters

In WP6, our evaluation focus was on four issues:

1) Fragment annotation: The purpose of this evaluation was to find out which fragment annotator would produce the most promising fragments as input for the graph building. As, unlike in use case 1, we did not have any manually annotated fragments to which we could have directly compared the automatically extracted ones, we did an extrinsic evaluation showing the effect of using different fragment annotators on the email categorization accuracy computed for the dataset.

2) Graph construction: The purpose of this evaluation was to compare different approaches towards building graphs. This included the choice and adaptation of EDAs from the open platform, but also the implementation of a "light version" EDA to be used for efficiency reasons. As, unlike in use case 1, we did not have any manually annotated graphs to which we could have directly compared the automatically constructed ones, we did an extrinsic evaluation showing the effect of using different approaches for computing edges on the email categorization accuracy computed for the dataset.
3) Computing category scores: A third parameter that is essential when using entailment graphs for email categorization is the strategy used for determining the best category (or categories) for each incoming email. We use a TF-IDF-based approach towards ranking the potential categories and evaluated the effect of using different TF-IDF variants on the email categorization accuracy computed for the dataset.
4) Direction of entailment: In most experiments shown below we use bidirectional entailments only, but we also evaluate the use of directional entailment.

As all of the above parameters interact with each other, we experimented with several combinations.

7.3.4.3 TF-IDF-based category ranking

In email categorization, the task is to assign matching categories to incoming emails. For determining the best-matching category (categories) for an incoming email, we approach the task as an information retrieval problem. In information retrieval, the task is to rank documents according to how well they match a user query. In the case of email categorization, the email corresponds to the user query, the documents correspond to the categories (each category is represented as the conjunction of the category description and all email texts associated to a particular category). The task then is to rank the categories according to how well they match the email text. We use a TF-IDF-based ranking approach, as is common in information retrieval.

For computing the baseline, we use a bag-of-words (BOW) representation for both the incoming email and the category, i.e. each (content word) token is considered a term when computing the category score.

As in the software of the industrial partners, the three best-matching categories are presented to the user (the support agent), we computed the top-n accuracy scores for n = {1,2,3}, where an email is scored positively if the top-n retrieved categories contain the correct category of the email.

In the table below we show the baseline results (BOW tokens only) achieved on the OMQ dataset.

	Top-1 accuracy	Top-2 accuracy	Top-3 accuracy
BOW baseline (ntn.ntn)	0.5701	0.7112	0.7742
BOW baseline (ltc.ltc)	**0.6149**	**0.7295**	**0.7953**

The results show that using the TF-IDF variant ltc.ltc (logarithmic term frequency and cosine normalization) achieves significantly better results than using standard TF-IDF (natural term frequency, no normalization). In the following tables, we will therefore show results for the ltc.ltc variant only.

7.3.4.4 Evaluating the usage of entailment graphs

7.3.4.4.1 Experiment 1: Fragments = Sentences

In this experiment, we use the BOW tokens plus the fragments extracted using the prototype implementation of the fragment annotator (*SentenceAsFragmentAnnotator*) as "terms". For building fragment graphs, we use the *AdvAdjAsModifierAnnotator* and the *Fragment-GraphLiteGeneratorFromCAS*. For computing entailment edges, we used the TIE configuration that produced the best results on the OMQ dataset in our EDA configuration evaluation (cf. section 7.2). The used confidence threshold was 0.8, which turned out to produce the most meaningful results. In this and all following experiments, we used the *LegacyGraphMerger* and *SimpleGraphOptimizer* for building entailment graphs.

In the categorization phase, fragments appearing in the same equivalence class in the graph (i.e., fragments considered equivalent) are treated as a single term, i.e., TF-IDF scores are computed at the level of equivalence classes.

	Top-1 accuracy	Top-2 accuracy	Top-3 accuracy
TIE_original (DB, tree, GN, DS)	0.6236	0.7388	0.8002

The results show that with this configuration, there is little improvement over the baseline. Analyzing the resulting graph, we noticed that, due to the internal complexity of the extracted fragments (some of the extracted sentences contained several dozens of tokens), only very few entailment relations were detected between the sentences in the dataset.

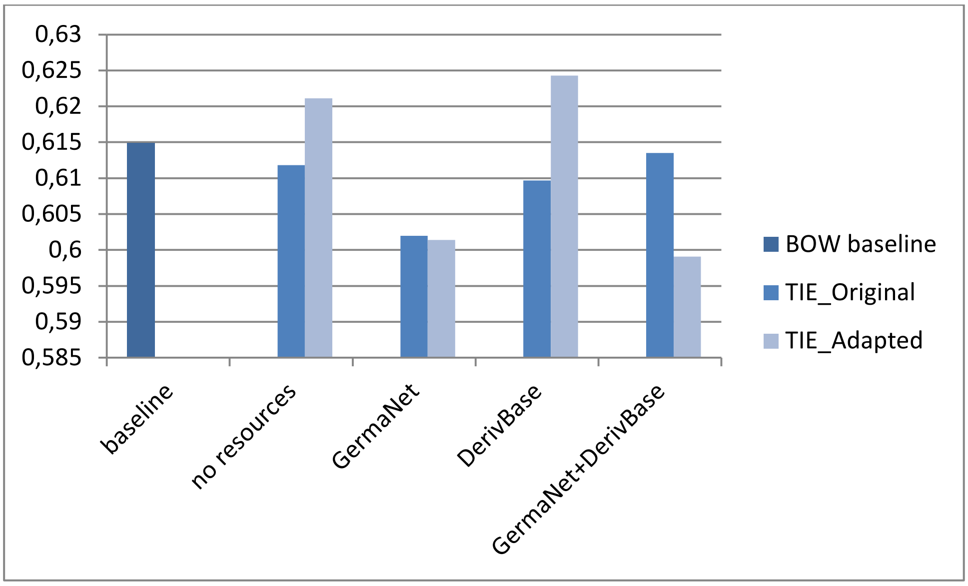
We therefore decided to go for a bottom-up approach: Starting with the smallest possible unit for a fragment, namely a single token, we would increase the complexity of the fragments.

7.3.4.4.2 Experiment 2: Fragments = Tokens

In our second experiment, we extracted single-token fragments (nouns, full verbs, adjectives and numbers) using the *TokenAsFragmentAnnotator* and four different configurations of TIE: TIE_Base (inflection only), TIE with GermaNet, TIE with DerivBase and TIE with both GermaNet and DerivBase. For our experiments, we used two versions of the TIE EDA: First, the TIE EDA as provided by the EOP and, second, the TIE EDA with the following adaptations:

- DerivBase: The original version of TIE uses a part-of-speech (POS) restriction for DerivBase, i.e., it only considers relationships between words in the same derivational family if they have the same POS. In the adapted TIE, we allow to also consider relationships between tokens with a different POS, e.g., „Erweiterung“ [*extension*] and„erweitern“ [*extend*]).
- GermaNet: The original version of TIE allows two configurations of GermaNet, one considering relationships between words with different POS tags, and another one considering only relationships between words with the same POS tag. In combination with DerivBase, it only allows for the GermaNet configuration with the POS restriction (same POS). In the adapted TIE, we allow for a configuration with no POS restriction for both GermaNet and DerivBase.
- Numbers: Using the original version of TIE, some number tokens (tokens recognized as “CARD” tokens by the tokenizer of the TreeTagger) are merged into the same equivalence class even though they express different numbers. In our adapted version of TIE, we only merge number tokens if they express the same number.

The following figure compares the results using TIE_Original and TIE_Adapted using different resources:



The results show that, while we are not able to beat the baseline using the original TIE EDA, with our TIE adaptations, we can achieve some improvements.

- Adaptation with respect to "CARD" tokens → large effect ("No resoures" configuration)
- GermaNet overgenerates -- > lot of wrong edges / equivalence classes
- DerivBase helps if no POS restriction
- Combination of germanet and Dervibase → strengehns the effect of ovegeeation

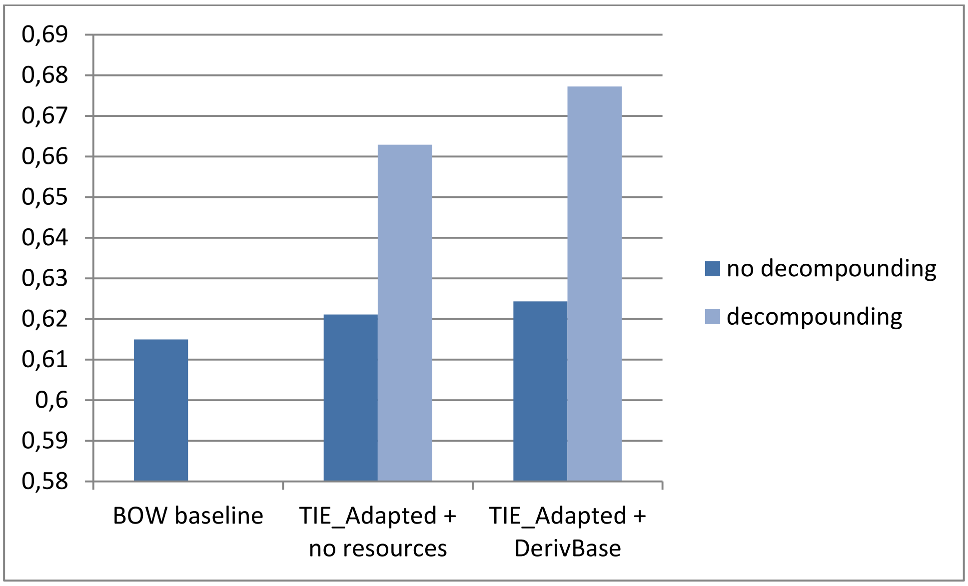
The best overall result (a top-1 accuracy of **62.43%**) is achieved with TIE_Adapated and DerivBase and is slightly better than the result achieved with sentence fragments.

7.3.4.4.3 Experiment 3: Fragments = Tokens + Compound components

In the next step, we evaluated the usage of decompounding. This was expected to have an effect on the results because compound words (written as a single token in German) are a common phenomenon of the German language and also appeared very frequently in our dataset.

In this experiment, we used the *TokenAsFragmentAnnotatorForGerman*, which annotates compound components in addition to tokens as fragments. This way, rather than only processing the compound itself, the compound plus all components are processed as separate fragments, both when building graph and when computing category scores for a new email. For example, during graph building, as soon as the token "Fehlermeldung" appears for the first time, rather than adding only a single node for "Fehlermeldung" to the graph, we now add three nodes: "Fehlermeldung", "Fehler" and "Meldung", each containing a mention referring to the interaction containing the token. Similarly, when computing category scores for a new email, if this email contains the token “Fehlermeldung”, we now extract and combine category distributions for “Fehlermeldung”, “Fehler”, and “Meldung” for computing the final score.

The results show that using decompounding has a large positive effect on categorization accuracy:



The best overall result (a top-1 accuracy of **67.72%**) is achieved using TIE_Adapted + DerivBase and Decompounding.

7.3.4.4.4 Experiment 4: Including edge information at token level

In the experiments described so far, we showed the effects of using bidirectional entailment information: Fragments occurring in the same equivalence class in the generated entailment graph were considered to be equivalent when computing TFIDF scores.

In this section, we show the results of using directional entailment information.

We integrated directional entailment information by also considering entailed edges when computing the final category confidence score. Whenever a matching node is found for a fragment annotated in the incoming email, we compute the *final score* for a particular category c_n based on the best-matching node plus all (directly) entailed nodes, weighting the entailed nodes based on the associated entailment confidence:

final_score(c_n) = $\frac{\sum_{x=1}^{m} edge\_score_x * node\_score(x,c_n)}{m}$, where

- m: number of matching nodes (best-matching and entailed)
- $node\_score(x,c_n)$: score associated to category c_n in node x

- *edge_score(x)*: score associated to the entailment edge between the best-matching node and the entailed node. This score is 1.0 for the best-matching node.

Carrying out several experiments using edge information, we did not see any clear positive effect on accuracy. While there were small improvements with some configurations, the accuracy degraded in others. A deeper analysis of the resulting graphs and categorization output would be required to get a deeper understanding of this matter.

7.3.4.4.5 Experiment 5: Fragments = Dependency relations

In the experiments we described so far, we used individual tokens (and components of compound words) as fragments.

In the next step, we evaluated the usage of more complex fragments consisting of two tokens. Two-token fragments were created by extracting all combinations of two content word tokens linked via a dependency relation in a sentence of the dataset, e.g. “System meldet” [*system shows*] and “meldet Fehler” [*shows error*] from the sentence “System meldet Fehler” [*system shows error*].

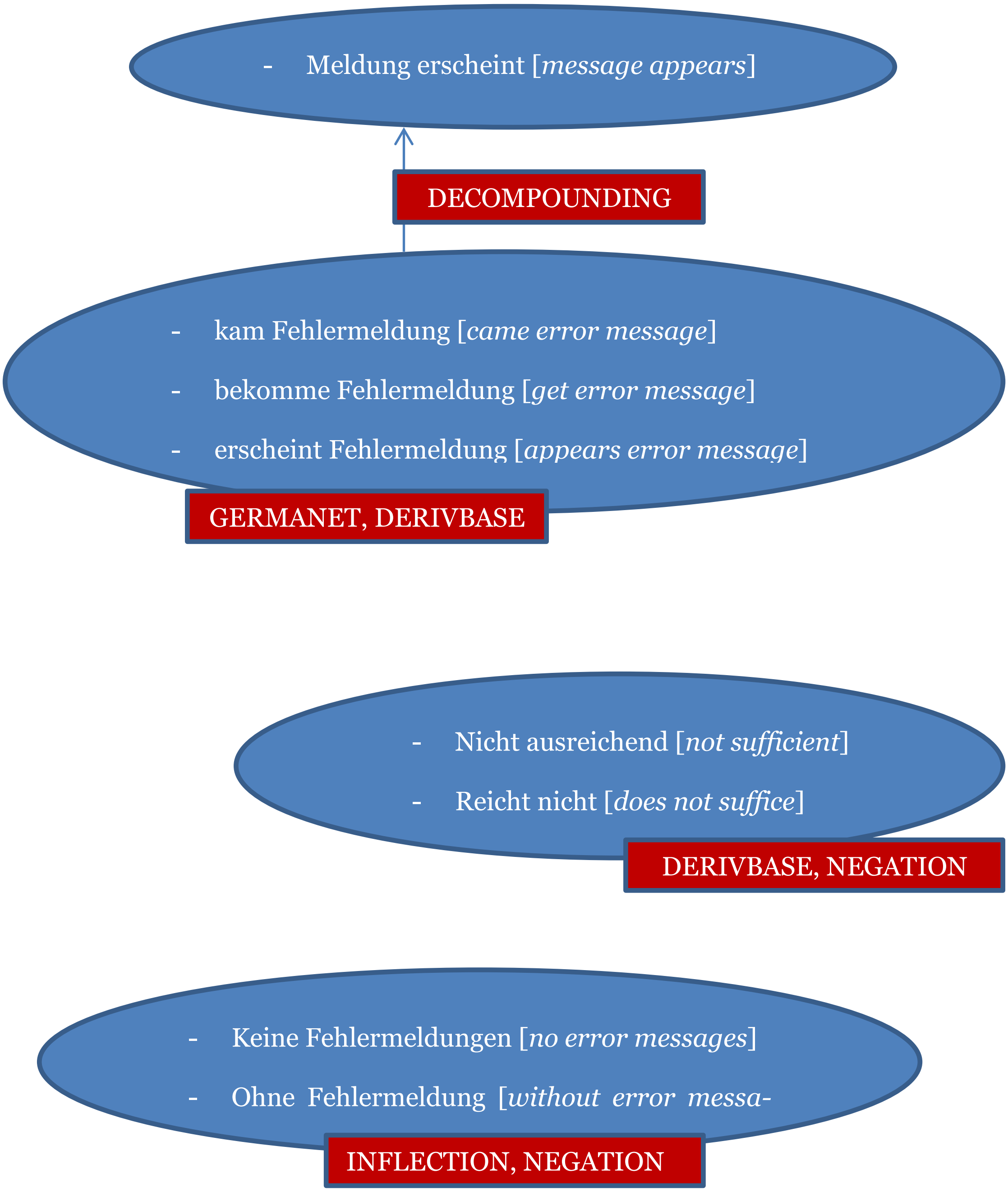
When processing these kinds of fragments with the EDAs provided by the EOP, we ran in to several problems. First of all, processing the huge number of two-token fragments extracted from the dataset was both extremely memory- and time-consuming. Second, as the existing EDAs were developed for and had been trained on sentence input, the output produced for these smaller fragments was not satisfying.

We therefore decided to implement a more efficient algorithm to process two-token fragments. This algorithm addresses the memory and time issue by processing T/H pairs as String objects rather than creating a CAS object for each T/H pair. It returns ENTAILMENT if each token in H can be mapped on some token in T (via some of the integrated resources). It can be configured for using: exact matching, Lemmatizer, DerivBase, GermaNet. For checking DerivBase or GermaNet relations, it uses classes provided by the EOP.

This algorithm requires no training and turned out to work well for short fragments (consisting of one or two tokens only).

For our experiments, we used the following POS filter for tokens: nouns, full verbs, adjectives, numbers, negation tokens (e.g., “ohne” [*without*], “kein” [*no*], “nicht” [*not*] “nichts” [*nothing*]), and particles of particle verbs.

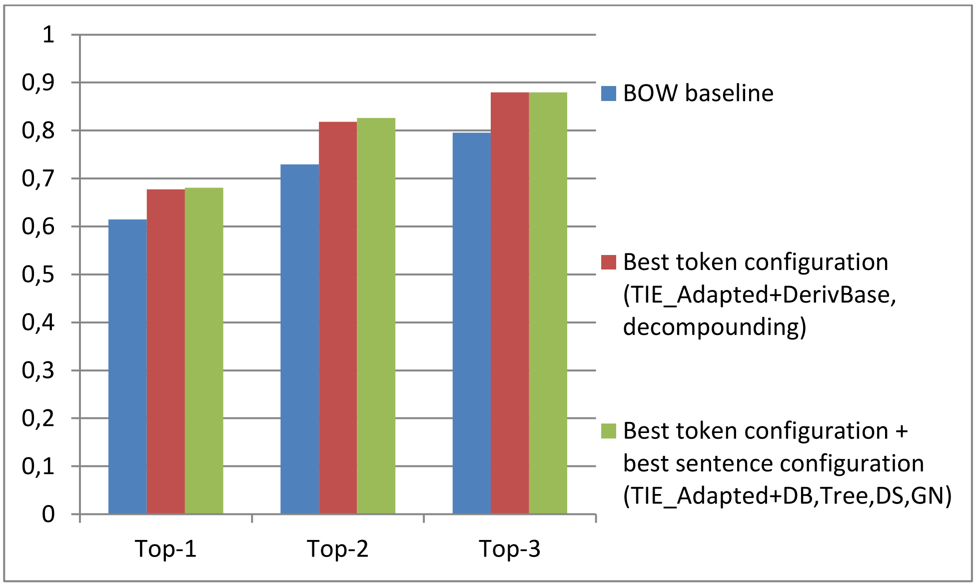
A sample subgraph (extracted from the generated one) is shown in the following (the boxes in read mention the resources used for creating the respective equivalence class / entailment edge):



While the resulting graph looks meaningful in most places, adding the two-token fragment graph to the token graph did not lead to better categorization results. Further investigation is needed to find out why.

7.3.4.4.6 Experiment 6: Combining the best token configuration with sentence fragments

In our last experiment, we combined the best-performing token graph with the sentence graph created with the best-performing TIE configuration. The following figure shows the results for top-1, top-2 and top-3 accuracy:



As the figure shows, there is a small accuracy improvement for top-1, top-2, and top-3 accuracy gained by adding sentence fragments to the graph. In the experiments we conducted, this configuration turned out to produce the best results on our dataset.

Integrating entailment graphs, we were able to beat the baseline by more than 10% (6,57 absolute percentage points). The improvements were due to the integration of DerivBase as a resource and to using decompounding to process compound words. The experiments revealed no / hardly any benefit of using directional entailment information or complex fragments.