D3.1.2: CORPUS CREATION & ANNOTATION MODULE (VER.2)

Grant Agreement number	ICT-248307
Project acronym	**PRESEMT**
Project title	**P**attern **RE**cognition-based **S**tatistically **E**nhanced **MT**
Funding Scheme	Small or medium-scale focused research project – STREP – CP-FP-INFSO
Deliverable title	**D3.1.2: Corpus creation & annotation module (ver.2)**
Version	v4
Responsible partner	MU
Dissemination level	Public
Due delivery date	30.6.2011 (+60 days)
Actual delivery date	30.8.2011

Project coordinator name & title	**Dr. George Tambouratzis**
Project coordinator organisation	**Institute for Language and Speech Processing / RC 'Athena'**
Tel	+30 210 6875411
Fax	+30 210 6854270
E-mail	giorg_t@ilsp.gr
Project website address	www.presemt.eu

Contents

1. EXECUTIVE SUMMARY 3

2. INTRODUCTION 4

3. WEB CRAWLING 5

3.1 WEB CRAWLING FOR PRESEMT 5

4. CHARACTER ENCODING DETECTION 6

4.1 BOILERPLATE REMOVAL 8
4.2 PAGE-LEVEL AND SITE-LEVEL METHODS 9
4.2.1 Segmentation 10
4.2.2 Classification 10

5. THE PRESEMT SOLUTION: JUSTEXT 12

5.1 DE-DUPLICATION 12
5.2 BRODER'S SHINGLING ALGORITHM 13
5.3 MAKING USE OF DUPLICATE N-GRAMS 14
5.4 FINDING DUPLICATE N-GRAMS WITH A SUFFIX ARRAY 15
5.5 REMOVING DUPLICATE TEXT BLOCKS 16
5.6 CZECH-SL CORPORA (MU) 18
5.7 GERMAN-SL CORPORA (GFAI) 19
5.8 GREEK-SL CORPORA (ILSP) 19
5.9 NORWEGIAN-SL CORPORA (NTNU) 19

6. REFERENCES 20

1. Executive summary

The present deliverable falls within Tasks T3.1: Mining the web for collecting resources and T3.2: corpus creation, analysis and annotation of WP3: Corpus extraction & processing algorithms. The work described concerns the compilation of huge monolingual corpora and of small bilingual ones, both of which will be utilised by the Main translation engine of the PRESEMT system. Details are also given on how these resources are processed and annotated.

The monolingual corpora are automatically created via crawling the web, a nowadays popular source of text data for multiple purposes. The automation of this procedure, however, calls for additional processing of the material collected, so as (a) to detect the original character encoding of each document and convert it to unicode, (b) identify and isolate non-textual data (called 'boilerplate') and (c) remove duplicate occurrences of content ('de-duplication'). The current deliverable discusses various methods for boilerplate removal and de-duplication and describes the methodology applied in the case of PRESEMT. The created monolingual corpora are utilised during the translation process for disambiguating translation equivalents. The following software used for processing the corpus data has been released as open source (under the new BSD license):

* chared [http://code.google.com/p/chared/]: character enconding detection tool
* jusText [http://code.google.com/p/justext/]: boilerplate removal tool
* onion [http://code.google.com/p/onion/]: a tool for detecting and removing duplicate content

The bilingual corpora are manually created, mainly from web content, and are of small size. It has been foreseen to compile 13 such corpora, which cover the project language pairs to be evaluated within PRESEMT. This resource will be utilised by the Phrase aligner module in order to create phrasing models and be accessed during the translation process for establishing the optimal sentence structure of a given SL text input for translation.

The deliverable has the following structure: Section 2 introduces the issue of using web as a text resource and the accompanying problems and Section 3 describes the method of web crawling for gathering large text resource over the web. Sections 4 and 5 present various methodologies for cleaning the web material collected (i.e. indentifying the 'good content') and removing the duplicate content respectively. Section 6 provides statistical data on the monolingual corpora collected so far for the purposes of the project, while Section 7 is dedicated to the bilingual corpora. References listed in Section 8 conclude the current deliverable.

2. Introduction

In the recent years, the web has become a popular source of textual data for linguistic research. The web provides an extremely large volume of texts in many languages. However, a number of problems have to be resolved in order to create collections (text corpora) which are appropriate for application in natural language processing. In this work package, we first describe how we crawl the web, and then two related problems are addressed: cleaning away 'boilerplate' and removing duplicate and near-duplicate content.

On most web pages, the main content is accompanied by so-called boilerplate content, such as navigation links, advertisements, headers and footers. Including the boilerplate in text corpora results in an undesirable over-representation of some common boilerplate words and phrases such as "*home*", "*search*", "*print*", etc. We have developed a novel heuristic-based approach to solve the problem.

Many texts on the web exist in multiple instances (e.g. mirrored websites, document revisions, quotations in discussion forums, etc). Duplicate data distorts corpus statistics and causes difficulties to users who search for linguistic data in web corpora. Many researchers have addressed the problem of identifying duplicate and near-duplicate web pages in the context of web search engines. However, the problem is typically limited to identifying almost identical documents. Pairs of web pages which contain significant amounts of both identical and different content are not considered near-duplicates by search engines. For text corpora, on the other hand, any duplicated data constitutes a problem.

Another resource required for PRESEMT is a small parallel corpus which can be used to model the structural mappings between the two languages.

In this report we first describe the methodology for collecting very large web corpora, including innovative techniques for boilerplate-removal and de-duplication (see Pomikálek (forthcoming)). Next, the process of compiling bilingual corpora is presented.

3. Web crawling

Building web corpora of over one billion words requires downloading millions of web pages. This is typically achieved by web crawling. Specialised software (a web crawler, also just crawler or a spider) is used to "crawl" over web pages by simulating clicking on the hypertext links. All the visited pages are downloaded. While the web crawling is straightforward in principle, a fast and robust web crawler capable of handling terabyte downloads is not easy to implement. Though many open source web crawlers exist, the production-ready ones are fairly rare. A popular choice in the web-as-corpus community is the Heritrix crawler developed by Internet Archive[1].

For performing a web crawl, a list of starting URLs is required. These can be obtained from search engines using the techniques described above. One of the essential requirements on web crawling for corpora is that mostly the web pages containing the texts in the language of interest are downloaded. The most straightforward way to achieve this is by making the crawler stick to the first level domains of the countries where the language is prevalently used (e.g. *.de* and *.at* for German). In general, it is a good idea to employ further filtering based on content-type (e.g. only HTML pages), size (very large files usually do not contain useful data), etc.

Once the data is downloaded, it has to be converted to plain text and cleaned of boilerplate content. A language filter (e.g. based on frequencies of character triples) should be applied to weed out pages not in the language of interest. Duplicate and near-duplicate data need to be removed. The web-crawled corpora have been pioneered by the WaCky community[2] who compiled large web corpora for several languages (e.g. German, Italian, English, French), each approaching the size of 2 billion words.

3.1 Web crawling for PRESEMT

Crawling for PRESEMT used Heritrix, with the settings as explored and set by Marco Baroni and colleagues at the University of Bologna for the DeWaC, ItWaC, FrWaC and UKWaC corpora. National domains (e.g. .gr for Greek) were used. Some web pages (for examples those with URLs ending in .jpg, .avi) were not downloaded on the basis of regular expression filtering of the URLs. Content-type filtering was also applied, based on the file header: an HTTP header is retrieved from the server first and the whole file is downloaded only if the content-type is "*text/html*". Content-length filtering based on the HTTP header is also applied. Only files between 5kB and 2MB are downloaded.

1 http://crawler.archive.org/
2 http://wacky.sslmit.unibo.it/doku.php?id=start

4. Character encoding detection

Text documents and Web pages in particular use various character encodings, such as iso-8859-1, windows-1251, utf-8, etc. In order to achieve a unified representation of non-ascii characters in a corpus, the character encoding of each source document has to be detected and converted to a single selected (unicode) encoding, such as utf-8. Converting a text from a character encoding X into a character encoding Y is straightforward. Finding the correct value of X is a challenging task though.

Web pages usually contain the information about the used character encoding in the meta-tags. However, this information is not always available and not always correct, and in general cannot be relied upon. Nevertheless, the character encoding can usually be reliably guessed from the textual contents of a Web page.

The problem of character encoding detection can be tackled at two different levels. First, no assumption is made about the language of the input text. In this case both the character encoding and the language of the input are guessed at the same time. Second, the language of the input is known and only the character encoding is detected. Existing systems which fall into the former category include for instance TextCat[3]. A notable representative of the latter category is Enca[4].

The disadvantage of the existing systems is that they support only a limited number of languages and encodings. The system we designed, on the other hand, covers a wide range of languages and the frequently used encodings for each language. In particular, all the languages relevant for the PRESEMT project (Czech, English, German, Greek, Italian, Norwegian) are supported.

Our detection system assumes that the language of the input is known. Note that when building monolingual corpora this is always guaranteed.[5] Also, by narrowing the problem down to a single language, higher accuracy can be achieved.

The system works as follows. First, a model is created for each supported language. This requires two kinds of input:

* The list of character encodings used for the given language
* A sample text in the language and in a known character encoding

We convert the sample text to all encodings from the list, creating N different files (where N is the number of encodings). Then, for each of the files we build a frequency list of all consequent triples of bytes. The N frequency lists (vectors) are then compared and the items (triples) with the same frequency in all the N vectors are discarded.

When detecting a character encoding of a document, its frequency vector is built and a scalar product is computed with each of the N model vectors. The character encoding associated with the model vector which produces the highest scalar product is returned as the encoding detected for the input document.

The advantage of building model vectors from the same data (only converted to N different encodings) is that the differences among the model vectors are only where the character encodings differ. If for instance all the N encodings are ascii compatible, then any triple of bytes containing only ascii characters will have the same frequency in all model vectors. Thus, this triple would add the same value to the scalar product with a tested document (vector) for all models. As such it cannot affect the order of the final scalar products (and the result of the classification) and thus it can be safely removed from all the model vectors. This both reduces the size of the vectors and speeds up the classification.

[3] http://www.let.rug.nl/vannoord/TextCat/
[4] http://gitorious.org/enca
[5] In fact, in a web corpus processing pipeline, character encoding detection has to be performed before language detection. Thus, the assumption that the input of the character encoding algorithm is in the language in question is not always valid. However, this is not a problem, since all texts in other languages will be refused by the language filter in the next step and it therefore doesn't matter whether their character encoding has been detected correctly or not.

Alternative approaches which build the models on N different texts in N different encodings prevent such pruning. Apart from that, similarity of the tested document to the texts used for training the models (regardless of their encoding) may bias the results of the encoding detection. This bias would be the more severe the more ascii characters (or more generally, the more characters with the same representation in all the encodings in question) are used in the training texts (e.g. very severe for English).

The reason for using frequencies of tuples (triples in our case) of characters rather than single characters is that some bytes represent multiple different commonly used characters in different encodings. Such bytes may cause errors in the encoding detection. This is best illustrated by an example. The byte a9 represents the character © (copyright symbol) in windows-1250 and the character Š (capital s with caron) in iso-8859-2. Both encodings are used for the Czech language. The Š character is much more frequent in Czech texts than the copyright symbol. Thus, a windows-1250 text containing a copyright symbol may be easily misclassified as iso-8859-2 since the a9 byte will have a higher frequency in the iso-8859-2 model. Since the two encodings are very similar, this kind of classification error is fairly likely, especially for short texts.

Building models on tuples of characters rather than on single characters helps to overcome this problem. The copyright symbol will be typically followed by a space whereas the Š character will be typically followed by another letter in a Czech text. Therefore, the two bytes a920 (© followed by a space in windows-1250) will certainly have a higher frequency in a windows-1250 model than in a iso-8859-2 model (for Czech). In our experiments, using triples of characters resulted in a higher accuracy of the encoding detection than using only pairs of characters.

In order to create models for our system, we collected a set of about 1000 Web pages for each language. This was done using the Corpus Factory tools as described in (Kilgarriff, 2010). We identified the encoding of each page by using the information in the meta-tags. Though we are well aware that this information is not always correct we believe the errors are so rare that they cause only an insignificant bias in the built models. We simply discarded all pages for which we haven't been able to determine the character encoding from the meta-tags.

First, for each language, we computed the relative frequencies of the encodings used in the Web pages and accepted those with a relative frequency above 0.5% as the encodings commonly used for the language. We then created a model vector for each of these encodings by converting the Web pages to the encoding and building the frequency vector as described above.

To evaluate the system, we performed a 5-fold cross-validation on the training data. The following table contains the results for the "PRESEMT languages".

	Czech		English		German		Greek		Italian		Norwegian	
	freq	accu-racy	freq	accu-racy	freq	accu-racy	freq	accu-racy	freq	accu-racy	freq	accu-racy
utf-8	60.2%	100.0%	56.9%	95.8%	54.6%	100.0%	68.5%	100.0%	54.2%	100.0%	63.0%	100.0%
windows-1250	32.2%	100.0%	0.3%	n/a	0.1%	n/a	0.2%	n/a	0.0%	n/a	0.1%	n/a
windows-1252	0.4%	n/a	9.4%	97.5%	6.5%	97.3%	3.1%	75.8%	7.1%	95.7%	7.0%	97.4%
windows-1253	0.0%	n/a	0.0%	n/a	0.0%	n/a	14.3%	99.3%	0.0%	n/a	0.0%	n/a
iso-8859-1	1.0%	89.5%	32.8%	90.9%	37.1%	85.8%	1.7%	71.2%	37.9%	85.1%	29.3%	88.2%
iso-8859-2	6.0%	99.6%	0.0%	n/a	0.1%	n/a	0.0%	n/a	0.1%	n/a	0.1%	n/a
iso-8859-7	0.0%	n/a	0.0%	n/a	0.0%	n/a	12.0%	97.2%	0.0%	n/a	0.0%	n/a
iso-8859-15	0.0%	n/a	0.0%	n/a	1.2%	85.6%	0.0%	n/a	0.0%	n/a	0.4%	n/a
training docs		801		668		773		879		771		740
w. avg accuracy		99.2%		93.5%		93.7%		97.9%		93.3%		95.7%

Note that the results may be harmed by incorrect "annotation" of the data. We manually inspected some of the misclassified pages and often found out that the algorithm detected the encoding of the page correctly, but the character encoding specified in the meta-tags was wrong. We therefore assume that the actual accuracy of our system is higher than indicated by the numbers above.

The average accuracy is weighted by the frequency of occurrence of the encodings. The rationale of the weighting is that it's more important that the algorithm correctly identifies the frequently used encodings than the rarely used ones.

4.1 Boilerplate removal

'Boilerplate' is usually defined rather vaguely as non-informative parts outside of the main content of a web page, typically machine-generated and repeated across the web pages of the same website. While some elements such as navigation menus or advertisements are easily recognised as boilerplate, for some other elements it may be difficult to decide whether they are boilerplate or not in the sense of the previous definition.

Imagine a web page from a news site which, apart from a full text of one article, contains an abstract of another article with a link to the full text. Does this abstract qualify as boilerplate? Is it informative? Is it the main content? It turns out that the notion of boilerplate is not easy to define with a single sentence. In the context of cleaning boilerplate from web pages, the term typically denotes any elements which constitute noise for the application of the web data. The exact definition is therefore application-specific.

Two fairly detailed specifications of boilerplate exist, in the form of boilerplate annotation guidelines: the guidelines for the annotators of the CleanEval competition gold standard[6] and the guidelines for using the KrdWrd annotation tool[7].

6 http://cleaneval.sigwac.org.uk/annotation_guidelines.html
7 https://krdwrd.org/manual/html/node6.html

The CleanEval guidelines instruct to remove boilerplate types such as:

* Navigation
* Lists of links
* Copyright notices
* Template materials (e.g. headers and footers)
* Advertisements
* Web-spam, such as automated postings by spammers
* Forms
* Duplicate material, such as quotes of the previous posts in a discussion forum.

The KrdWrd guidelines are more recent. They arose from the CleanEval guidelines, but they go one step further. Only the text made up of complete sentences is taken as the main content here. Elements such as lists and enumerations are classified as boilerplate unless they contain full sentences.

It is noteworthy that this is a significant difference from CleanEval where the inclusion of list items in the cleaned data is specifically required. Clearly, the KrdWrd aims to define the boilerplate in a way so that the remaining content constitutes good corpus data. The texts consisting of complete sentences are of the highest value for text corpora. Data such as lists or tables, on the other hand, are rather counterproductive for linguistic research. The CleanEval's specification of boilerplate is more conservative and may, for instance, be appropriate for information retrieval, but less so for creating web corpora. As this work is done in the context of web corpora, the KrdWrd's perspective on boilerplate is more relevant. Still, as the two views overlap to a high extent, the datasets originating from both KrdWrd and CleanEval are used for our experiments. The results on the KrdWrd's Canola dataset are considered more important though.

It should also be noted that CleanEval includes web-spam and duplicate material in the definition of boilerplate. Little effort has been invested into detecting this kind of content by CleanEval contestants though. Only one system (Victor) tries to identify inter-page duplicate material. None of the participants has made an attempt to detect web-spam. This is not surprising as while most boilerplate types can be identified based on structural features (surrounding HTML mark-up) and shallow text features (such as the number of tokens in a text block), a fundamentally different approach is required for detecting both web-spam and duplicate content. Recognising web-spam is a challenging task which is out of the scope of this work package; finding duplicated and near-duplicated texts is addressed in the next subsections.

4.2 Page-level and site-level methods

The approaches to automated boilerplate removal can be divided into two main groups: page-level and site-level. The page-level algorithms process each web page individually. Within the site-level methods, multiple pages from the same website are processed at once.

Site-level methods take advantage of the fact that web pages within the same website typically use the same or similar template. Consequently, the boilerplate sections show common characteristics across the website and can be identified by exploiting similarities among web pages.

Nevertheless, site-level methods introduce practical problems as there may not be enough pages from the same website available, either because they do not exist (the website is very small) or they have not been downloaded for a given data set. The grouping of the web pages per website before processing is a more complicated task than handling each page separately. It may be especially problematic, should the processing be done while the data is still being collected.

Page-level boilerplate removal methods take a single web page as input. This makes them more flexible and easier to use than the site-level methods.

Most boilerplate removal algorithms operate in two steps: segmentation and classification.

4.2.1 Segmentation

In the segmentation step, the input HTML page is separated into semantically coherent blocks. In the subsequent steps, each block is classified as main content (clean text) or boilerplate. The blocks should be sufficiently homogeneous in terms of boilerplate content, i.e. each block should ideally contain either boilerplate or clean text, but not a mixture of both. Different segmentation algorithms are used across boilerplate removal methods as well as different levels of granularity.

The segmentation granularity typically constitutes a trade-off between the homogeneity of blocks and the ease of classification. If the segmentation is too fine-grained, individual blocks may be very small and may not carry enough information to be classified reliably. If, on the other hand, the segmentation is too coarse-grained, it is more likely that heterogeneous blocks will be formed.

Simple approaches to segmentation (preferred by most algorithms) include treating each DOM[8] text node as a single block or splitting the page at specified HTML tags.

4.2.2 Classification

The classification step decides for each block whether it is main content (clean text) or boilerplate. Depending on the application of the cleaned data, more than two classes may be supported. Some CleanEval contestants, for instance, attempt to sub-classify the clean blocks as paragraphs, headers or list items.

The commonly used classification methods include both heuristics and supervised machine learning techniques, such as Conditional Random Fields (CRFs), Logistic regression classifiers, Support Vector Machines (SVMs), decision trees, Naive Bayes. The lists of features used by different boilerplate cleaning methods for making the classification decisions (heuristics or machine learning-based) overlap to a high extent. The features can be divided into three main groups:

* Structural features based on HTML mark-up: The most frequently used ones are tag density (the number of HTML tags divided by the number of words/tokens), number of links and link density (the proportion of tokens inside <a> tags), occurrence of certain tags and their count, parent tag type, etc.
* Textual features capture the properties of the text contained in the block. Typical textual features include number of words or characters, number of sentences, average sentence length, proportion of function words, frequency of certain keywords, proportion of capital letters or punctuation marks, etc.
* Visual features are derived from a visual form of a web page as rendered by web browsers. Most researchers make use of visual features such as size and shape of a block, distance from the centre of the page or from its borders, font size and weight, etc.

It is noteworthy that extracting visual features may be both computationally expensive and problematic as (i) the web page must be rendered and (ii) external resources such as images or style sheets may be required. For these reasons, most researchers avoid using visual features. Besides, there is no evidence that systems employing visual features would perform better than systems which do not.

8 Document Object Model (DOM): http://www.w3.org/DOM/

Until recently, the Body Text Extraction (BTE) algorithm (Finn et al., 2001) has been the leading player. It removes boilerplate using a simple heuristic based on the density of HTML tags. The idea is that the main body contains only little formatting and is therefore sparse in terms of HTML tags. The navigation links, advertisements and alike, on the other hand, contain a lot of tags, and tend to be at the beginning and the end of the page.

The BTE algorithm views a page as having three parts, beginning (boilerplate), middle (content) and end (boilerplate). The task is then to detect the junctions between these three elements, which is done by finding the points at which the ratio of tags to non-tags changes most steeply.

5. The PRESEMT solution: jusText

In early experiments, BTE was used as a pre-processing stage. Boilerplate was often found in the data identified as duplicate content. This is not surprising as boilerplate is often repeated in web pages within the same website. These experiments revealed a lot of boilerplate that BTE has failed to remove. In the search for a more effective algorithm, jusText has been created.

The algorithm uses a simple method of segmentation. The contents of some HTML tags are (by default) visually formatted as blocks by web browsers. The idea is to form textual blocks by splitting the HTML page on these tags with the hope that they will be sufficiently homogeneous in terms of good and boilerplate content. The evaluation section demonstrates that blocks of this kind can contain both, but it is too rare to constitute a problem.

Several observations can then be made about the blocks created in this way:

1. Short blocks which contain a link are almost always boilerplate.
2. Any blocks which contain many links are almost always boilerplate.
3. Long blocks which contain grammatical text are almost always good, whereas all other long blocks are almost always boilerplate.
4. Both good (main content) and boilerplate blocks tend to create clusters, i.e. a boilerplate block is usually surrounded by other boilerplate blocks and vice versa.

Deciding whether a text is grammatical or not may be tricky, but a simple heuristic can be used based on the volume of function words (stop words). While a grammatical text will typically contain a certain proportion of function words, few function words will be present in boilerplate content such as lists and enumerations.

The key idea of the algorithm is that long blocks and some short blocks can be classified with very high confidence. All the other short blocks can then be classified by looking at the surrounding blocks.

jusText has been extensively evaluated, and found to outperform all other known algorithms. For a detailed account of the algorithm and its performance see Pomikálek (forthcoming).

5.1 De-duplication

Many texts are present on the web in multiple instances. Common types of duplicates and near-duplicates include mirrored websites, multiple presentation styles of the same web page (e.g. one for viewing in a web browser and one for printing), document revisions, similar or identical news articles at different news sites, quotations of previous posts in online discussion forums, etc. Duplicate content is problematic for many applications of web data. For instance, imagine a web search engine which indexes all web pages without worrying about duplicates. It could then easily happen that the first page of web search results for a given query would only contain hits in identical documents. Such a search engine would not be very useful. The situation is similar with web corpora. Users might get many duplicate concordance lines when searching in a corpus containing duplicate texts. Moreover, duplicate content may bias results derived from statistical processing of corpus data by artificially inflating frequencies of some words and expressions. Identifying and removing duplicate and near-duplicate texts is therefore essential for using the web data in text corpora.

When referring to duplicates, it is important to distinguish between naturally and artificially repeated texts. It is perfectly normal that some language phenomena, such as words, expressions and even full sentences are used repeatedly and independently. Such recurrences are natural and are not considered duplicates (at least in the scope of this work). On the other hand, there are texts which are taken copy-and-paste from other sources (and possibly slightly modified) rather than created independently. Such a repetition is no longer representative of normal language use. It is extremely unlikely that two different language users would independently produce two completely identical paragraphs or even full documents. When referring to duplicates and near-duplicates, these copy-and-paste recurrences are meant. Some researchers also use the term co-derivative texts.

Identifying duplicate and near-duplicate texts is fairly easy for small data collections. Here, we can simply compare each pair of documents (or other units, such as paragraphs). Deciding whether two documents are identical is trivial. It is also not difficult to decide whether two documents are similar (near-duplicate). Many metrics are available for measuring the similarity of documents, such as Levenshtein edit distance (Levenshtein, 1966), Broder's resemblance (Broder, 1997) or cosine similarity of document vector space models. For small data sets, the similarity of each pair of documents can be computed and the pairs with a similarity value above certain threshold reported as near-duplicates.

However, the problem gets difficult for large data collections where comparing each pair of documents would be prohibitively expensive. Today, web corpora comprising billions of words are fairly common. Such corpora are created from millions of web pages and would require trillions of pairwise document comparisons to be performed.

For finding exact duplicates, document checksums (fingerprints) can be used. The checksum is an output of a hash function, such as MD5[9]. Two identical documents will always have the same checksum whereas there is typically a high probability for the checksums of two different documents to be different. The advantage of using checksums is that they are far shorter than the original documents and it is far cheaper to compare them. They can be easily stored in memory. Duplicated documents can then be identified and removed with a single pass over the data set by probing the checksum of each document against the checksums of previously seen documents.

The situation is more complicated with near-duplicates. With standard hash functions, a change in a single byte of the input changes the output completely. It is therefore not possible to assume that the checksums of similar documents will be similar or even identical. The problem of finding near-duplicated documents has been addressed by many researchers, mostly in the context of web search engines, by using special fingerprinting techniques.

5.2 Broder's Shingling algorithm

Broder (2000) uses a special fingerprinting scheme designed specifically for detecting near-duplicate documents. Each document is represented as a set of shingles (or n-grams), which are sequences of any n consequent words. For example, for n = 6, the text "what shall we do with a drunken sailor" contains three shingles:

(what; shall; we; do; with; a)

(shall; we; do; with; a; drunken)

(we; do; with; a; drunken; sailor)

9 http://www.w3.org/TR/1998/REC-DSig-label/MD5-1_0

Each document is then represented by a small random sample of shingles, thus economising on space and time. A document similarity measure is then computed for each document pair according to the number of shared shingles in their corresponding samples. The risk of misclassification can be kept arbitrarily low by changing the number of shingles used to represent a document.

Most of the research on identifying duplicate and near-duplicate content in large data collections has been done in the context of web search engines. Here, the goal is to identify identical and almost identical pairs of web pages, i.e. those which only differ in small details, such as a timestamp or a visitor count. Broder's and other methods are effective for this task and they scale up very well since minor differences between documents can be detected by using very small document fingerprints.

However our analysis of web pages shows that almost-identical pages represent only a small percentage of web documents which have a significant amount of content in common. Pairs of web pages with an intermediate level of similarity (say 50-80% of shared content) are fairly frequent. While these pairs are probably of little interest for web search engines, they definitely cannot be ignored when creating web corpora as they contain a lot of undesirable duplicate texts.

Applying the methods developed by search engine companies to removing this duplicate text leads either to a significant loss in accuracy or to increasing the size of the fingerprints and reducing the scalability of the method. It is easy to see why this happens. The algorithms select some small samples of the original documents and estimate the similarities of documents by matching the samples. For almost identical documents, a few samples are sufficient since it is likely that the identical parts of the documents will be sampled and matched. However, as the documents become less similar, the risk of sampling different parts increases. This reduces the reliability of the method unless more samples are used.

The document sampling principle is common for most near-duplicate detection fingerprinting schemes and the described problem can be expected to apply in all of these schemes.

5.3 Making use of duplicate n-grams

Bernstein and Zobel (2004) suggested that most of the near-duplicate detection algorithms select inappropriate document samples for fingerprints. They pointed out that those samples which occur only once in the whole data set are not of any use for identifying near-duplicate documents. Including these unique samples in document fingerprints only increases their size and/or harms the accuracy.

Bernstein and Zobel (2004) thus proposed an iterative algorithm (SPEX) for finding all shingles (n-grams) which occur at least twice in the data set. For reasonably long shingles, the duplicated ones represent only a small percentage in most real world document collections. They demonstrated this on a 476MB collection of newswire articles from the Los Angeles Times. Out of 65,900,076 distinct shingles of length eight (8-grams) only 907,981 (less than 1.4%) occurred two or more times.

Since the duplicated shingles are fairly rare, it is possible to use them for creating reasonably sized document fingerprints. Such fingerprints can be used for computing the values of document similarity functions (such as resemblance) exactly rather than estimating them. Smaller fingerprints can be created by sub-sampling. Though in this case the similarities can no longer be computed exactly, it should still be possible to achieve more accurate estimates than from equally sized fingerprints based on virtually random sampling methods.

The SPEX algorithm computes the list of duplicated shingles in an iterative way. The core idea is as follows. A shingle S of length n can only be duplicated if both of its sub-shingles S1 and S2 of length n1 are duplicated. If S1 or S2 is unique then it is clear that S must be unique, too. For example, the bigram nascent effort can only be duplicated if each of the words nascent and effort occurs at least twice in the collection. SPEX starts from a list of duplicated unigrams (single words). Since the number of distinct words is usually small even for large data collections, it is possible to hold a counter for each word in a hash table. Thus, finding the duplicated words is straightforward.

However, with the increasing length of shingles, the number of distinct ones increases quickly. The number of distinct long shingles (such as 10-grams) is usually close to the size (number of words) of the whole data set. For large collections, this may easily exceed a normal capacity of RAM. Therefore, SPEX prunes the shingles known to be unique based on the results of the previous iteration. For each n-gram, both (n1)-grams are extracted and if one of them is not found among the duplicated (n1)-grams (identified in the previous iteration), the specific n-gram is ignored. This reduces memory requirements to some extent.

Rychlý and Pomikálek (2008) show that the memory requirements of SPEX are still quite high, especially throughout the third iteration. The problem is that the number of unique trigrams is typically high whereas much fewer unique bigrams exist since many get repeated as a result of a normal language use. Thus, when the list of duplicated trigrams is built, a counter for many unique trigrams is needed which are not pruned since they contain naturally duplicated bigrams.

5.4 Finding duplicate n-grams with a suffix array

While it is clear that duplicate n-grams are very useful for finding near-duplicate data, the high memory requirements of the SPEX algorithm are problematic. We have therefore developed an alternative algorithm for finding duplicate n-grams with a constant amount of memory.

An obvious way to get a list of duplicate n-grams is to generate all n-grams and sort them. External sorting can be used for large collections. The input is split into sufficiently small chunks which can be sorted in memory one by one. Each sorted chunk is dumped to a hard disk. As a last step, the sorted chunks are merged to form the final sorted list. The problem of this approach is that a large amount of data has to be processed. Note that the size of the list of all n-grams is n times the size of the whole corpus.

We have proposed two optimisations to the process. First, rather than sorting n-grams, we build a suffix array using a suffix sorting algorithm. This can be done with much less memory than sorting n-grams. Thus, larger chunks can be processed and the number of chunks is reduced. Generating the list of sorted n-grams from a suffix array is straightforward.

Second, we compress the temporary files (sorted chunks) to reduce the required disk space. We start by mapping words to numeric IDs in order to work with integers rather than with character strings. The IDs are assigned in an increasing order starting from 0. Each newly encountered word is mapped to the next unassigned ID. (The first word in the corpus is mapped to 0; the second word is mapped to 1 unless it is the same as the first word, etc.). Frequent words are encountered early and thus they get small IDs.

Once the words are mapped to IDs we can think of the whole corpus as a long sequence of integers. We split it to smaller sub-sequences which can be processed in memory. For each sub-sequence, a suffix array is built using an efficient suffix sorting algorithm; a list of sorted n-grams (represented as sequences of n integers) is generated from the suffix array and stored in a temporary file.

When saving the sorted n-grams, Elias codes (Witten et al., 1999) are used to represent the word IDs. This compresses the data since frequent words are mapped to small integers and these are represented by short bit-strings in Elias codes. Additional compression is achieved by omitting the longest common prefix with the previous n-gram.

In order to build a suffix array, 12 bytes per word are required (4 bytes to represent the word ID and 8 bytes in auxiliary data structures). Thus, by using 1GB of memory, we can process chunks of roughly 89.5 million words. When sorting n-grams directly, we would need approximately 6n bytes per n-gram (assuming 5 bytes per word plus word separators) or 4n bytes per n-gram by using word enumeration (assuming 32-bit integers). For 10-grams, the latter (more favourable) approach still requires 40/12 or 3.33 times more memory than suffix sorting, i.e. 3.33GB of RAM would be needed for processing 89.5 million words.

According to our experimental results, the size of the compressed temporary files is about 3 times the size of the corpus. Direct external sort of n-grams would require n times the size of the corpus.

We have successfully applied the suffix sorting based algorithm to extract duplicate 10-grams from a corpus of more than 9 billion words. It took 18.5 hours on a single 2.4 GHz machine and required no more than 1.5GB RAM.

Since the time complexity of the proposed algorithm is close to linear and the memory requirements are constant, the algorithm should be practical even for processing much larger corpora.

5.5 Removing duplicate text blocks

Having identified the duplicate content, the next step is to remove it. Removing near-duplicate data on document level is often problematic, especially with respect to documents with an intermediate level of duplicate content. Consider a largish document with 50% of duplicate content and 50% of unique content. By keeping the document we allow duplicate texts in the corpus. By discarding the document we lose good texts. In an ideal case we would want to discard only the duplicate part and keep the unique part.

An obvious way to do so is to work with smaller units than documents, such as paragraphs or even sentences. This is fairly straightforward. However, it introduces a problem of fragmentation.

A full document is typically a sufficiently independent unit in the sense that it can be fully understood by a reader without requiring any external information (except for eventual background knowledge in the relevant area). The meaning of a sentence or a short text block taken out of context, on the other hand, can often be unclear. It may be difficult to recognise the senses of the words in such a text.

When applying a de-duplication algorithm on units smaller than full documents, it can happen that some documents are broken into small fragments (stubs) for which the context information is not available since the surrounding text blocks have been removed as duplicates. Depending on the intended use of the corpus it may be desirable to remove the stubs. The problem of document fragmentation has already been discussed in relation to boilerplate removal, with the context-sensitive classification of the jusText algorithm reducing the fragmentation to a high extent. This part of the algorithm is reused for the same purpose in the de-duplication process.

The PRESEMT monolingual corpora: statistics

The following table provides detailed statistical information on the monolingual corpora collected so far and their size after the process of boilerplate removal and de-duplication.

Table 1: Statistical data on PRESEMT monolingual corpora (1st year of the project)

Language	English	Italian	German	Greek	Czech	Norwegian
Corpus name	**enTenTen / BiWeC**	**itTenTen**	**deTenTen / BigDeWaC**	**grTenTen**	**czTenTen**	**noTenTen**
final size (*in tokens*)	3,658,726,327	3,076,812,674	2,874,779,294	1,073,266,453	2,028,422,159	2,267,156,701
downloaded volume (*gzipped*)	129 GB	272 GB	291 GB	157 GB	418 GB	364 GB
downloaded volume uncompressed645 GB	645 GB	1360 GB	1455 GB	785 GB	2090 GB	1820 GB
downloaded (*unique*) URLs	13,638,928	33,459,999	43,160,992			
original main content (*words*)	6,805,296,135	7,740,199,568				
original boilerplate content (*words*)	7,843,420,305	14,814,859,910				
after removing exact duplicates						
docs	3,357,252	5,335,839	8,237,310	3,357,894	5,982,830	
tokens	4,765,119,530	5,017,409,779	4,880,335,291	1,952,309,530	3,325,287,880	
duplicate 10-grams	212,864,389	295,369,351	331,258,395	131,047,973		
after removing duplicate text blocks						
docs	2,838,738	4,020,968	5,752,857	2,208,243	3,785,229	6,015,349
tokens	3,658,726,327	3,076,812,674	2,874,779,294	1,073,266,453	2,028,422,159	2,267,156,701
duplicate 10-grams	6,757,185	12,230,555	7,196,387			
	words = whitespace separated character strings (wc -w)					
	tokens = unitok.py (universal tokeniser) output					

Bilingual corpora collection

Whereas the core monolingual corpora should be huge, comprising billions of words, the bilingual corpora required for each language pair need only comprise a few thousand words. The bilingual corpora serve a two-fold purpose: they are utilised by the Phrase aligner module, so as to extract phrasing models per language pair, and the Structure selection module accesses them for establishing the optimal sentence structure of a given SL text input for translation.

The bilingual corpora created by the consortium members have been compiled according to the following criteria:

1. The corpus size should amount to 200 sentences of variant length.
2. It should be preferably collected from a bilingual or multilingual website.
3. After the corpus has been collected, the source language side should be modified so as to bring it as "close" as possible to the target language side (e.g. avoid metaphors or elliptical constructions, smooth out divergences between the two languages etc.). A copy of the original corpus version should be kept, as this can be used later for additional experiments.
4. The SL side of the modified corpus should be tagged and lemmatised. The output should be corrected.
5. The TL side of the modified corpus should be tagged, lemmatised and parsed. The output should be corrected.
6. The corrected outputs of steps 5 and 6 will be fed to the Phrase aligner module.

The following table illustrates the bilingual corpora collected so far per language pair, whereas the subsequent subsections describe data sources:

Table 2: PRESEMT monolingual corpora (1st year of the project)

		Target Language		
		English	German	Italian
Source Language	Czech	1	1	
	English			
	German	1		
	Greek	1	1	
	Norwegian	1	1	

5.6 Czech-SL corpora (MU)

The Czech-to-English corpus was prepared 1997 by the Department of English and American Studies, Faculty of Arts, Masaryk University in cooperation with the NLP Centre, Faculty of Informatics, Masaryk University. It is a small parallel corpus which allows one to work with entire texts in translation analysis rather then short extracts. At the same time it aims at acquiring experience that could be used in creating a larger parallel corpus of English and Czech in the future. Its original name is KACENKA (in Czech: Korpus anglicko- český – elektronicky nastroj Katedry anglistiky – acronym also means 'small duck'). It contains 3,297,283 words from which the respective number of sentences was selected.

The Czech-to-German corpus was prepared by the Department of German Language and Literature, Pedagogical Faculty, Masaryk University in cooperation with the NLP Centre, Faculty of Informatics. It was made accessible in 2002 on the server of the NLP Centre FI MU. From 2003 it contains 184 texts (newspapers, fiction) and 3,637,825 tokens in the Czech part from which the respective number of sentences was selected.

For both of these corpora, the alignment at paragraph and sentence level was performed manually. The Czech text was annotated by the desam tagger, while the English and German texts were processed with TreeTagger[10]. Their use is limited to research and teaching purposes only.

Both Czech-source corpora are in the following files: ***.sent** – sentences (one sentence per line) and ***.vert** – vertical (one word per line).

5.7 German-SL corpora (GFAI)

The German-to-English bilingual corpus comprises 164 sentence pairs of 6,767 tokens in total. It originates from a multilingual EU website (http://europa.eu/abc/12lessons/index_en.htm), referring to various EU-related issues. The German text has been tagged and lemmatised with TreeTagger and RFTagger[11], while the TL side (English) has been processed with TreeTagger, yielding tag, lemma and phrase annotations.

5.8 Greek-SL corpora (ILSP)

The Greek-to-English bilingual corpus was collected manually from a multilingual website of EU (http://europa.eu/abc/history/index_en.htm) and its content refers to the history of the European Union. It numbers 200 sentence pairs and contains 6,659 tokens. The Greek side of the corpus has been tagged and lemmatised with the ILSP FBT Tagger & Lemmatiser (Papageorgiou et al., 2000), while the English side has been processed with TreeTagger, yielding tag, lemma and phrase annotations.

The Greek-to-German bilingual corpus was also extracted manually from a multilingual website of EU (http://www.eea.europa.eu/), its content spanning various topics such as air pollution, environment and health, household consumption etc. It numbers 200 sentence pairs and contains 8,705 tokens. The ILSP FBT Tagger & Lemmatiser has been used for tagging and lemmatising the Greek text, while the German one has been processed with TreeTagger and RFTagger, yielding tag, lemma and phrase annotations.

5.9 Norwegian-SL corpora (NTNU)

As the Greek and German corpora were derived from the multilingual website about the history of the EU, NTNU decided to go with the same material. However, as Norway is not an EU member, there is no Norwegian version of this site. Therefore the relevant sentences were translated from the Danish version (which is closest) to Norwegian. Currently NTNU is in the progress of "anglification" / "germanification" (i.e. step 4 above), in order to bring the Norwegian translations closer to the target language (English / German).

[10] http://www.ims.uni-stuttgart.de/projekte/corplex/TreeTagger/DecisionTreeTagger.html
[11] http://www.ims.uni-stuttgart.de/projekte/corplex/RFTagger/

6. References

Bernstein Y. and Zobel J., 2004. A scalable system for identifying co-derivative documents. Proceedings of String Processing and Information Retrieval Symposium, pages 55-67.

Broder A. Z., 1997. "On the Resemblance and Containment of Documents". Proceedings of Compression and Complexity of Sequences 1997, pp. 21-27, IEEE Computer Society.

Broder A. Z., 2000. Identifying and filtering near-duplicate documents. Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching, pages 110.

Finn A., Kushmerick N. and Smyth B., 2001. Fact or fiction: Content classification for digital libraries. In DELOS Workshop: Personalisation and Recommender Systems in Digital Libraries.

A. Kilgarriff, S. Reddy, J. Pomikálek, and A. PVS. A corpus factory for many languages. Proc. LREC, Malta, 2010.

Levenshtein, V.I., 1966. Binary codes capable of correcting deletions, insertions, and reversals. Soviet Physics Doklady, 10, 8, pp. 707-710.

Papageorgiou H., Prokopidis P., Giouli V. and Piperidis S., 2000.A Unified POS Tagging Architecture and its Application for Greek. In Proceedings of the 2nd Language Resources and Evaluation Conference, pp. 1455-1462, Athens, Greece.

Pomikálek J. (forthcoming). Removing Boilerplate and Duplicate Content from Web Corpora (PhD thesis). Masaryk University, Brno, Czech Republic.

Pomikálek J. and Rychlý P., 2008. Detecting co-derivative documents in large text collections. In Proceedings of LREC, 2008. Marrakech, Morrocco.

Witten I.H., Moffat A. and Bell T.C., 1999. Managing Gigabytes: Compressing and Indexing Documents and Images. Morgan Kaufmann.