TUD Palladian Overview

David Urbansky, Klemens Muthmann, Philipp Katz, Sandro Reichert
TU Dresden, Department of Systems Engineering, Chair Computer Networks, IIR Group, Germany

October 24, 2011

Contents

Chapter 1

Introduction

Internet Information Retrieval (IIR) is a research domain in computer science concerned with the retrieval, extraction, classification, and presentation of information from the Internet. The toolkit provides functionality which is often needed to perform IIR tasks such as crawling, classification, and extraction of various types of information.

1.1 What Palladian is

Palladian is a collection of algorithms for text processing focused on **classification**, **extraction**, and **retrieval**. The concept of Palladian is system to reuse algorithms that are freely available and build upon them to drive research. When trying to learn and advance in a new field of research, one has to play around with many code snippets from various authors, this toolkit tries to ease this process by making external libraries accessible through a single interface. This way new algorithms can be quickly compared to the state-of-the-art. The best results from students at the Dresden University of Technology find their way into the toolkit allowing other users to create more advanced programs in the future.

Our main contributions to the research community are:

1. New algorithms in the fields of classification and extraction that can easily be tested and applied to other projects.
2. Single interface access to bundles of well established algorithms.
3. Helping make research more transparent by fomenting users to reproduce results with the included algorithms.

Figure 1.1 shows the main packages of Palladian. The focus is evidently on retrieval (Crawler, API access, feed reading...), preprocessing (tokenization, sentence splitting...), classification (KNN, Dictionary Classifier...), and extraction (named entities, tags...).

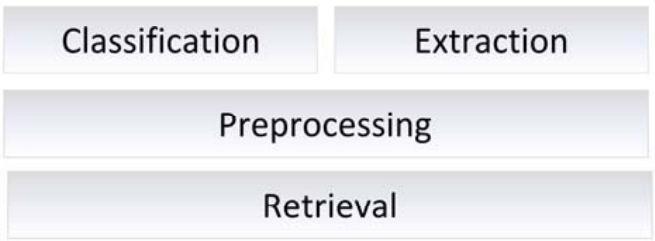

Figure 1.1: Important Packages of Palladian.

1.2 What Palladian is NOT

Palladian is not a full natural language processing suite nor does it contain a *full* set of algorithms in the fields of classification, extraction, and information retrieval. Palladian is not a commercial product either, we like to answer questions as soon and comprehensive as possible but cannot guarantee this support.

1.3 Who the Intended User is

In general everybody is welcome to use Palladian, but the researchers that would develop algorithms and would like to quickly test or compare them are the focus. Researchers likely to help other researchers, that won't mind a software that is not perfect, bug-free or commercial.

1.4 License

The complete source code is licensed under the Apache License 2.0. All source files should include the following license snippet at the very top.

```
Copyright 2010 David Urbansky, Klemens Muthmann
Licensed under the Apache License, Version 2.0 (the "License"); you may not
use this file except in compliance with the License. You may obtain a copy of
the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```

1.5 Evaluation Measures for IIR Systems

This sections outlines important evaluation measures which are typically used for comparing and scoring algorithms and techniques in (Internet) Information Retrieval. Our goal is to give just a brief overview, more details can be found in the referenced literature.

1.5.1 Precision, Recall, and F-measure

Considering a specific query (e. g. *apples OR oranges*), a set of relevant and non-relevant documents and a subset of retrieved – i. e. found – documents, we can distinguish between four disjoint sets, which can be classified as combinations of the attributes "true" or "false" with "positive" or "negative", as depicted in Figure 1.2. Documents which have been filtered out by the system errenously are classified as "false negatives", irrelevant documents which are presented to the user are classified as "false positives". Precision P and Recall R can be determined as stated in Equations 1.1 and 1.2. Intuitively, both take a range between 0 and 1 inclusively, where 1 denotes the optimum.

$$Precision = P = \frac{|\,relevant,\ retrieved\ docs\,|}{|\,retrieved\ docs\,|} = \frac{|\,true\ positives\,|}{|\,true\ positives\,| + |\,false\ positives\,|} \tag{1.1}$$

$$Recall = R = \frac{|\,relevant,\ retrieved\ docs\,|}{|\,relevant\ docs\,|} = \frac{|\,true\ positives\,|}{|\,true\ positives\,| + |\,false\ negatives\,|} \tag{1.2}$$

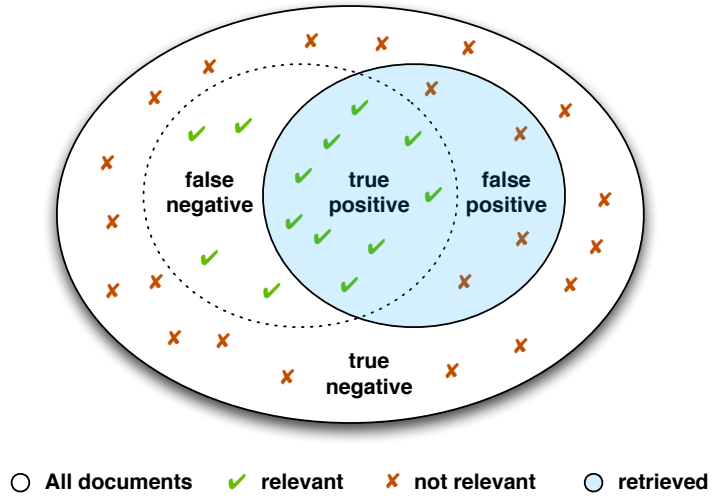

Figure 1.2: Set of Documents as Basis for Precision and Recall [44].

The F-measure combines both the Precsion and the Recall to one value. Usually the so called F_1 score is calculated by employing a harmonic mean as depicted in Equation 1.3. The universal F-measure which allows weighting, and thus prioritizing, Precision and Recall values differently is described in [33, p. 156].

$$F_1 = \frac{2 \cdot P \cdot R}{P + R} \tag{1.3}$$

1.5.2 Mean Squared Error, Root Mean Squared Error

The Mean Squared Error (MSE) is used for measuring the deviation of estimated or predicted numeric values from their actual values. Therefor, the average of the quadratic differences between each actual value y and the predicted value $\bar{y}$ is calculated. By squaring down the differences, the bigger errors are weighted stronger compared to small errors. The Root Mean Squared Error (RMSE) additionally extracts the square root from the MSE, as depicted in Equation 1.4.

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \bar{y}_i)^2} \tag{1.4}$$

1.5.3 Precision at k, Average Precision, Mean Average Precision

Precision, Recall and F-measure as described in Section 1.5.1 are metrics, which only consider unsorted sets of items. To evaluate ordered lists, which are ranked by relevance for example, the following measures can be employed.

"Precision at k" denotes the precision for a ranked sublist of the first k documents as illustrated in Equation 1.5. The binary function $rel(i)$ takes a value of 1, if the document at position i is relevant for the current query, 0 elsewise [25]. For example, $P(k = 10)$ with 5 relevant and 5 irrelevant documents results in a value of 0.5.

$$P(k) = \frac{\sum_{i=1}^{k} rel(i)}{k} \tag{1.5}$$

Equation 1.6 shows the Average Precision (AP), which is the sum of the precisions from Equation 1.5 where $rel(i)$ equals a value of 1, divided by the total count of relevant documents in the sublist of n results [25]. Table 1.1 gives a calculation example.

$$AP = \frac{\sum_{k=1}^{n} P(k) rel(k)}{R} \tag{1.6}$$

Rank k	relevant?	# relevant	P(k)	AP
1	✓	1	1	1
2		1	$1/2 = 0.5$	1
3	✓	2	$2/3 \approx 0.67$	$(1 + 2/3)/2 \approx 0.83$
4	✓	3	$3/4 = 0.75$	$(1 + 2/3 + 3/4)/3 \approx 0.81$
5	✓	4	$4/5 = 0.8$	$(1 + 2/3 + 3/4 + 4/5)/4 \approx 0.80$
6	✓	5	$5/6 \approx 0.83$	$(1 + 2/3 + 3/4 + 4/5 + 5/6)/5 \approx 0.81$
7		5	$5/7 \approx 0.71$	$(1 + 2/3 + 3/4 + 4/5 + 5/6)/5 \approx 0.81$

Table 1.1: Example for calculating Precision and Average Precision for a ranked list.

1.5.4 Mean Reciprocal Rank

1.6 Alternative and Complimentary Toolkits

Some functionalities of this toolkit are covered in other libraries. If you can't find the functionality you need in Palladian you might want to take a look at these alternatives. Some of the objectives of Palladian are to create new functionalities or improve existing ones, not to reinvent the wheel and redo work that functions well in other libraries.

1. **AlchemyAPI** [4] is a commercial web-service that can be used via several programming languages. The service offers named entity recognition, text classification, language identification, concept tagging, keyword extraction, content scraping, and web page cleaning. The service comes in 4 variants: free, basic, professional, and metered.

2. **Apache Mahout** [47] is a Java-based machine learning library. Its main features are collaborative filtering, user and item based recommenders, (fuzzy k-means clustering, mean shift clustering, latent dirichlet process allocation, singular value decomposition, parallel frequent pattern mining, complementary naive bayes classifier, and a random forest decision tree based classifier. The library is licensed under the Apache Software license.

3. **Balie** [8] is a Java-based information extraction library. Its main features are language identification, tokenization, sentence boundary detection, and named entity recognition (using dictionaries). The library is licensed under the GNU GPL and supports English, German, French, Romanian, and Spanish as input languages.

4. **Classifier4J** [40] is a very simple Java library for text classification. Besides Bayesian and vector classification, it offers a text summary feature.

5. **ContentAnalyst** [14] is a commercial platform for text analytics. The platform's main features are concept search, dynamic clustering, near-duplicate document identification, automatic summarization, text classification, and latent semantic indexing.

6. The **Dragon Toolkit** [62] is a Java-based development package for information retrieval and text mining. Its main features are text classification, text clustering, text summarization, and topic modeling.

7. **FreeLing** [7] is a natural language processing library written in C++. Its main features are Text tokenization, sentence splitting, morphological analysis, sSuffix treatment, retokenization of clitic pronouns, flexible multiword recognition, contraction splitting, probabilistic prediction of unkown word categories, named entity detection, recognition of dates, numbers, ratios, currency, and physical magnitudes, PoS tagging, chart-based shallow parsing, named entity classification, WordNet based sense annotation and disambiguation, Rule-based dependency parsing, and nominal correference resolution. It is licensed under GPL and supports Spanish, Catalan, Galician,

Italian, English, Welsh, Portuguese, and Asturian as languages. An online demo is available under `http://garraf.epsevg.upc.es/freeling/demo.php`.

8. **GATE** [15] is a Java-based text mining and processing framework. The framework itself comes with few text processing features but many plugins can be used and chained into a text engineering pipeline. The framework is licensed under the GNU Lesser General Public License.

9. The **Illinois Cognitive Computation Group** [20] has a list of ready to use programs for semantic role labeling, text chunking, named entity tagging, named entity discovery, PoS tagging, unsupervised rank aggregation, and named entity similarity metrics.

10. **JTMT** (Java Text Mining Toolkit) [54] is an assorted collection of Java code for various text mining and information retrieval tasks. Different components are explained in-depth in the author's blog. The toolkit is licensed under the Lesser GNU General Public License (LGPL).

11. **Julie NLP** [56, 18] is a Java-based toolkit of UIMA based text processing components. The toolkit can be used for semantic search, information extraction, named entity recognition, and text mining. The Toolkit is licensed under the Common Public License.

12. **Language Computer** [26] provides commercial products for sentence splitting, tokenization, PoS tagging, named entity recognition, co-reference resolution, attribute extraction, relationship extraction, event extraction, question answering, and text summarization.

13. **Lemur Project** [27] provides a toolkit with search engines and tools for research tasks in information retrieval and text mining. The project was founded in 2000 by the Center for Intelligent Information Retrieval (CIIR) at the University of Massachusetts, Amherst. Updates for the toolkit are released on a regular basis. The toolkit is written in C++, but also offers C# and Java APIs. Beside the Lemur toolkit itself, the project offers a search engine called "Indri" and a browser plugin to capture users' query and browsing behaviours.

14. **Lingo3G** [28] is a text clustering engine that organizes text collections into hierarchical clusters. The software is commercial but [52] offers an open source alternative for text clustering algorithms written in Java. The algorithms integrate with other programming or scripting languages such as PHP, Ruby, and C# too.

15. **LingPipe** [5] is a text processing toolkit using computational linguistics. LingPipe is written in Java. Its main features are topic classification, named entity recognition, clustering, PoS tagging, sentence detection, spelling correction, database text mining, string comparisons, interesting phrase detection, character language modeling, chinese word segmentation, hyphenation and syllabification, sentiment analysis, language identification, singular value decomposition, logistic regression, expectation maximization, and word sense disambiguation. LingPipe is available under a free license for academic use and several commercial licenses.

16. **Mallet** [35] is a Java-based toolkit for statistical natural language processing. Its main features are text classification, sequence tagging (PoS tagging), topic modeling, and numerical optimization. The toolkit is licensed under the Common Public License.

17. **MinorThird** [13] is a Java-based toolkit for text processing. Its main features are annotating text, named entity recognition, and text classification. The toolkit is licensed under the BSD license.

18. **MontyLingua** [29] is a Python and Java-based toolkit for natural language processing (English only). Its main features are tokenization, PoS tagging, lemmatization, and natural language summarization. The toolkit is free for non-commercial use and licensed under the MontyLingua version 2.0 License.

19. **MorphAdorner** [36] is a Java-based command line program for text processing. Its main features are language recognition, lemmatization, name recognition, PoS tagging, noun pluralization, sentence splitting, spelling standardization, text segmentation, verb conjugation, and word tokenization. The program is licensed under a NCSA style license.

20. **NaCTeM Software Tools** [37] are programs for natural language processing and text mining that are made available by the National Centre for Text Mining. The programs include functionality for PoS tagging, syntactic parsing, named entity recognition, sentence splitting, text classification, and sentiment analysis.

21. **NLTK** [31] is a Python-based natural language processing toolkit. Its main features are tokenization, stemming, PoS tagging, text classification, and syntactic parsing. The toolkit is licensed under the Apache 2.0 license.

22. **OpenCalais** [42] is a web service that performs named entity recognition, fact and event extraction. The web service is free for commercial and non-commercial use but limited to 50,000 transactions a day. A professional plan is available too including more transactions and an service license agreement.

23. **OpenNLP** [43] is a toolkit of various open source natural language processing packages. The goal of this toolkit is to integrate several stand-alone projects to increase the interoperability. Algorithms in this toolkit include but are not limited to tokenization and named entity recognition.

24. The **RASP System** [9] is a C and Lisp-based toolkit for natural language processing (English only). Its main features are tokenization, PoS tagging, lemmatization, morphological analysis, and grammar-based parsing. The toolkit is free for non-commercial use and licensed under the RASP System License.

25. The **Rosette Linguistic Platform** [46] is a software suite that can perform name translation, name matching, named entity recognition, morphological analysis, and language identification. The suite works for 55 European, Asian, and Arabic languages. The software is a commercial product.

26. **The Semantic Discovery Toolkit** [50] is a set of tools under active development focussing on NLP, machine learning, XML parsing, crawling, parallel processing. It further provides wrapper classes for several related implementations. The toolkit is under GNU LGPL.

27. **Stanford NLP** [51] is a set of Java-based natural language processing libraries. Their main features are PoS tagging, named entity recognition, Chinese word segmentation, and classification. The software distributions are licensed under the GNU Public License.

28. **SRILM – The SRI Language Modeling Toolkit** [53] is a C++-based toolkit for language modeling. Its main features are speech recognition, statistical tagging and segmentation, and machine translation. The toolkit is free for non-commercial use and licensed under the SRILM Research Community License.

29. **TextAnalyst** [55] is a commercial text processing software offering text summarization, semantic information retrieval, meaning extraction, and text clustering.

30. **VisualText** [58] is a natural language processing software that addresses named entity recognition, text indexing, text filtering, text classification, text grading, and text summarization.

31. **Weka** [19] is a Java-based machine learning and data mining library. The library contains a large set of machine learning algorithms such as Support Vector Machines, Neural Networks, Naive Bayes, k-nearest neighbor for but not limited to (text) clustering, (text) classification, and regression. The library is licensed under the GNU General Public License.

Chapter 2

Toolkit Structure

Palladian's source code is managed using Git[1], which offers several advantages compared to other version control systems like SVN. To get familiar with Git, we recommend looking at one of the several tutorials or cheat sheets which can be found with the search engine of your choice. Git is available for all major platforms. You can either use it via command line, utilize a graphical stand alone frontend like TortoiseGit[2] or install the plugin EGit[3] to integrate Git's functionality directly into Eclipse.

The Palladian Git repository is located at Bitbucket[4]. To receive access to the repository, first register an account at Bitbucket, then contact one of the Palladian developers and ask for an activation of your account, stating your username. The URL for the main Palladian repository is as follows:

```
https://bitbucket.org/palladian/palladian.git
```

To obtain a checkout using Git's command line tool, you typically enter:

```
git clone https://johndoe@bitbucket.org/palladian/palladian.git
```

2.1 Directories

The typical directory structure after a successful checkout looks as follow.

```
palladian
 |- config
 |- data
    |- evaluation
    |- temp
    |- test
 |- documentation
    |- book
    |- javadoc
    |- slides
 |- exe
 |- src
    |- main
       |- java
       |- resources
    |- test
```

[1] http://git-scm.com/
[2] http://code.google.com/p/tortoisegit/
[3] http://eclipse.org/egit/; if you are using Eclipse 3.6 or newer, you can install EGit using the Eclipse Marketplace.
[4] http://bitbucket.org/

```
      |- java
      |- resources
|- dev
```

2.2 Config Folder

The config folder contains a template for the default Palladian configuration file and the database schema needed to run some of the components. The files are explained in the following sections.

2.2.1 palladian.properties.default

The file `palladian.properties.default` is a template for the main Palladian configuration file. You need to copy this file and rename it to `palladian.properties`. The file may be located at one of three positions. The toolkit checks them in the following order:

1. The folder config inside folder on the local file system specified by the environment variable `$PALLADIAN_HOME`.
2. In the root of the classpath the application is run from. This usually is the root of the JAR you are running or the root folder where all `*.class` files are located.
3. Inside the `config` folder at the folder you are running your Palladian powered application from.

The configuration file is structured into several sections explained in the following paragraphs. Inside your own copy of `palladian.properties` you need to adapt these settings to your local requirements.

Model Paths

Dataset Paths

Database

Crawler Crawler settings for the ws.palladian.web.Crawler. All settings can be set in Java code as well.

```
# maximum number of threads during crawling
crawler.maxThreads = 10

# stop after x pages have been crawled, default is -1 and means unlimited
crawler.stopCount = -1

# whether to crawl within a certain domain, default is true
crawler.inDomain = true

# whether to crawl outside of current domain, default is true
crawler.outDomain = true

#  number of request before switching to another proxy, default is -1 and means never switch
crawler.switchProxyRequests = -1

# list of proxies to choose from
crawler.proxyList = 83.244.106.73:8080
crawler.proxyList = 83.244.106.73:80
crawler.proxyList = 67.159.31.22:8080
```

Classification Classification settings for the ws.palladian.classification.page.ClassifierManager.

```
# page classification configurations

# percentage of the training/testing file to use as training data
classification.page.trainingPercentage = 80

# create dictionary on the fly (lowers memory consumption but is slower)
classification.page.createDictionaryIteratively = false

# alternative algorithm for n-gram finding (lowers memory consumption but is slower)
classification.page.createDictionaryNGramSearchMode = false

# index type of the classifier:
# 1: use database with single table (fast but not normalized and more disk space needed)
# 2: use database with 3 tables (normalized and less disk space needed but slightly slower)
# 3: use lucene index on disk (slow)
classification.page.dictionaryClassifierIndexType = 2

# if page.dictionaryClassifierIndexType = 1 or 2, specify type of database
# 1: mysql (client server)
# 2: h2 (embedded)
classification.page.databaseType = 2
```

API keys The api keys that are used by the toolkit components are specified here. You may need to apply for API keys at the provider's page.

```
api.alchemy.key =
api.bing.key =
api.google.key =
api.google.tranlsate.key =
api.hakia.key =
api.opencalais.key =
api.yahoo.key =
api.yahoo_boss.key =
api.bitly.login =
api.bitly.key =
api.mixx.key =
api.reddit.username =
api.reddit.password =

# access to Collecta XMPP and REST API
# see: http://developer.collecta.com/APIindex/
api.collecta.key =

api.majestic.key =
api.compete.key =
```

Database Configuration Database settings for the ws.palladian.persistence.DatabaseManager.

```
db.driver = com.mysql.jdbc.Driver
db.jdbcUrl = jdbc:mysql://localhost:3306/tudiirdb?useServerPrepStmts=false&cachePrepStmts=false
db.username = root
db.password =

# false = persistent, true = RAM only (10x faster) (only for H2 databse)
db.inMemoryMode = true
```

Feed Discovery The feed discovery is used to detect RSS and Atom feeds on crawled web pages. You can specify related settings in this section of the property file.

```
# search engine for discovery, see SourceRetrieverManager for avail. constants
# for example ...: Yahoo Boss = 5, Yahoo Boss News = 10
feedDiscovery.searchEngine = 6

# maximum number of threads for discovery
feedDiscovery.maxDiscoveryThreads = 10

# feed urls containing these fragments will be ignored
feedDiscovery.discoveryIgnoreList = gdata.youtube.com
feedDiscovery.discoveryIgnoreList = wikipedia.org
feedDiscovery.discoveryIgnoreList = podcast
feedDiscovery.discoveryIgnoreList = comments
feedDiscovery.discoveryIgnoreList = forum
feedDiscovery.discoveryIgnoreList = gallery
feedDiscovery.discoveryIgnoreList = special:recentchanges
feedDiscovery.discoveryIgnoreList = popularthreads
feedDiscovery.discoveryIgnoreList = answers.yahoo.com
feedDiscovery.discoveryIgnoreList = feeds.digg.com

# location of the list for feeds discovered by the Crawler
feedDiscovery.crawlerDiscoveryList = data/status/crawler_discovered_feeds.txt

# if enabled, only the first feed of each page is returned
# if disabled, all Atom- or RSS feeds are returned
feedDiscovery.onlyPreferred = true

# maximum number of threads when aggregating/adding feeds
feedDiscovery.maxAggregationThreads = 5

feedDiscovery.downloadAssociatedPages = true
```

Page Segmentation The page segmentation settings.

```
# length of q-grams
pageSegmentation.lengthOfQGrams = 9

# amount of q-grams
pageSegmentation.amountOfQGrams = 5000

# threshold needed to be similar
pageSegmentation.similarityNeed = 0.689

# maximal depth in DOM tree
pageSegmentation.maxDepth = 100

# number of similar documents needed
pageSegmentation.numberOfSimilarDocuments = 5

# threshold of variability
# from green to red; from low variability to high variability
pageSegmentation.step1 = 0.0
pageSegmentation.step2 = 0.14
pageSegmentation.step3 = 0.28
pageSegmentation.step4 = 0.42
```

```
pageSegmentation.step5 = 0.58
pageSegmentation.step6 = 0.72
pageSegmentation.step7 = 0.86
```

2.3 Data Folder

The data folder contains files that are used during runtime of several components.

2.3.1 Model Folder

The models folder contains learned models that can be reused. See section **??** for an explanation on how to access our extensive collection of data sets and models.

2.3.2 Temp Folder

The temp folder is not part of the repository. Some functions may however create this folder and write temporary data. It is not part of the SVN and should not be committed when automatically created.

2.3.3 Test Folder

The test folder contains data that is used for running jUnit tests. It is not part of the SVN and should not be committed when automatically created.

2.4 Documentation Folder

The documentation folder contains help files to understand the toolkit. This very document is located in the book folder and a Javadoc can be found here too.

2.5 Exe Folder

The exe folder contains all runnable jar files in separate folders including a sample script to run the program and a readme.txt that explains the run options.

2.6 Src Folder

The src folder contains all source files of the toolkit. You may need to put the log4j.properties file here in order to use custom logging settings. Alternatively you include the config folder in the class path.

Chapter 3

Capabilities

In this chapter we discuss the functionalities of Palladian in detail. The purpose of each section is to give a general understanding of how the algorithms work and how to use them practically.

3.1 Classification

3.1.1 Text Classification

Text classification is the process of assigning one or more categories to a given text document. There are several types of text classification which are shown in Figure 3.1. Palladian can classify text in a single category, multiple categories, or in a hierarchical manner.

The text classification components are built from scratch and do not rely on external libraries such as Weka. In this section we will explain which features can be used for the classification, the basic theory of the classifiers, and how the performance of a classifier can be evaluated. In each section, we will describe theory and how the classification components can be used programmatically.

Classification Type

Palladian supports simple single category classification, tagging, and hierarchical classification. Each classifier needs to know in which type it has to perform classification. This information is stored in the ClassificationTypeSetting object which is then passed to the classifier. Please read the Javadoc of that class for more detailed information.

Features

Features are the input for a classifier. In text classification we have a long string as an input from which we can derive several features. All of Palladian's classifiers work with n-grams. N-grams are sets of tokens of the length n. Palladian can preprocess text with character or word-level n-grams. See Section 3.4.3 for more details.

Sometimes you do not want to have all n-grams to be features for the classifier. You can simply disallow certain features by putting them in a stop word list. These n-grams will then be ignored for the classification.

All these settings are stored in a FeatureSetting object which is passed to the classifier. Please read the Javadoc of that class for more detailed information.

Text Classifiers

The text classifiers perform the actual classification task by calculating the most relevant category (or categories) for the input document. Two text classifiers are implemented, a dictionary based classifier and a k-nearest neighbor classifier. Both are explained in more detail in the following paragraphs.

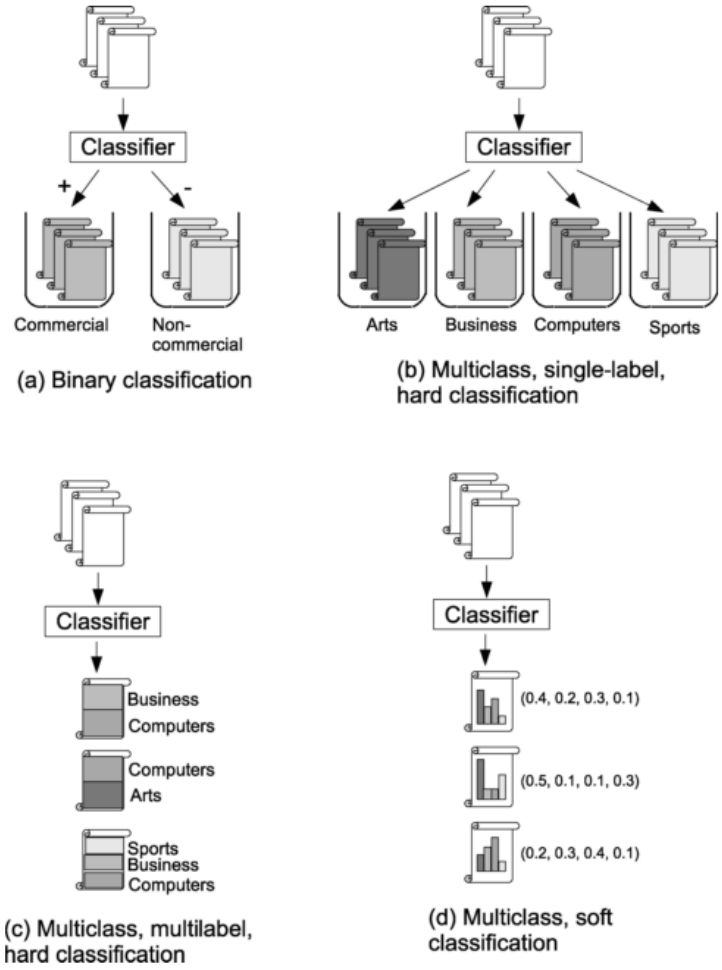

Figure 3.1: Types of classification[61].

K-Nearest Neighbor Text Classifier The KNN classifier uses the n-grams of the training documents to place them in a high dimensional vector space. The dimensions of the space equal the total number of available n-grams. Each training document is therefore a vector in that highly dimensional space. A new, unclassified document is now put into that vector space and by using distance function the k nearest neighbors are found for that document. Each of these neighbors votes with its own class, the more votes for one class the more likely that the new document belongs to that class.

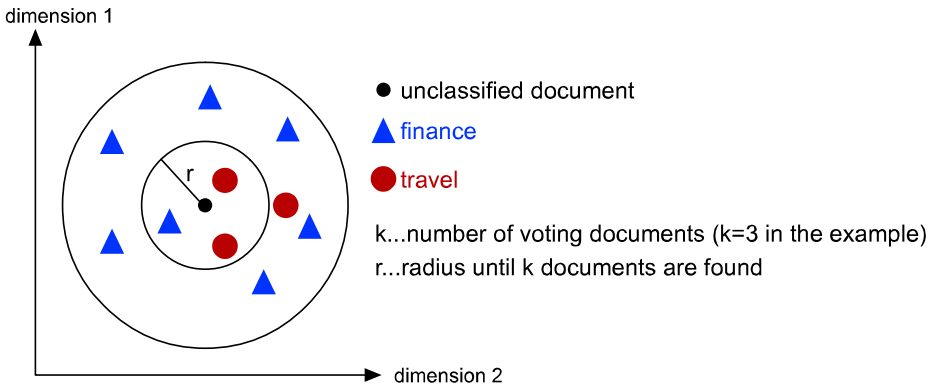

Figure 3.2: A simple KNN example.

Figure 3.2 shows a simple example of how the KNN classifier works. We limited the dimension to two for easier understanding. In this scenario we want to classify the given document (black dot in the middle) into one of the two categories "finance" (blue triangles) or "travel" (red dots). We calculate the distance between the new document and all training documents and consider the votes of the nearest three. In the example, two of these three document vote for "travel" which would let us classify our input document into that class.

The distance between two documents is calculated as shown in Equation 3.1 where $d1$ and $d2$ are the two documents. The shorter the distance, the more similar the documents.

$$distance(d1, d2) = \frac{1}{numberOfMatchingNGrams} \tag{3.1}$$

Dictionary-Based Classifier The dictionary-based classifier[1] learns how probable each n-gram is for each given category and assigns the most probable category (or categories) to the input document.

A dictionary is built at training stage by counting and normalizing the co-occurrences of one n-gram and a category. The dictionary might then look as shown in Table 3.1 where each column is a category (finance, travel, and science) and each row is an n-gram. In each cell we now have the learned relevance for each n-gram and category $relevance(ngram, category)$. The sum of the relevances in each row must add up to one.

Table 3.1 shows an example dictionary matrix. The n-gram "money" is more likely to get the category "finance" ($relevance(money, finance) = 0.6$) than "science" ($relevance(money, science) = 0.25$) while the n-gram "beach" is most likely to appear in the category "travel" ($relevance(beach, travel) = 0.85$.

n-gram	finance	travel	science
money	0.6	0.15	0.25
beach	0.1	0.85	0.05
paper	0.3	0.2	0.5

Table 3.1: N-Gram dictionary with relevances for categories.

To classify a new document, we again create all n-grams, look up the relevance scores in the dictionary and assign the categories with the highest probability. The probability for each category and given document is calculated as shown in Equation 3.2 where $N_{Document}$ is the set of n-grams for the given document.

$$CategoryProbability(category, document) = \sum_{n \, \epsilon \, Nurl} relevance(n, category) \tag{3.2}$$

The dictionary can be stored in memory, in an embedded H2 database, or in a client/server MySQL database. These settings can be made in the classification.conf file in the conf folder (see Section 2.2).

Evaluation

In order to find out which classifier works best with which feature settings, you can evaluate these combinations. The ws.palladian.classification.page.ClassifierManager has the $learnBestClassifier$ method to run the evaluation on all given classifiers with the given evaluation setting object ws.palladian.classification.page See Section 3.1.1 for more details of how to perform the evaluation programmatically.

The output of the evaluation will be three csv files that hold information about the combinations of classifier, dataset, training percentage, and the final performance for the combination. The performance is measured in precision, recall, and F1. The three files are stored in the data/temp folder and hold the following information.

[1] This classifier won the first Research Garden (`http://www.research-garden.de`) competition where the goal was to classify product descriptions into 8 different categories. See press release at `http://www.research-garden.de/c/document_library/get_file?uuid=e60fa8da-4f76-4e64-a692-f74d5ffcf475&groupId=10137`

averagePerformancesDatasetTrainingFolds.csv This file holds the performance measures for each classifier, averaged over all given datasets, training percentages, and folds in the cross validation.

averagePerformancesTrainingFolds.csv This file holds the performance measures for each classifier and dataset combination, averaged over all given training percentages and folds in the cross validation.

averagePerformancesFolds.csv This file holds the performance measures for each classifier, dataset, and training percentage combination, averaged over all given folds in the cross validation.

Best Practices

This section describes how to prepare training and testing data to learn a model, evaluate, and use a text classifier.

Preparing the Training/Testing Data The data can be specified in a simple text file. There are three classification options, namely, one-category classification, hierarchical classification and multi-category classification. They all require a similar structure of the data.

One-Category Classification We write one URL and one category separated with a single space on each line. For example:

```
http://www.google.com search
http://www.fifa.com sport
http://www.oscars.com entertainment
```

Hierarchical Classification We write one URL and multiple categories separated with a single space on each line. The categories must be in the correct order, so the first category is the main one, all following are subcategories of each other. For example:

```
http://www.google.com search search_engine
http://www.fifa.com sport team_sports soccer
http://www.oscars.com entertainment movies awards usa
```

Multi-Category Classification We write one URL and multiple categories separated with a single space on each line. The order of the categories (tags) does not matter. For example:

```
http://www.google.com search image_search video_search
http://www.fifa.com soccer sport free_time fun ball_game results to_read
http://www.oscars.com entertainment movies films awards watch video stars
```

Training a classifier Before we can use a classifier, we need to learn a model. The model is an internal representation of the learned data. After learning a model, a classifier can applied to unseen data. We now have prepared the training and testing data so we can now learn the models. The classifier is saved as a lucene index or a database under the name of the classifier with a "Dictionary" suffix. The result of the learning are three files: the classifier ("CLASSIFIER.ser"), the dictionary object ("CLASSIFIERDictionary.ser"), and the actual dictionary as the index or database (e.g. "CLASSIFIERDictionary.h2.db"). More settings can be configured in the config/classification.conf file. See 2.2.1 for more information.

Listing 3.1 shows an example for how to train and save a classifier. In this case we train a classifier that can classify the language of given documents. As training data we use a list of web pages of different languages from Wikipedia.

```
// create a classifier mananger object
ClassifierManager classifierManager = new ClassifierManager();

// specify the dataset that should be used as training data
Dataset dataset = new Dataset();

// set the path to the dataset
String dsPath = "data/datasets/classification/language/index.txt";
dataset.setPath(dsPath);

// tell the preprocessor that the first field in the file is a link to
// the actual document
dataset.setFirstFieldLink(true);

// create a text classifier by giving a name and a path
// where it should be saved to
String dcn = "LanguageClassifier";
String dcp = "data/models/languageClassifier/";
TextClassifier classifier = new DictionaryClassifier(dcn,dcp);

// specify the settings for the classification
ClassificationTypeSetting cts = new ClassificationTypeSetting();

// we use only a single category per document
cts.setClassificationType(ClassificationTypeSetting.SINGLE);

// we want the classifier to be serialized in the end
cts.setSerializeClassifier(true);

// specify feature settings that should be used by the classifier
FeatureSetting featureSetting = new FeatureSetting();

// we want to create character-level n-grams
featureSetting.setTextFeatureType(FeatureSetting.CHAR_NGRAMS);

// the minimum length of our n-grams should be 3
featureSetting.setMinNGramLength(3);

// the maximum length of our n-grams should be 5
featureSetting.setMaxNGramLength(5);

// we assign the settings to our classifier
classifier.setClassificationTypeSetting(classificationTypeSetting);
classifier.setFeatureSetting(featureSetting);

// now we can train the classifier using the given dataset
classifierManager.trainClassifier(dataset, classifier);
```

Code Listing 3.1: Training a classifier.

Using a classifier After we trained a model for a classifier we can apply it to unseen data. Let's use the model we just trained to classify the language of a new document.

Listing 3.2 shows how to use a trained classifier.

```
// the path to the classifier we want to use
String path = "data/models/languageClassifier/LanguageClassifier.ser";

// load the language classifier
TextClassifier classifier = ClassifierManager.load(path);
```

```
6
7  // create a classification document that holds the result
8  ClassificationDocument classifiedDocument = null;
9
10 // classify the little text (if classifier works it would say Spanish)
11 classifiedDocument = classifier.classify("Yo solo  s    que no  s    nada.");
12
13 // print the classified document
14 System.out.println(classifiedDocument);
```

Code Listing 3.2: Use a trained text classifier.

You can also try the language classifier online at `http://www.webknox.com/wi#detectLanguage`.

Evaluating a Classifier To get an idea of how good a trained classifier works, we can evaluate it using test data which is structured the same way as the training data. Listing 3.3 shows how to evaluate a trained classifier, you will see that is very similar to training a classifier. Make sure that you evaluate the classifier using disjunct data, otherwise the evaluation results are invalid .

```
1  // create a classifier mananger object
2  ClassifierManager classifierManager = new ClassifierManager();
3
4  // the path to the classifier we want to use
5  String path = "data/models/languageClassifier/LanguageClassifier.ser";
6
7  // specify the dataset that should be used as testing data
8  Dataset dataset = new Dataset();
9
10 // the path to the dataset (should NOT overlap with the training set)
11 dataset.setPath("data/datasets/classification/language/index.txt");
12
13 // tell the preprocessor that the first field in the file is a link
14 // to the actual document
15 dataset.setFirstFieldLink(true);
16
17 // load the language classifier
18 TextClassifier classifier = ClassifierManager.load(path);
19
20 // now we can test the classifier using the given dataset
21 ClassifierPerformance classifierPerformance = null;
22 classifierManager.testClassifier(dataset, classifier);
```

Code Listing 3.3: Evaluating a trained text classifier.

Testing parameter combinations As you have seen, you can train the classifier using different parameters. So how can you be sure that you set the parameters correctly? Do they work well on different datasets? Is the chosen classifier always better than others? In order to answer these questions with hard data you can automatically run different combinations of classifiers, settings, and datasets as shown in Figure 3.3. The green line shows the combination that was found to perform best. In the end you will get one evaluation csv with information about how the combination performed. You can then manually pick the best performing settings.

Listing 3.4 shows you how to do just that.

```
1  ClassifierManager classifierManager = new ClassifierManager();
2
3  // build a set of classification type settings to evaluate
4  List<ClassificationTypeSetting> ctsList;
5  ctsList = new ArrayList<ClassificationTypeSetting>();
```

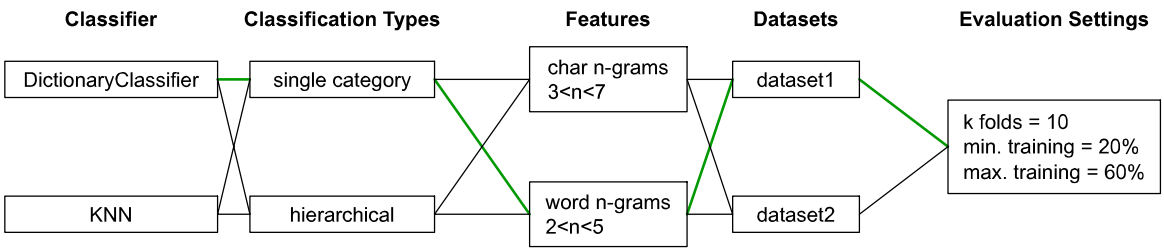

Figure 3.3: Combinations for training a solid classifier.

```
6  ClassificationTypeSetting cts = new ClassificationTypeSetting();
7  cts.setClassificationType(ClassificationTypeSetting.SINGLE);
8  cts.setSerializeClassifier(false);
9  ctsList.add(cts);
10
11 // build a set of classifiers to evaluate
12 List<TextClassifier> classifiers = new ArrayList<TextClassifier>();
13 TextClassifier classifier = null;
14 classifier = new DictionaryClassifier();
15 classifiers.add(classifier);
16 classifier = new KNNClassifier();
17 classifiers.add(classifier);
18
19 // build a set of feature settings for evaluation
20 List<FeatureSetting> featureSettings = new ArrayList<FeatureSetting>();
21 FeatureSetting fs = null;
22 fs = new FeatureSetting();
23 fs.setTextFeatureType(FeatureSetting.CHAR_NGRAMS);
24 fs.setMinNGramLength(3);
25 fs.setMaxNGramLength(7);
26 featureSettings.add(fs);
27
28 fs = new FeatureSetting();
29 fs.setTextFeatureType(FeatureSetting.CHAR_NGRAMS);
30 fs.setMinNGramLength(2);
31 fs.setMaxNGramLength(5);
32 featureSettings.add(fs);
33
34 fs = new FeatureSetting();
35 fs.setTextFeatureType(FeatureSetting.WORD_NGRAMS);
36 fs.setMinNGramLength(2);
37 fs.setMaxNGramLength(5);
38 featureSettings.add(fs);
39
40 // build a set of datasets that should be used for evaluation
41 Set<Dataset> datasets = new HashSet<Dataset>();
42 Dataset dataset = new Dataset();
43 dataset.setPath("dataset1.txt");
44 datasets.add(dataset);
45 dataset = new Dataset();
46 dataset.setPath("dataset2.txt");
47 dataset.setSeparationString("#");
48 datasets.add(dataset);
49
50 // set evaluation settings
51 EvaluationSetting evaluationSetting = new EvaluationSetting();
52 evaluationSetting.setTrainingPercentageMin(20);
53 evaluationSetting.setTrainingPercentageMax(80);
54 evaluationSetting.setkFolds(5);
```

```
55 evaluationSetting.addDataset(dataset);
56
57 // let's take the time
58 StopWatch stopWatch = new StopWatch();
59
60 // train and test all classifiers in all combinations
61 classifierManager.learnBestClassifier(ctsList, classifiers,
62                                        featureSettings,
63                                        evaluationSetting);
64
65 System.out.println("finished training and testing classifier
66           combinations in " + stopWatch.getElapsedTimeString());
```

Code Listing 3.4: Learning the best parameter combination for a text classifier.

3.1.2 Language Detection

Language detection is the task of determining the language of a given text. In the last section we explained how a text classifier can be trained. We used this training method to learn a dictionary based classifier for language detection and evaluated it against other state-of-the-art algorithms. We compared our algorithm to JLangDetect [23], Google Translation API [17], and the Alchemy API [3]. We used the Multilingual Parallel Corpus of the Joint Research Centre of the European Commission in version 3.0 [24]. This corpus contains about half a million documents in 22 languages about politics. Since we could not train the language detectors from Google and Alchemy, the comparison is not fair to both of them but we put the results here to see how they are performing with their classifiers. Due to API restrictions we had to crop the length of the text to maximum of 100 characters. We trained both, JLangDetect and Palladian on a sample of 100 documents per language. JLangDetect was trained with character level n-grams between one and three[2] while Palladian was trained with word level n-grams between one and three. We then tested all algorithms an a test set containing again 100 documents per language. Since the classifier performance is very sensitive to the length of the given text, we evaluated all algorithms with short texts ($\leq$ 10 characters), medium texts ($\leq$ 30 characters), and long texts (no length limitation, but usually about 20 to 200 sentences). Table 3.2 shows the evaluation results. All results in brackets are not representative since many documents were not classified by the API.

System	Short Texts	Medium Texts	Long Texts
JLangDetect	**74.64%**	87.91%	99.64%
Alchemy API	(97.78%)	69.35%	(88.27%)
Google API	57.32%	80.44%	(99.21%)
Palladian	62.36%	**88.18%**	**99.91%**

Table 3.2: Evaluation results for language detection systems.

As we can see, JLangDetect performs very well across all text lengths. Palladian is, however, slightly superior as soon as the text length is longer than about 10 to 20 characters.

3.2 Extraction

3.2.1 Keyword Extraction / Controlled Tagging

It is often interesting which words describe a given text most. These words are often called "keywords" or "tags". Palladian is able to use a weighted, controlled vocabulary and perform tagging of a text using TF-IDF and tag correlations.

[2]This setting has been said to be work well by the author of the JLangDetect, see `http://www.jroller.com/melix/entry/nlp_in_java_a_language`

The ControlledTaggerIndex is the basis for the tagging. This index contains information about the controlled tag vocabulary, tag frequencies, a stem-map, and a correlation table for all tags. The tag index is built using training data of single text files that have been assigned with weighted tags. This index can be serialized as a model file for later usage.

To tag a new text with the most relevant keywords, the ControlledTagger tokenizes the input text and stems the tokens, which means that tokens like "blogging" and "blogs" are consolidated to the common term "blog".

All tokens that match a tag from the controlled vocabulary in the index are held in a bag-of-words which also stores the frequencies of the tokens in the input text. Using this data, we can calculate the TF-IDF values for each tag candidate. The TF-IDF scores are used to rank the tags initially. The determined tag candidates are then processed by two re-ranking steps to improve the final results:

Prior probabilities considers the probability of a tag occuring in the training data. This way, more popular tags are prioretized.

Correlations considers the probability of a specific pair of tags co-occuring in the training data. For example, the trained model might suggest a strong correlation between the pair "apple" and "iphone". The strenghts of such correlations can be taken into account for the re-ranking.

ControlledTagger was designed to be trained with large amounts of manually tagged documents. For our experiments we relied on the DeliciousT140 dataset which can be obtained from [38]. The dataset was crawled from Delicious and contains over 140.000 tagged documents. For convenient access to the dataset, Palladian provides the class DeliciousDatasetReader. The DeliciousDatasetReader gives access to the XML-based dataset and supports various filter operations. For a usage example please take a look at the main method of the class.

The ControlledTagger can be configured using the ControlledTaggerSettings class. You can configure the following aspects:

1. TF-IDF threshold under which the tags should be discarded or alternatively, the number of fixed tags per document.
2. The correlation re-ranking modus.
3. The stemmer to use. We can choose between different SnowballStemmer implementations for various natural languages. For further informations about Snowball, please consult [49].
4. The stop words list.
5. A list of regular expression that the tags need to match in order to be assigned.

Listing 3.5 shows the training of the ControlledTagger with data from Delicious and the finally the serialization of the trained model.

```
1  // set up the ControlledTagger
2  final ControlledTagger tagger = new ControlledTagger();
3
4  // tagging parameters are encapsulated by ControlledTaggerSettings
5  ControlledTaggerSettings taggerSettings = tagger.getSettings();
6
7  // create a DeliciousDatasetReader + Filter for training
8  DeliciousDatasetReader reader = new DeliciousDatasetReader();
9  DatasetFilter filter = new DatasetFilter();
10 filter.addAllowedFiletype("html");
11 filter.setMinUsers(50);
12 filter.setMaxFileSize(600000);
13 reader.setFilter(filter);
14
15 // train the tagger with 20.000 train documents from the dataset
16 DatasetCallback callback = new DatasetCallback() {
```

```
17      @Override
18      public void callback(DatasetEntry entry) {
19          String content = FileHelper.readFileToString(entry.getPath());
20          content = HTMLHelper.htmlToString(content, true);
21          tagger.train(content, entry.getTags());
22      }
23  };
24  reader.read(callback, 20000);
25
26  // save the model for later usage
27  tagger.save("data/models/controlledTaggerModel.ser");
```

Code Listing 3.5: Training the controlled tagger.

Listing 3.6 shows how to load a trained model into the ControlledTagger and use it for tagging text contents.

```
1  ControlledTagger tagger = new ControlledTagger();
2
3  // load the trained model, this takes some time;
4  // if you will tag multiple documents,
5  // make sure to move this outside the loop!
6  tagger.load("data/models/controlledTaggerModel.ser");
7
8  // assign tags according to a web page's content
9  PageContentExtractor extractor = new PageContentExtractor();
10 String content = extractor.getResultText(
11         "http://arstechnica.com/open-source/news/2010/10/" +
12         "mozilla-releases-firefox-4-beta-for-maemo-and-android.ars");
13 List<Tag> assignedTags = tagger.tag(content);
14
15 // print the assigned tags
16 CollectionHelper.print(assignedTags);
```

Code Listing 3.6: Using the controlled tagger.

The keyword extraction is state-of-the-art and beats the comparable system "Maui"[41] by 18.9% in the F1-Score. More information about the controlled tagging can be found in [44].

3.2.2 Web Page Content Extraction

Content oriented web pages such as blogs or news articles contain not only the text of interest but also clutter such as the navigation, footer, header, and ads. In order to automatically process the text without the clutter, we need to extract the text content. Figure 3.4 shows an example web page where the main article is in the green box. Everything else is just clutter and should not be extracted when looking for the unique article.

To perform this kind of extraction we could use wrapper approaches which are explained in more detail in [10]. However, these approaches require manual wrapper construction or semi-supervised learning which is too cumbersome. Another approach would be to detect the template of the web page and get only the contents of the main block (the green box). Template detection usually works by comparing one page with several other pages of the same domain to find the fix and changeable contents. A template based approach is available in Palladian, check chapter 3.2.3. This however requires several HTTP requests and therefore more time and bandwidth.

Palladian offers two approaches together with thier corresponding implementations which are based on analyzing single pages and employing certain heuristics to identify relevant sections of web pages:

`PalladianContentExtractor` extracts clean sentences from (English) texts. While short phrases are not included in the output, this approach will extract all readable content from a page without distinguishing specific page areas, like content or comment sections.

Figure 3.4: Article content of a web page marked with a green box [44].

`ReadabilityContentExtractor` is more focused on general content. It works on the page's DOM trees; each DOM element is analyzed and ranked based on the length of the contained text, the link density, and the frequency of other elements in relation to the text length. Longer text fragments usually indicate a relevant part of the article, while a high link density is more likely to indicate that the analyzed element is part of the navigation for example. Also the attributes "id" and "class" are analyzed. If the attribute values contain keywords such as "entry", "content", or "text", the element is more likely to contain relevant content as if the attribute values contain keywords such as "header", "footer", or "sidebar". The implementation of the web page content extraction adopted great parts of the Firefox extension "Readability"[3][4]. The used heuristics in Readability have shown to be quite accurate for a wide range of web pages.

Additionaly, Palladian provides a wrapper class for Boilerpipe[5], a Java library which is based on the algorithms described in [12]. This gives Palladian's user the possibility to access three different content extraction techniques via a common interface. Listing 3.7 gives a short usage example.

```
1 WebPageContentExtractor extractor = new ReadabilityContentExtractor();
2 // or
3 // WebPageContentExtractor extractor = new PalladianContentExtractor();
4 // WebPageContentExtractor extractor = new BoilerpipeContentExtractor();
5
6 String url = "http://www.wired.com/gadgetlab/2010/05/iphone-4g-ads/";
7
```

[3] http://lab.arc90.com/experiments/readability/

[4] By now, a bunch of alternative Java-based Readability ports are available:
https://github.com/ifesdjeen/jReadability,
https://github.com/basis-technology-corp/Java-readability,
https://github.com/karussell/snacktory

[5] http://code.google.com/p/boilerpipe/

```
8  // this method is heavily overloaded and accepts various types of input
9  extractor.setDocument(url);
10
11 // get the main content as text representation
12 String contentText = extractor.getResultText();
13
14 // get the main content as DOM representation
15 Document contentDocument = extractor.getResultDocument();
16
17 // get the title
18 String title = extractor.getResultTitle();
```

Code Listing 3.7: Using the PageContentExtractor.

We evaluated our Readability-inspired approach against Boilerpipe. For evaluation we used the "L3S-GN1" data set which is provided by the authors of Boilerpipe and consists of 621 manually annotated web pages with news articles from 408 different sites. The annotations which were assigned by humans categorize areas of web pages into different groups "Headline", "Full text", "Supplemental", "Related content", "Comments" and "Not content", where "Full text" represents the main article content of a web page (green box in figure 3.4).

We used both content extraction approaches on the data set and scored their results by comparing their extracted text content with the human extracted content using Levenshtein[6] similarity. Table 3.3 shows the results of our evaluation. The number of wins denotes how many times one approach achieved a better e. g. more similar result than the other.

System	Average Similarity	# Wins	# Errors
Boilerpipe (`ArticleExtractor`, version 1.1)	88.93%	143	6
Readability port (SVN revision 1444)	90.91%	474	2

Table 3.3: Evaluation results for web page content extraction approaches.

3.2.3 Web Page Template Detection

3.2.4 Named Entity Recognition

Named Entity Recognition (NER) is the task of recognizing and disambiguating known and unknown entities in documents. In order to understand the task we need to understand what an *entity* is. An entity is a collection of rigidly designated chunks of text that refer to exactly one or multiple identical, real or abstract concept instances. These instances can have several aliases and one name can refer to different instance [60]. So for example "Iron Man 2" and "Iron-Man 2" are two different chunks of text which however refer to the same movie. Common types of entities are people (e.g. "Jim Carrey"), locations (e.g. "Los Angeles"), and organizations (e.g. "Google Inc.").

A named entity recognizer is therefore a system or technique that tries to detect entities in a given natural language text. For example, an NER system that is used to detect people, locations, and organizations should be able to find the bold entities in the following text:

Bill Gates founded **Microsoft** with **Paul Allen** in 1975 in **Albuquerque, New Mexico**. The headquarter is located in **Redmond, Washington** and the company is led by CEO **Steve Ballmer**.

Taggers

Palladian wraps 8 Named Entity Recognizers using a single interface making it easier to test and substitute them in real world applications.

[6] `http://www.merriampark.com/ld.htm`

Alchemy The Alchemy API is a web-based service that also offers Named Entity Recognition. Using this NER requires the application to have a developer key and Internet access. Alchemy covers a wide range of concepts. For more details see `http://www.alchemyapi.com/api/entity/types.html` or the JavaDoc.

Illinois Learning-based Java Students from the University of Illinois developed this Recognizer [20]. The recognizer comes with a large set of lists that help the recognizer to tag concepts such as corporations, countries, jobs, movies, people, songs, and more. More information can also be found on `http://l2r.cs.uiuc.edu/~cogcomp/asoftware.php?skey=FLBJNE`.

Julie The Julie Lab of the University of Jena added a named entity recognizer to their NLP tool-suite [18]. The tagger is based on conditional random fields and comes with three models from the biomedical domain. Other models can however trained using Palladian. For more information see the pdf which is contained in their stand-alone download from `http://www.julielab.de/Resources/Software/NLP+Tools/Download/Stand_alone+Tools.html`.

LingPipe The LingPipe library [5] also offers an NER. Unfortunately the tagger comes with no models. Models for each desired concept need to be trained manually. More information can be found on `http://alias-i.com/lingpipe/demos/tutorial/ne/read-me.html`.

OpenCalais The Open Calais API [42] is a web-based service that also offers Named Entity Recognition. The service requires a developer key and access to the Internet. Open Calais covers many concept as covered on the documentation `http://www.opencalais.com/documentation/calais-web-service-api/api-metadata/entity-index-and-definitions` and in the JavaDoc.

OpenNLP The OpenNLP toolkit [43] provides a maximum entropy based approach for named entity recognition. The toolkit comes with a small set of pre-trained models covering concepts such as locations, organizations, and persons. More details can be found in the JavaDoc.

Stanford As part of the Stanford NLP library [51] [22] developed a named entity recognizer that is based on conditional random fields (CRF). The tagger was trained on the CoNLL 2003 data and comes with a small set of classifiers that can recognize persons, locations, and organizations. For more details see `http://www-nlp.stanford.edu/software/crf-faq.shtml` and the JavaDoc.

TUD The TUD named entity recognizer was developed at the Dresden University of Technology. The recognizer is based on a list lookup and dictionary classification approach. So far no trained models exist for this tagger.

Tagging a Text

To use an NER you need two things: A text that you want to tag and a model for the tagger that has been trained to recognize the entities you want to find in the given text. The tagging works the same for all NERs and is shown in Listing 3.8.

```
1 // create the tagger
2 NamedEntityRecognizer ner = new TUDNER();
3
4 // the text to be tagged
5 String inputText = "The iphone 4 is the successor of the iphone 3gs.";
6
7 // the path to the model which should be used for tagging
8 String modelPath = "data/models/tudner/phone.model";
9
10 // tag the text using the specified model
```

```
String taggedText = ner.tag(inputText, modelPath);
System.out.println(taggedText);
// desired output
// The <PHONE>iphone 4</PHONE> is the successor
// of the <PHONE>iphone 3gs</PHONE>.

// avoid loading the model each time using it
ner.loadModel("data/models/tudner/phone.model");
ner.tag(inputText);
```

Code Listing 3.8: Tagging a text using a named entity recognizer.

In order to find out which tags were trained for the model you can create a meta file with one tag name per line, give it the same name as the model adding the suffix "_meta" and save it as a text file next to the model. Listing 3.9 shows how you can get a list of tags that a model can apply.

```
// create the tagger
NamedEntityRecognizer ner = new TUDNER();

// the meta file must be called phone_meta.txt
List<String> tags = ner.getModelTags("data/models/tudner/phone.model");

// print the tags to the console
CollectionHelper.print(tags);
```

Code Listing 3.9: Reading a model's meta data.

Training a Tagger

Some of the available taggers come with trained models that can be used to detect entities. However, the concepts they were trained for might not be sufficient for the application you have in mind, therefore, you need to create your own training set with entities from your domain and learn a tagger using that data. This section explains which tagging formats are available and how models for the available taggers can be trained.

First we start with the tagging formats which can be used to create a training set.

Tagging Formats Many tagging types have been developed which makes interoperability between different systems harder. Palladian can handle many of these formats and is able to transform a tagged text within these formats.

The different tagging are explained in this section using the same example text: "Bill Gates founded Microsoft in 1975.".

Slash The slash notation tags a given text with a slash and the tag right after the word. The sample text in the slash notation would look as show below.

```
Bill/PERSON Gates/PERSON founded Microsoft/ORG in 1975/DATE.
```

Bracket The bracket notation wraps the word in brackets. The sample tags would look as shown below.

```
[Bill Gates PERSON] founded [Microsoft ORG] in [1975 DATE].
```

Column (BIO) The column notation is one of the most widely used. For example the CoNLL NER dataset uses this format. The text is tokenized and on each line of the tagged text there is only one token (word) with its related tag separated by a space or tab. The BIO format means **B**eginning, cont **I**nue, and **O**utside. The BIO format can help to distinguish entities of the same type that are written close together.

The sample text in the column format would look as shown below. The "O" tag stands for "outside", that is no matching tag has been found.

```
Bill PERSON
Gates PERSON
founded O
Microsoft ORG
in O
1975 DATE
. O
```

The sample text in the column BIO format would look as shown below.

```
Bill B-PERSON
Gates I-PERSON
founded O
Microsoft B-ORG
in O
1975 B-DATE
. O
```

If we had another person name right after the first one, our tagger might think that it was only one person if we use the column notation. In the column BIO notation, both persons could be distinguished because the second one would start with B-PERSON.

XML The XML notation is very common too and is the favored one in Palladian. Similarly to the brackets notation, the XML notation wraps the word in an XML tag. The sample tags would look as shown below.

```
<PERSON>Bill Gates</PERSON> founded <ORG>Microsoft</ORG> in <DATE>1975</DATE>.
```

List The list notation is a separate file which contains references to the marked entities in the text. For each marked entity it contains its name, its start and stop index, and its tag. The sample list file for the sample text would look as shown below.

```
Bill Gates, 0-10, PERSON
Microsoft, 19-28, ORG
1975, 32-36, DATE
```

The FileFormatParser can transform the formats back and forth as shown in Figure 3.5. As you can see, we can transform every format to the XML notation which will be used to create an annotation list for the tagged text.

Using the training set Once you have created your training set, you should use the FileFormatParser to transform it into a tab separated column tagging format (if it isn't in this format already).

It is not possible to train a model for the web-based taggers Alchemy and OpenCalais. All other taggers can be trained using the train method, that requires the training file and a model configuration parameter which is different for each tagger.

Let us assume we want to train a model for each trainable tagger to recognize mobile phone names. You should have tagged training data in column separated form as shown below. Usually, the more training data you have, the more stable the trained model. This training data is stored in a tsv file called "phoneTraining.tsv".

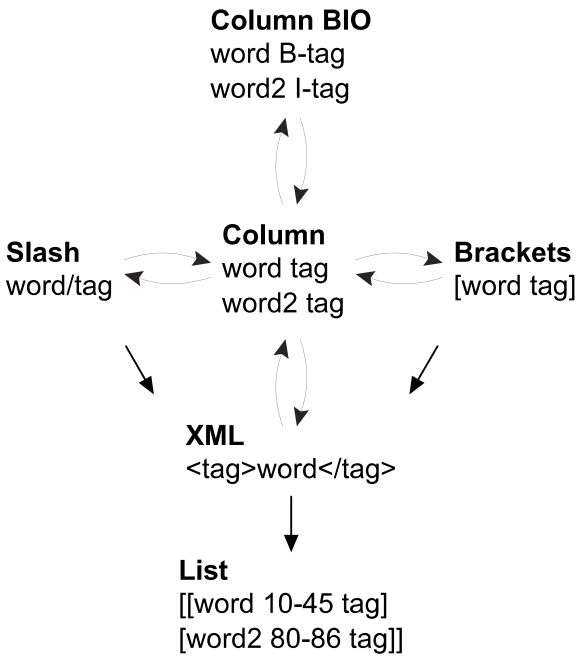

Figure 3.5: Possible transformation of tagging formats.

```
The O
iphone PHONE
4 PHONE
sells O
very O
well O
. O
```

Each tagger can be trained as shown in Listing 3.10. The only parameter which is used differently for each tagger is the "modelConfig". Table 3.4 shows what that parameter means for each trainable recognizer.

```
1 NamedEntityRecognizer ner = new StanfordNER();
2 boolean successful = ner.train("phoneTraining.tsv", modelConfig);
3 String output = "successful";
4 if (!successful) {output = "not successful";}
5 System.out.println("Training of "+ner.getName()+" was "+output);
```

Code Listing 3.10: Train a named entity recognizer.

Evaluating a Tagger

To see how good a trained tagger performs, we need test data which is tagged in the same manner as the training data. There are different versions of how to evaluate the performance. Let's go through them using an example. For example, let us assume that a human expert created the following markup [16]:

NER	Parameter meaning	Example
IllinoisLbjNER	Path to a configuration file, in this file you can specify which features and parameters should be used and where the trained model should be saved	.../illinoisner/baselineFeatures.config
JulieNER	Output path for the trained model. You can add a further parameter with the path to a configuration file, in this file you can specify which features and parameters should be used	.../juliener/phoneModel.mod and [.../juliener/tutorial/featconfig.conf]
LingPipeNER	Output path for the trained model	.../lingpipe/phone.model
OpenNLPNER	Output path for the trained model, the filename must follow the format "openNLP_TAG.bin.gz" since you can only train one tag per model	.../opennlp/openNLP_phone.bin.gz
StanfordNER	Path to a configuration file, in this file you can specify which features and parameters should be used and where the trained model should be saved	.../lingpipe/training/austen.prop
TUDNER	Output path for the trained model	.../tudner/phoneModel.model

Table 3.4: Configuration parameters to train named entity recognizers.

```
Unlike <PERSON>Robert</PERSON>, <PERSON>John Briggs Jr</PERSON> contacted
<ORG>Wonderful Stockbrockers Inc</ORG> in <LOCATION>New York</LOCATION>
and instructed them to sell all his shares in <ORG>Acme</ORG>.
```

Furthermore, let us assume that a NER system created the following markup [16] for the same text:

```
<LOCATION>Unlike</LOCATION> Robert, <ORG>John Briggs Jr</ORG> contacted
Wonderful <ORG>Stockbrockers</ORG> Inc <DATE>in New York</DATE> and
instructed them to sell all his shares in <ORG>Acme</ORG>.
```

The only correct match between the correct solution and the NER system output is `<ORG>Acme</ORG>`, all other markups are some kind of errors.

Error Types In classification tasks it is often possible to determine the true positives, false positives etc. but in NER it can sometimes help to be more precise about the classes. For example, two false positives are not necessarily equally wrong. Consider a system that had to tag person names in text, and it tagged "A good start" and "Jim Carrey was" as persons. While the first occurrence is obviously totally wrong, the second has to be considered wrong too but the system only did not find the right hand boundary correctly and tagged the word "was" too. In the previous example we can see five different errors an NER system can make [11]. The errors are shown and explained in the Table 3.5 [16].
Due to the variety of combinations to use the error types, three main evaluation methods have evolved during the years.

Exact-Match Evaluation The exact-match evaluation is the most simple one and does not take the different error types into account. A correct assignment must have the boundaries and the classification correct. The final score for the NER system is a micro-averaged f-measure (MAF). The NER system from the example would get the following scores.

- Precision = Correct / Total Assigned = 1 / 5 = 20%
- Recall = Correct / Total Possible = 1 / 5 = 20%
- MAF = 20%

Correct Solution	System Output	Error
Unlike	<LOCATION>Unlike</LOCATION>	The system tagged an entity where there is none.
<PERSON>Robert</PERSON>	Robert	The system missed to tag an entity.
<PERSON>John Briggs Jr </PERSON>	<ORG>John Briggs Jr</ORG>	The system tagged the entity but classified it incorrectly.
<ORG>Wonderful Stockbrockers Inc</ORG>	<ORG>Stockbrockers</ORG>	The system tagged the entity but the boundaries are wrong.
<LOCATION>New York </LOCATION>	<DATE>in New York</DATE>	The system found an entity but classified it incorrectly and got the boundaries wrong.

Table 3.5: NER features [16].

MUC Evaluation The MUC evaluation method takes all five errors from Table 3.5 into account and scores a system along two axis, the TYPE and the TEXT axis. If an entity was classified correctly (regardless of the boundaries) the TYPE is assigned correct. If an entity was found with the correct boundaries (regardless of its type) the TEXT is assigned correct. For both axis, three measures are kept: the number of possible entities (POS), the number of actual assigned entities by the system (ACT) and the number of correct answers by the system (COR). MUC als uses the MAF as the final score for the NER system. As the usual f-measure, also the micro-averaged f-measure is the harmonic mean between precision and recall. For the example we can calculate the MUC score for the system as follows.

- COR = 4 (2 times TYPE correct, 2 times TEXT correct)
- ACT = 10 (5 times TYPE assigned, 5 times TEXT assigned)
- POS = 10 (5 times TYPE, 5 times TEXT)
- Precision = COR / ACT = 4 / 10 = 40%
- Recall = COR / POS = 4 / 10 = 40%
- MAF = 40%

Listing 3.11 shows how we can programmatically evaluate a tagger.

```
1  // initialize the tagger
2  NamedEntityRecognizer ner = new TUDNER();
3
4  // the path to the test data
5  String testFilePath = "testData.xml";
6
7  // the path to the model that we want to evaluate
8  String modelPath = "data/models/tudner/phone.model";
9
10 // store the evaluation results in an object
11 EvaluationResult eResult = null;
12
13 // evaluate the model using the test data
14 eResult = ner.evaluate(testFilePath, modelPath, TaggingFormat.XML);
15
16 // print out the evaluation result
17 System.out.println(eResult);
18
19 // get interesting values in exact match and MUC mode
20 String r1 = "F1 Exact: " + eResult.getF1(EvaluationResult.EXACT_MATCH);
21 String r2 = "F1 MUC: " + eResult.getF1(EvaluationResult.MUC);
```

```
System.out.println(r1);
System.out.println(r2);
```

Code Listing 3.11: Evaluating the performance of a named entity tagger.

3.2.5 Web Page Age Detection

The age of a web page can often be a useful indicator about the freshness of the information. It might also be used in classification or ranking of web documents. Palladian is able to detect the age of many web pages by using four main techniques:

1. Reading the **HTTP header** of a web page and look for the date that the page has changed. Although this information is often absent of incorrect it sometimes is the only date we get.

2. Recognize a date in the **URL** of the web page. Especially blogs use the URL style which often reads similar to "http://domain.tld/posts/YYYY/MM/PostName". In these cases the date in the domain is a very good indicator of the web page's age.

3. Recognizing and ranking dates in the **structure and content** of the web page. Especially news pages start or end their news posts with the location and the date that the event they are reporting about happened. Since many dates might be found in the content it is necessary to rank them among each other. This is done using indicators such as the nearness to certain keywords such as "published" for example.

4. Searching for **inbound links from archives**. Some web pages can be found in archives with a date. If none of the other techniques returns valuable results it is possible to look up the URL in an archive, although the chances to find it are quite low.

Listing 3.12 shows how to detect the age of a web page. For more information on the algorithms used, see [34]. You can also check out the functionality of the web page age detection online at `http://www.webknox.com/wi#detectAge`.

```
// the URL of the page we want to know the age from
String url = "http://www.bbc.co.uk/news/world-europe-11432849";

// ExternalSearch is a boolean to switch on and off reference and
// archive techniques. Standard should be false (off).
boolean externalSearch = false;

// get the highest ranked date for this web page
ExtractedDate date = WebPageDateEvaluator
                                .getBestRatedDate(url, externalSearch);

// print the extracted date (should be 29.09.2010)
System.out.println(date);
```

Code Listing 3.12: Detecting the age of a web page.

3.3 Retrieval

3.3.1 Web Searcher

The `WebSearcher` is a class that can query a number of sources such as search engines and web pages with terms and retrieve matching results.

Basic Features

Basic features are:

- Query the Google search engine (unlimited queries, top 64 results only).
- Query Bing search engine (unlimited)
- Query Hakia search engine.
- Query Twitter.
- Query Google Blog search.

Some of the search APIs require API keys which can be specified in the config/palladian.properties file or set programmatically.

How To

The following code snippet shows how to initialize the `WebSearcher` and get a list of (English) URLs from the Bing search engine for the exact search "Jim Carrey".

```
1  // create web searcher object
2  WebSearcher searcher = new WebSearcher();
3
4  // set maximum number of expected results
5  searcher.setResultCount(10);
6
7  // set search result language to english
8  searcher.setLanguage(LANGUAGE_ENGLISH);
9
10 // set the query source to the Bing search engine
11 searcher.setSource(WebSearcherManager.BING);
12
13 // search for "Jim Carrey" in exact match mode (second parameter = true)
14 List<String> resultURLs = searcher.getURLs("Jim Carrey", true);
15
16 // print the results
17 CollectionHelper.print(resultURLs);
```

Code Listing 3.13: Retrieving result URLs from a search engine.

3.3.2 Document Retriever

The *DocumentRetriever* can be used do download documents from the WWW or create W3C documents from (X)HTML files from the hard disk.

Basic Features

Basic functionalities include:

- Download and save contents of a web page.
- Download documents in parallel using threads.
- Use `DownloadFilter` to specify which contents are allowed.
- Switch proxies after a certain number of requests to avoid being blocked

How To

The following code shows how to instantiate a simple `DocumentRetriever`.

```
// create the object
DocumentRetriever retriever = new DocumentRetriever();

// download and save a web page including their headers in a gzipped file
retriever.downloadAndSave("http://cinefreaks.com", "page.gz", true);

// create a retriever that is triggered for every retrieved page
RetrieverCallback crawlerCallback = new RetrieverCallback() {
     @Override
     public void onFinishRetrieval(Document document) {
          // do something with the page
          LOGGER.info(document.getDocumentURI());
     }
};
retriever.addRetrieverCallback(crawlerCallback);

// set the maximum number of threads to 10
retriever.setMaxThreads(10);

// the retriever should automatically use different proxies
// after every 3rd request (default is no proxy switching)
retriever.setSwitchProxyRequests(3);

 // set a list of proxies to choose from
List<String> proxyList = new ArrayList<String>();
proxyList.add("83.244.106.73:8080");
proxyList.add("83.244.106.73:80");
proxyList.add("67.159.31.22:8080");
retriever.setProxyList(proxyList);

// give the retriever a list of URLs to download
retriever.add("http://www.cinefreaks.com");
retriever.add("http://www.imdb.com");

// download documents
Set<Document> documents = retriever.start();
CollectionHelper.print(documents);

// or just get one document
Document webPage = retriever.getWebDocument("http://www.cinefreaks.com");
LOGGER.info(webPage.getDocumentURI());
```

Code Listing 3.14: Using the web crawler.

3.3.3 Web Crawler

The web crawler can be used to crawl domains or the web in general.

Basic Features

Basic functionalities include:

- Automatically crawl in- and/or outbound links from web pages.
- Use URL rules for the crawling process.

How To

The following code shows how to instantiate a simple crawler that starts at http://www.dmoz.org and follows all in- and outbound links. The URL of each crawled page is printed to the screen. The crawler will use 10 threads, changes the proxy after every third request and stops after having crawled 1000 pages. Instead of setting the parameters using the code, we can also specify them in the config/palladian.properties file.

```
// create the crawler object
Crawler crawler = new Crawler();

// create a callback that is triggered for every crawled page
RetrieverCallback crawlerCallback = new RetrieverCallback() {
    @Override
    public void onFinishRetrieval(Document document) {
        LOGGER.info("downloaded the page " + document.getDocumentURI());
    }
};
crawler.addCrawlerCallback(crawlerCallback);

// stop after 1000 pages have been crawled (default is unlimited)
crawler.setStopCount(1000);

// set the maximum number of threads to 1
crawler.setMaxThreads(1);

// the crawler should automatically use different proxies after every
// 3rd request (default is no proxy switching)
crawler.getDocumentRetriever().setSwitchProxyRequests(3);

// set a list of proxies to choose from
List<String> proxyList = new ArrayList<String>();
proxyList.add("83.244.106.73:8080");
proxyList.add("83.244.106.73:80");
proxyList.add("67.159.31.22:8080");
crawler.getDocumentRetriever().setProxyList(proxyList);

// start the crawling process from a certain page, true = follow links
// within the start domain, true = follow outgoing links
crawler.startCrawl("http://www.dmoz.org/", true, true);
```

Code Listing 3.15: Using the web crawler.

3.3.4 Web Feeds

Palladian's `ws.palladian.web.feeds` package offers various functionalities for tasks related to RSS and Atom web feeds. The most important classes include:

- `FeedDownloader` can be used for retrieving web feeds. The class basically wraps ROME library [45] for feed parsing but adds some additional techniques and fallbacks for parsing non-standard and malformed feeds, which ROME normally cannot parse.
- `FeedReader` is responsible for continuously aggregating web feeds using various different algorithms to predict feed's update behaviours.
- `FeedDiscovery` allows searching for web feeds using standard web search engines like Yahoo! and the so called "autodiscovery" feature[7], to detect feeds on web pages.

[7] `http://diveintomark.org/archives/2003/12/19/atom-autodiscovery` (also works for RSS)

- `FeedDatabase` implements the feed's package persistence layer using a relational database. The necessary MySQL database schema can be found in `config/feedsDbSchema.sql`[8].
- `FeedImporter` is used for adding new web feeds to the database.
- The class `Feed` and its associated `FeedItems` model a web feed's data structure.

Code Listing 3.16 gives a minimal sample use case employing the described classes.

```
1  // search feeds for "Porsche 911"
2  FeedDiscovery feedDiscovery = new FeedDiscovery();
3  feedDiscovery.setSearchEngine(SourceRetrieverManager.YAHOO_BOSS);
4  feedDiscovery.addQuery("Porsche 911");
5  feedDiscovery.setResultLimit(100);
6  feedDiscovery.findFeeds();
7  Collection<String> feedUrls = feedDiscovery.getFeeds();
8  CollectionHelper.print(feedUrls);
9
10 // download a feed
11 FeedDownloader feedDownloader = new FeedDownloader();
12 Feed feed = feedDownloader.getFeed("http://rss.cnn.com/rss/edition.rss");
13 List<FeedItem> feedItems = feed.getItems();
14 CollectionHelper.print(feedItems);
15
16 // initialize the FeedDatabase for storing the data
17 FeedStore feedStore = new FeedDatabase();
18
19 // add some feed URLs to the database
20 FeedImporter feedImporter = new FeedImporter(feedStore);
21 feedImporter.addFeeds(feedUrls);
22
23 // start aggregating news for the feeds in the database
24 // (this is an infinite loop)
25 FeedReader feedReader = new FeedReader(feedStore);
26 feedReader.aggregate(false);
```

Code Listing 3.16: Using the `feeds` package.

3.3.5 Wiktionary Database

Palladian offers a simple interface to the Wiktionary[9] data. So far, there are two databases available, one in German and one in English. The database contains information about each word. This information is in particular:

- Language, the language of the word. For example, English, German, Spanish...
- Type, the type of the word. For example, Noun, Adjective, Verb...
- Synonyms, words which have a similar meaning. For example, notebook = laptop.
- Hypernyms, words which are ancestrally related to the word. For example, notebook =¿ computer. Hypernyms are only in the German DB since the English Wiktionary does not provide them.

Code Listing 3.17 shows how to access the database. We create the WordDB with the path to the stored database. Then we can query the database with a string and in case the word is found we get a Word object holding basic information such as language and type. We can then call `aggregateInforamtion(word)` to find synonyms and hypernyms belonging to the word.

[8] You can use phphMyAdmin or MySQL command line utilities to import the schema into your database.
[9] http://www.wiktionary.org/

```
// load a word DB
WordDB wordDB = new WordDB("data/models/wordDatabaseEnglish/");

// you can load the database into the memory for faster read access
// wordDB.loadDbToMemory();

// search a word in the dtabase
Word word = wordDB.getWord("freedom");
LOGGER.info(word);

// find synonmys and hypernyms for the word
wordDB.aggregateInformation(word);
LOGGER.info(word);
```

Code Listing 3.17: Using the interface to the Wiktionary WordDB.

Other languages from the Wiktionary project can be parsed after extending the WiktionaryParser. The Wiktionary dumps can be found at `http://dumps.wikimedia.org/backup-index.html`.

3.3.6 MediaWiki Crawler

The MediaWiki crawler can be used to retrieve content and metadata from Wiki pages. This component is not really a crawler since it uses the MediaWiki API [1] to get all information, anyhow, we call it a crawler. To communicate with the API, the crawler is based on the Java Wiki Bot Framework (jwbf) [2].

Basic Features

Basic functionalities include:

- Download the content of a complete wiki.
- Store content in relational data base.
- Select namespaces to crawl.
- Access login protected Wikis.
- For a single page, extract:
  - page title,
  - HTML content of the most recent version, rendered by wiki (without navigation bar on left side),
  - complete history, for every revision: revisionID, timestamp of modification and author,
  - all links to other pages within the same wiki.
- Contineous crawling, i.e. keep data base up to date.
- Crawl multiple Wikis in parallel.
- Notification to process new pages in your own code.

Architecture

Figure 3.6 shows the architecture of the MediaWiki crawler. The classes *MediaWikiCrawler* and *PageConsumer* are the central classes, holding the control flow for fetching pages from the wiki (MediaWikiCrawlerfrom) and processing the extracted data (PageConsumer), for details see 3.3.6. Package *data* provides classes to temporary store fetched pages and their metadata while package *persistence* is used as a persistance layer to store extracted data in a database and load the crawler configuration. Package *queries* extends some queries of the Java Wiki Bot Framework (jwbf). All communication with the MediaWiki API is done via jwbf.

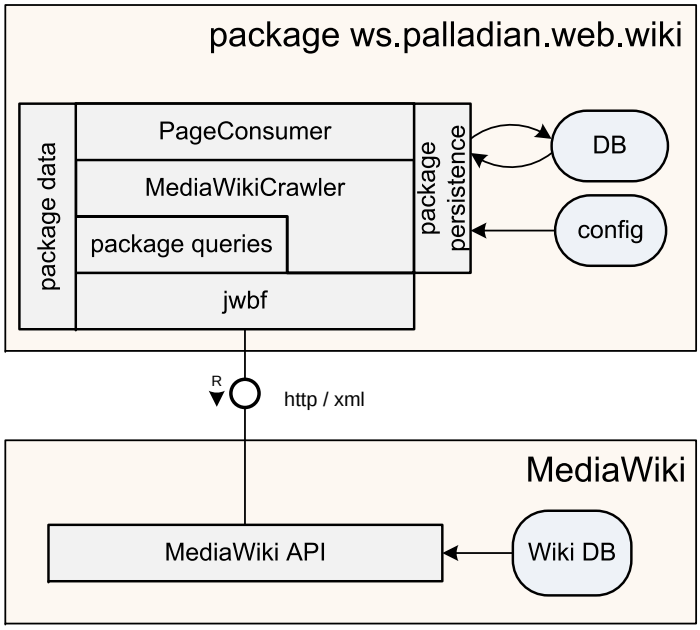

Figure 3.6: MediaWiki crawler architecture.

How To

The necessary MySQL database schema can be found in `config/mwCrawlerDbSchema.sql`, the crawler's configuration can be found in `config/mwCrawlerConfiguration.yml`, see 3.18. For every Wiki to crawl, at least *wikiName* and *wikiURL* are required.

```
# sample
    # Unique name of this wiki, like "Wikipedia (en)". Required parameter.
- wikiName: "Wikipedia (en)"
    # Path to the Wiki, like "http://en.wikipedia.org/". Required parameter.
  wikiURL: "http://en.wikipedia.org/"
    # Path to api.php, relative from wikiURL, like "/w/" for wikipedia. Must
        not contain the file name api.php.
    # Do not forget the "/" in  the end of the path if there is any.
    # Optional parameter. Leave empty if unknown.
  pathToAPI: "/w/"
    # Path to wiki pages, relative from wiki_url, like /wiki/ as used in
       wikipedia (resulting path is http://de.wikipedia.org/wiki/).
    # Do not forget the "/" in  the end of the path if there is any.
    # Optional parameter. Leave empty if unknown.
  pathToContent: "/wiki/"
    # User name to use for crawling.
    # Optional parameter.
  crawlerUserName: "myname"
    # The user's password. Optional parameter.
  crawlerPassword: "mysecret"
    # The namespaceIDs to include into crawling, separated by comma ",". All
        pages within these namespaces are crawled.
    # namespaceIDs that do not exist in the Wiki are ignored.
    # known bug: to crawl a single namespace, add comma like "1,", otherwise
        all namespaces are crawled!
    # Optional parameter. Leave empty to crawl all namespaces.
  namespacesToCrawl: "0, 1, 12"
```

Code Listing 3.18: MediaWiki crawler configuration.

To run the MediaWiki crawler, do the following three steps:

1. Create the required database tables by executing mwCrawlerDbSchema.sql.
2. Change the configuration mwCrawlerConfiguration.yml to setup all Wikis to crawl.
3. Start the crawler like in listing 3.19.

```
1 final int queueCapacity = 1000;
2 final int pageConsumers = 5;
3
4 LinkedBlockingQueue<WikiPage> pageQueue = new LinkedBlockingQueue<WikiPage>(
      queueCapacity);
5 MWConfigLoader.initialize(pageQueue);
6 for (int i = 1; i <= pageConsumers; i++) {
7   Thread consumer = new Thread(new PageConsumer(pageQueue), "Consum-" + i);
8   consumer.start();
9 }
```

Code Listing 3.19: MediaWiki crawler initialization.

Understand and extend the crawler

As already introduced, the crawler has two central classes—MediaWikiCrawler and PageConsumer—to control the work flow. For every wiki to crawl one MediaWikiCrawler is created, running as a thread, "producing" *WikiPage*-objects when fetching a new page or revision and adding them to a central *pageQueue*. PageConsumer-threads wait for new pages and are used to process the extracted data. The current PageConsumer implementation contains basic functionality only, it has to be extended (inheritance). Override *consume(WikiPage page)* to do something useful with every page that is taken from the pageQueue.

The MediaWikiCrawler itself should not be changed unless more information (e.g. the page content of every revision) is required. Figure 3.7 shows the state machine of the crawler. The initialization is done by *MWConfigLoader* which loads the configuration from data/mwCrawlerConfiguration.yml, updates the configuration in the database and starts one MediaWikiCrawler-thread per Wiki to crawl. Every MediaWikiCrawler does the following:

first start: If the crawler is started the first time, it gets all namespaces from the API and stores them in the database. Depending on the parameter *namespacesToCrawl* in mwCrawlerConfiguration.yml, all or the selected namespaces are marked for crawling. The next step is to fetch the titles of all page from the API. Next, for each title, the page content, Wiki-links to other pages in the same Wiki and all revisions are fetched from the API. As soon as a page has been processed completely, it is put to the pageQueue to be processed by one of the PageConsumers. When done, the crawler writes the current time to database and enters a continuous crawling mode. Caution: On startup, every crawler checks whether this Wiki has been crawled completely before. If not, it deletes all page-reated data from database.

wiki already crawled: If the wiki has already been crawled completely, the crawler enters the continuous crawling mode. It wakes up periodically and checks the Wiki for new pages and new revisions of existing pages. Every new/updated page is added to the pageQueue.

3.4 Preprocessing

3.4.1 Tokenization

A *Token* is a sequence of characters that can be categorized according to the tokenization rules. *Tokenization* is the process of transforming a text into a sequence of tokens. Table 3.6 shows example token types. The text "Today I lost $1000 dollar playing poker." could consists of 8 tokens for example. What token types exist depend on the application. The dollar sign could be a single token for example. Palladian uses regular expressions to perform the tokenization.

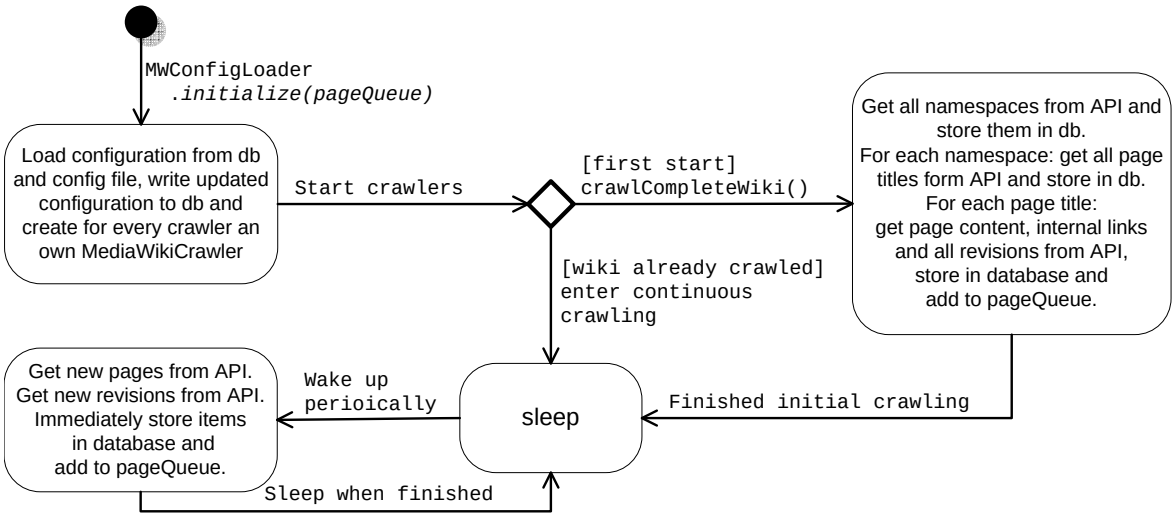

Figure 3.7: MediaWiki crawler UML state machine.

Character sequence	Token type
many	word
$1000	amount of money
26.09.2010	date
.	punctuation

Table 3.6: Example types of tokens.

One can also see each sentence of a text as a token, in this case the tokenization process is called sentence splitting as explained int he next section.

3.4.2 Sentence Splitting

Palladian has a rudimentary implementation for the common need for sentence splitting. Palladian implementation works with hand-crafted rules and thus does not require a model. Sentences try to be splitted on periods, question marks, and exclamation marks but there are also rules that try to prevent splitting sentences at ellipses. For example, the following is online one sentence although it contains several periods: “Sometimes sentenes contain many periods...really!”.

The following code shows how the sentence splitting can be used:

```
1 String inputText = "This is a sentence. This is another one!";
2 List<String> sentences = Tokenizer.getSentences(inputText);
3 CollectionHelper.print(sentences);
4 // prints:
5 // This is a sentence
6 // This is another one!
```

Code Listing 3.20: Using the sentence splitter.

You can also get a specific sentence by providing a phrase that is part of the sentence using the *getSentence* method.

3.4.3 Creating N-Grams

N-grams are sets of tokens of the length n. We can distinguish two main types of n-grams:

1. **Character level n-grams** use each character of the string as a token. For example, from the string "It is sunny" we can create the following set of 3-grams: $it, ti, is, is, ss, su, sun, unn, nny$. The number of n-grams in a set can be calculated as $ngrams = numberOfTokens - n + 1$. In our example that means $9 = 11 - 3 + 1$.

2. **Word level n-grams** use each word (separated with white space) of the string as a token. For example, from the string "It is so nice and sunny today" we can create the following set of 3-grams: $Itisso, issonice, soniceand, niceandsunny, andsunnytoday$. The number of n-grams in a set can be calculated as $ngrams = numberOfTokens - n + 1$. In our example that means $5 = 7 - 3 + 1$. If you want to have single words as features for the text classifier you can simply use unigrams or bigrams which are n-grams with $n = 1$ or $n = 2$ respectively.

The document preprocessor allows you to create a set of n-grams with different length too. For example, you can create all 2-grams, 3-grams, and 4-grams for the given input text.

Listing 3.21 shows how you can create n-grams from a given text.

```
// the input text that we want to separate into n-grams
String inputText = "a cat runs funnily";

// store a set of n-grams
Set<String> ngrams = null;

// calculate all character n-grams of length 3
ngrams = Tokenizer.calculateCharNGrams(inputText,3);

// calculate all word n-grams of length 3
ngrams = Tokenizer.calculateWordNGrams(inputText,3);

// calculate all character level n-grams of length 3 to 5
ngrams = Tokenizer.calculateAllCharNGrams(inputText,3,5);

// calculate all character word n-grams of length 1 to 3
ngrams = Tokenizer.calculateAllWordNGrams(inputText,1,3);
```

Code Listing 3.21: Creating n-grams.

3.4.4 Noun Pluralization and Singularization

Palladian is able to transform most English singular nouns to their plural and back. For example, "city" becomes "cities" and "index" becomes "indices".

Listing 3.22 shows the simple usage of the singularization and pluralization using the WordTransformer class.

```
String singular = "city";
String plural = "";
plural = WordTransformer.wordToPlural(singular);
singular = WordTransformer.wordToSingular(plural);
System.out.println(singular);
System.out.println(plural);
// prints:
// cities
// city
```

Code Listing 3.22: Transforming words from singular to plural and vice versa.

3.5 Miscellaneous

The toolkit contains many helper functionalities for reoccurring tasks in the `ws.palladian.helper` package. The following code snippet shows several sample usages of some of the functions.

```
// sort a map by its value in ascending order (2nd parameter = true)
Map m = CollectionHelper.sortByValue(map, true);

// reverse a list
List l = CollectionHelper.reverse(list);

// print the contents of a collection
CollectionHelper.print(collection);

// get the runtime of an algorithm and print it (2nd parameter = true)
long startTime = System.currentTimeMillis();
for (int i = 0; i < 10000; i++) {
        int c = i * 2;
}
DateHelper.getRuntime(t1, true);

// (de) serialization of objects
FileHelper.serialize(obj, "obj.ser");
Object obj = FileHelper.deserialize("obj.ser");

// rename, copy, move and delete files
FileHelper.rename(new File("a.txt"), "b.txt");
FileHelper.copyFile("src.txt", "dest.txt");
FileHelper.move(new File("src.txt"), "dest.txt");
FileHelper.delete("src.txt");

// get files from a folder
File[] files = FileHelper.getFiles("folder");

// zip and unzip a text
FileHelper.zip("text", "zipFile.zip");
String t = FileHelper.unzipFileToString("zipFile.zip");

// perform some action on every line of an ASCII file
final Object[] obj = new Object[1];
obj[0] = 1;

LineAction la = new LineAction(obj) {

    @Override
    public void performAction(String line, int lineNumber) {
        System.out.println(lineNumber + ": " + line + " " + obj[0]);
    }
}
FileHelper.performActionOnEveryLine(filePath, la);

// round a number with a number of digits
double r = MathHelper.round(2.3333, 2);

// remove HTML tags
String r = HTMLHelper.removeHTMLTags("<a>abc</a>",
                                     true, true
                                     true, true);

// trim a string
String t = StringHelper.trim(" _to trim++++");
```

```
57
58 // reverse a string
59 String r = StringHelper.reverse("abc");
60
61 // encode and decode base64
62 String e = StringHelper.encodeBase64("abc");
63 String d = StringHelper.decodeBase64(e);
```

Code Listing 3.23: Miscellaneous functions.

Chapter 4

Where to Go from Here?

If you can't find something that you need, there is a list of similar projects in Section 1.6 that you can scan through. If you still can't find it, research the topic, implement the code and commit it back to Palladian.

4.1 Referenced Libraries

Palladian makes excessive use of third party libraries. We do not intend to re-implement code but rather to built on it and create something superior. Here an incomplete list of libraries the toolkit uses:

- Apache Commons [6] for many standard tasks in string and number manipulation and more.
- iText [21] for creating PDF documents.
- Log4j [30] for logging.
- Lucene [32] for indexing and making learned models persistent.
- NekoHTML [39] to clean up the HTML of web pages in order to process them correctly.
- ROME [45] for parsing RSS and Atom feeds.
- SimMetrics [48] to calculate similarities of strings.
- Twitter4j [57] to query the Twitter API.
- Weka [19] for machine learning.

4.2 History

The foundation of Palladian's code came out of the WebKnox project[59] that was started in 2008. The code is in development by students of the Dresden University of Technology. Contributors are:

- Christopher Friedrich
- Martin Gregor
- Philipp Katz
- Klemens Muthmann
- Silvio Rabe
- Sandro Reichert

- Julien Schmehl
- David Urbansky
- Robert Willner
- Martin Werner
- Martin Wunderwald
- Stephan Zepezauer

Bibliography

[1] MediaWiki API. http://en.wikipedia.org/w/api.php, accessed February 21, 2011.

[2] Java Wiki Bot Framework, 2011. http://jwbf.sourceforge.net/, accessed February 21, 2011.

[3] http://www.alchemyapi.com/api/lang/, accessed November 12, 2010.

[4] http://www.alchemyapi.com/, accessed July 11, 2010.

[5] http://alias-i.com/lingpipe, accessed July 9, 2010.

[6] http://commons.apache.org/.

[7] J. Atserias, B. Casas, E. Comelles, M. González, L. Padró, and M. Padró. FreeLing 1.3: Syntactic and semantic services in an open-source NLP library. In *Proceedings of the 5th International Conference on Language Resources and Evaluation (LREC06)*, pages 48–55, 2006. http://www.lsi.upc.edu/~nlp/papers/atserias06.pdf.

[8] http://balie.sourceforge.net/.

[9] T. Briscoe, J. Carroll, and R. Watson. The second release of the RASP system. In *Proceedings of the COLING/ACL on Interactive presentation sessions*, pages 77–80. Association for Computational Linguistics, 2006.

[10] Chia-Hui Chang, Mohammed Kayed, MohebR Girgis, and Khaled Shaalan. A Survey of Web Information Extraction Systems. 2006.

[11] Chris Manning. Doing Named Entity Recognition? Don't optimize for F1. Website, Blog, August 2006.

[12] Christian Kohlschütter, Peter Fankhauser, and Wolfgang Nejdl. Boilerplate Detection using Shallow Text Features. In *Proceedings of the third ACM international conference on Web search and data mining*, 2010.

[13] W. Cohen. MinorThird: Methods for Identifying Names and Ontological Relations in Text using Heuristics for Inducing Regularities from Data. 1, 2004. cohen2004minorthird, accessed July 11, 2010.

[14] http://contentanalyst.com/html/tech/technologies.html, accessed July 11, 2010.

[15] D. Cunningham, D. Maynard, D. Bontcheva, and M. Tablan. GATE: A framework and graphical development environment for robust NLP tools and applications. 2002.

[16] David Nadeau. *Supervised Named Entity Recognition: Learning to Recognize 100 Entity Types with little Supervision.* PhD thesis, Ottawa-Carleton Institute for Computer Science, 2007.

[17] http://code.google.com/apis/language/translate/v2/using_rest.html, accessed November 12, 2010.

[18] U. Hahn, E. Buyko, R. Landefeld, M. M
"uhlhausen, M. Poprat, K. Tomanek, and J. Wermter. An overview of JCoRe, the JULIE lab UIMA component repository. In *Proceedings of the LREC*, volume 8, pages 1–7, 2008.

[19] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. Witten. The WEKA data mining software: An update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, 2009. `http://www.cs.waikato.ac.nz/~ml/index.html`.

[20] `http://l2r.cs.uiuc.edu/~cogcomp/software.php`.

[21] `http://itextpdf.com/`.

[22] Jenny Rose Finkel, Trond Grenager, and Christopher Manning. Incorporating Non-local Information into Information Extraction Systems. In *Proceedings of the 43nd Annual Meeting of the Association for Computational Linguistics (ACL 2005)*, pages 363–370, 2005. `http://nlp.stanford.edu/~manning/papers/gibbscrf3.pdf`.

[23] `http://www.jroller.com/melix/entry/nlp_in_java_a_language`, accessed November 12, 2010.

[24] `http://wt.jrc.it/lt/Acquis/`, accessed November 12, 2010.

[25] Kavi Mahesh, Oracle Corporation. Text Retrieval Quality: A Primer, 1999. `http://www.oracle.com/technetwork/database/enterprise-edition/imt-quality-092464.html`, accessed April 13, 2011.

[26] `http://www.languagecomputer.com/technology/`, accessed July 11, 2010.

[27] `http://www.lemurproject.org/`, accessed January 7, 2011.

[28] `http://carrotsearch.com/lingo3g-overview.html`, accessed July 11, 2010.

[29] H. Liu. MontyLingua: An end-to-end natural language processor with common sense, 2004. `http://web.media.mit.edu/~hugo/montylingua/`, accessed July 11, 2010.

[30] `http://logging.apache.org/log4j/`.

[31] E. Loper and S. Bird. NLTK: The natural language toolkit. In *Proceedings of the ACL-02 Workshop on Effective tools and methodologies for teaching natural language processing and computational linguistics*, volume 1, page 70. Association for Computational Linguistics, 2002. `http://www.nltk.org/`, accessed July 11, 2010.

[32] `http://lucene.apache.org`.

[33] C. D. Manning, P. Raghavan, and H. Schütze. *An Introduction to Information Retrieval*. Cambridge University Press, 2009.

[34] Martin Gregor. Altersbestimmung von Webseiten. Master's thesis, Dresden University of Technology, 2010.

[35] A. McCallum. Mallet: A MAchine Learning for LanguagE Toolkit, 2002. `http://mallet.cs.umass.edu/index.php`, accessed July 11, 2010.

[36] `http://morphadorner.northwestern.edu/`.

[37] `http://www.nactem.ac.uk/software.php`, accessed July 11, 2010.

[38] Natural Language Processing and Information Retrieval Group, Universidad Nacional de Educación a Distancia. DeliciousT140 Dataset. `http://nlp.uned.es/social-tagging/delicioust140/`, accessed October 8, 2010.

[39] `http://nekohtml.sourceforge.net/`.

[40] Nick Lothian. Classifier4J, 02 2005. `http://classifier4j.sourceforge.net/`, accessed March 06, 2011.

[41] Olena Medelyan, Eibe Frank, and Ian H. Witten. Human-competitive tagging using automatic keyphrase extraction. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing*, pages 1318–1327. Association for Computational Linguistics, 2009.

[42] `http://www.opencalais.com/`, accessed July 11, 2010.

[43] `http://opennlp.sourceforge.net/about.html`, accessed September 23, 2010.

[44] Philipp Katz. NewsSeecr – Clustering und Ranking von Nachrichten zu Named Entities aus Newsfeeds. Master's thesis, Dresden University of Technology, 2010.

[45] `https://rome.dev.java.net/`, accessed October 11, 2010.

[46] `http://www.basistech.com/products/`, accessed July 11, 2010.

[47] M. Settings. Apache Mahout-scalable machine learning algorithm. *health*, 2:67.

[48] `http://www.dcs.shef.ac.uk/~sam/simmetrics.html`.

[49] `http://snowball.tartarus.org/`, accessed October 8, 2010.

[50] Spence Koehler. The Semantic Discovery Toolkit. `http://code.google.com/p/semanticdiscoverytoolkit/`, accessed October 20, 2011.

[51] `http://nlp.stanford.edu/software/index.shtml`.

[52] J. Stefanowski and D. Weiss. Carrot 2 and language properties in web search results clustering. *Advances in Web Intelligence*, pages 955–955, 2003. `http://project.carrot2.org/`, accessed July 11, 2010.

[53] A. Stolcke. SRILM-an extensible language modeling toolkit. In *Seventh International Conference on Spoken Language Processing*, volume 3, pages 901–904. Citeseer, 2002. `http://www.speech.sri.com/projects/srilm/`, accessed July 11, 2010.

[54] Sujit Pal. jtmt – Java Text Mining Toolkit. `http://jtmt.sourceforge.net/`, accessed April 21, 2011.

[55] `http://www.megaputer.com/textanalyst.php`, accessed July 11, 2010.

[56] K. Tomanek, E. Buyko, and U. Hahn. An uima-based tool suite for semantic text processing. In *UIMA Workshop at the GLDV*, volume 11, 2007.

[57] `http://twitter4j.org/en/index.htm`.

[58] `http://www.textanalysis.com/Products/VisualText/visualtext.html`, accessed July 11, 2010.

[59] `http://www.webknox.com`.

[60] 9 2010. `http://www.webknox.com/blog/2010/09/named-entity-definition/`.

[61] Xiaoguang Qi and Brian D. Davison. Web page classification: Features and algorithms. 2009.

[62] X. Zhou, X. Zhang, and X. Hu. Dragon Toolkit: Incorporating auto-learned semantic knowledge into large-scale text retrieval and mining. In *Proceedings of the 19th IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*. Citeseer, 2007.