Discovering Information Relevant to API Elements Using Text Classification

Gayane Petrosyan

School of Computer Science
McGill University, Montreal

December 12, 2013

A thesis submitted to McGill University in partial fulfilment of the requirements of the degree of Master of Science.

Copyright ©Gayane Petrosyan; December 12, 2013

ACKNOWLEDGEMENTS

First of all I would like to express my gratitude to my supervisor Prof. Martin Robillard for his excellent guidance, patience, and demanding a high quality of work. I would also like to thank Prof. Renato De Mori for his expertise and numerous advice throughout my research. I also thank all SWEVO research group members: Christoph Treude, Annie Ying, Yam Chhetri, Mangala Gowri Krishnamoorthy, Gias Uddin for assistance in data collection and assistance. I thank Barthélémy Dagenais, a former SWEVO research group member, for his readiness to help me throughout my research with usability issues of his former research work - Recodoc. I would like to thank Maris Jukss and Ouais Alsharif for their help with data collection. I thank all SWEVO group members and Computation and Logic group members for the friendly atmosphere in the lab.

I would also like to thank Luys foundation for financial support, without which starting my master degree would not have been possible.

I thank my friends Hayk Grigoryan for numerous advice, insightful comments and review of my thesis, Hasmik Alvrtsyan and Vahe Musoyan for advice on statistics, and of course, people who always support and encourage me: my parents, my brother and the one who was always there, to Pierre-Luc Bacon.

ABSTRACT

With the growing size of Application Programming Interfaces (APIs), both API usability and API learning become more challenging. API learning resources are often crucial for helping developers learn an API, but they are distributed across different documents, which makes finding the necessary information more challenging.

This work focuses on discovering relevant sections of tutorials for a given API type. We approach this problem by identifying API types in an API tutorial, dividing the tutorial into small fragments and classifying them based on linguistic and structural features. The system we developed can ease information discovery for the developers who need information about a particular API type. Experiments conducted on five tutorials show that our approach is able to discover sections relevant to an API type with 0.79 average precision, 0.73 average recall, and 0.75 average F1 measure when trained and tested on the same tutorial. When trained on four tutorials and tested on a fifth tutorial the average precision is 0.84, average recall is 0.62, and the F1 measure is 0.71.

ABRÉGÉ

Avec la taille grandissante des interfaces de programmation (API), l'aptitude à l'utilisation ainsi que la facilité d'apprentissage deviennent des préoccupations de premier ordre. La disponibilité de ressources d'apprentissage des API est de grande importance pour parvenir à developer efficacement à partir de différentes sources de documentation.

Ce mémoire est consacré au problème de découverte automatique de sections pertinentes contenues dans les tutoriels des API. Nous traitons ce problème en commençant par l'identification du type d'API d'un tutoriel pour ensuite le diviser en fragments qui seront classés d'après leurs propriétés structurelles et linguistiques. Le système que nous avons développé rend le processus de découverte de sections de tutoriel beaucoup plus facile. Une évaluation de notre système a été réalisée avec cinq tutoriels et montre que notre approche peut découvrir des sections pertinentes avec une précision moyenne de 0.79, 0.73 en moyenne de rappel, et 0.75 de mesure moyenne F1 lorsque entraîné et testé pour le même tutoriel. Lorsqu'entraîné depuis quatre tutoriels et testé dans avec le cinquième, nous obtenons 0.84 de précision moyenne, 0.62 de moyenne de rappel, et finalement 0.71 de mesure F1

Contents

Contents vi

List of Tables viii

List of Figures 1

1 Introduction 2

2 Problem Description 5

2.1 Problem Formulation 5

2.2 Experimental Corpus 7

2.3 Background 8

2.4 Related Work 16

3 Preprocessing and Classification 20

3.1 Finding API Elements 20

3.2 Tutorial Segmentation 22

3.3 Relevance Classification 28

4 Annotating The Experimental Corpus 46

4.1 Annotation Tool 46

4.2 Annotation Process 47

4.3 Annotation Results 48

5 Classification Results and Experiments **53**

5.1 Classification Results 53

5.2 Results for Different Set of Features 55

5.3 Cross-Tutorial Testing 57

5.4 Learning Curves 58

5.5 IR Comparison 59

6 Conclusion and Future Work **62**

6.1 Conclusion 62

6.2 Future Work 63

Bibliography **65**

A Annotation Guide **71**

B Multi-word concepts **73**

C Tutorial Statistics **80**

List of Tables

2.1 Selected Tutorials for Study 8

3.1 Recodoc Results for Studied Tutorials 22

3.2 Segmentation Results for Studied Tutorials 25

3.3 Real-Valued Features 35

3.4 Tutorial Level Features 37

3.5 Section Level Features 38

3.6 Sentence Level Features 39

3.7 Dependency-based Features 41

3.8 A Few Examples of Dependencies 44

3.9 Comparative results for SVM and MaxEnt classifiers 45

4.1 Annotation Results 49

4.2 Tutorial Statistics 50

5.1 LOOCV Results for All Tutorials 53

5.2 Coverage Provided by Classification per API Element 55

5.3 Classification Results for Different Set of Features 56

5.4 Cross Tutorial Testing Results 58

5.5 MaxEnt vs. CosSim 61

C.1 JodaTime Statistics per Section 80

C.2 JodaTime Statistics per API type 81
C.3 Math Statistics per Section 82
C.4 Math Statistics per API type 84
C.5 Collections(official) Statistics per Section 87
C.6 Collections(official) Statistics per API type 89
C.7 Collections(Jenkov) Statistics per Section 91
C.8 Collections(Jenkov) Statistics per API type 94
C.9 Smack Statistics per Section 95
C.10 Smack Statistics per API type 97

List of Figures

1.1 Collections framework tutorial - Section “General-Purpose Deque Implementations” 3

2.1 Smack - Section “Chat” 6

3.1 Section Lengths for JodaTime API Tutorial 23

3.2 Number of API Types per Section 26

3.3 Number of Sections Mentioning API Type 27

3.4 Case 1: Before Modification 29

3.5 Case 1: After Modification 29

3.6 Case 2: Before Modification 30

3.7 Case 2: After Modification 30

3.8 A Section from Math Library Tutorial: Highlighted for `KalmanFilter` API Element 32

3.9 An Example from JodaTime API Tutorial 42

4.1 Annotation Tool 47

4.2 Number of Relevant API Types per Section 51

4.3 Number of Sections with Relevant API Type 52

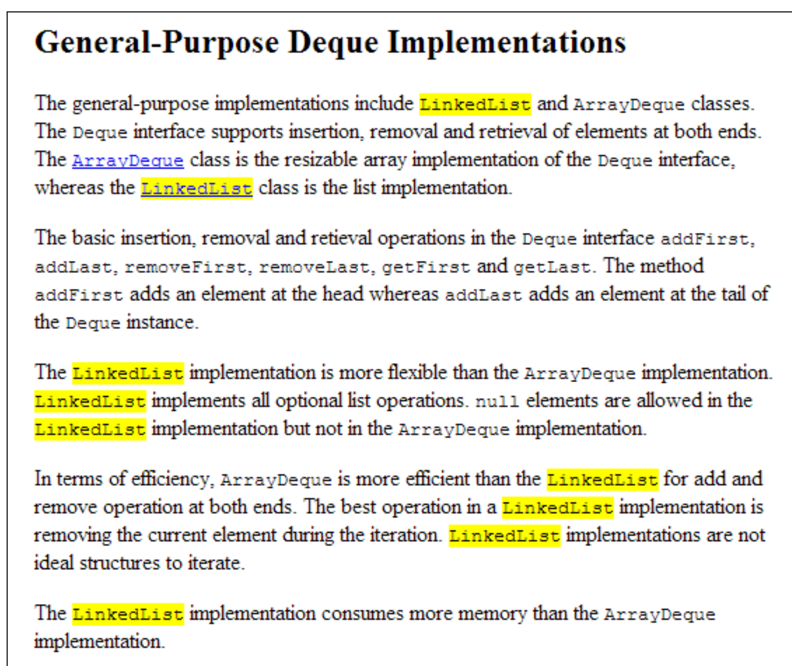
5.1 Learning Curves 60

1 Introduction

The number and the size of Application Programming Interfaces (APIs) are continuously growing. Applications become increasingly dependent on APIs. For example, Java SE 6, which is the core of all Java applications, contains 3774 classes and 203 packages. Programmers, both novice and experienced, are faced with the problem of learning about the vast number of APIs. Given the time at their disposal, it is not possible for programmers to learn all the APIs they need in depth.

Numerous studies of API usability identified the critical role of good API usage examples. As identified by Robillard [28], out of 80 Microsoft developers, 78% mentioned that they refer to API documentation and 55% to code examples for learning API. However, API reference documentation can be long and overwhelming. Javadocs and .Net documentations contain obvious content in 43% and 51% of randomly sampled API type documentations respectively [16]. The second most popular learning resource for API identified by Robillard [28] were code examples. The lack of provided API resources, especially good examples, is one of the main obstacles to learning APIs. According to participants, the main roles of code examples are providing “best practices”, informing about the design of the API, providing rationale, and confirming programmers’ intuition about how things work. Though code examples are one of the key learning resources and play a significant role in understanding and using APIs, they will not be that useful without any description. Nasehi et al. [22] studied a different characteristics of a good code example. One of the outcomes of that study is that good code examples should be accompanied with some description of the code.

API tutorials are API learning resources that combine both descriptive text and code examples. Tutorials usually are organized as a sequence of API usage examples, where each section addresses a solution for a particular programming task. However, browsing the table of content of an API tutorial might not help to find useful information about a particular API element. First of all, as API tutorials are task based, titles usually describe the task (e.g. “Error Handling” is a title in Apache.Commons.Math library tutorial, and “Input and Output” in Joda-Time API user guide). Second, the titles of tutorials sometimes are not informative at all, such as “Next”, “Example”, “Overview”, “General case”, etc. Finally, a part of a tutorial might contain useful information about an API element even if the section is not specifically about this API element.



General-Purpose Deque Implementations

The general-purpose implementations include `LinkedList` and `ArrayDeque` classes. The `Deque` interface supports insertion, removal and retrieval of elements at both ends. The `ArrayDeque` class is the resizable array implementation of the `Deque` interface, whereas the `LinkedList` class is the list implementation.

The basic insertion, removal and retieval operations in the `Deque` interface `addFirst`, `addLast`, `removeFirst`, `removeLast`, `getFirst` and `getLast`. The method `addFirst` adds an element at the head whereas `addLast` adds an element at the tail of the `Deque` instance.

The `LinkedList` implementation is more flexible than the `ArrayDeque` implementation. `LinkedList` implements all optional list operations. `null` elements are allowed in the `LinkedList` implementation but not in the `ArrayDeque` implementation.

In terms of efficiency, `ArrayDeque` is more efficient than the `LinkedList` for add and remove operation at both ends. The best operation in a `LinkedList` implementation is removing the current element during the iteration. `LinkedList` implementations are not ideal structures to iterate.

The `LinkedList` implementation consumes more memory than the `ArrayDeque` implementation.

Figure 1.1: Collections framework tutorial - Section “General-Purpose Deque Implementations”

For example, Figure 1.1 shows a section from the Collections framework tutorial within the official Java Tutorials which is titled “General-Purpose Deque Implementations” and is mainly about different implementations of the `Deque` interface. The section compares `ArrayDeque` and `LinkedList` implementations and mentions useful information for both. However, a user browsing for `LinkedList` would hardly access a section titled “General-

Purpose Deque Implementations". Searching would not help either because we observed that in over 50% of cases a section is not directly relevant to an API element that is mentioned within it.

The other alternative to browsing API tutorials is using search engines. General search engines are not designed to support programmers in their tasks. Although search engines are used a lot by programmers, usually programmers need to open the search results to assess the relevance of a page for their task [29]. Stylos and Myers [29] also note that the assessment of the relevance of the result page depends on the expertise of the programmer. The less experienced are the programmers, the harder it is for them to assess the relevance of the result page. Hoffmann et al. [13] also stressed the importance of the search engines but mentioned that useful information can be distributed across different search results. Both Stylos, Myers [29] and Hoffmann et al. [13] proposed search tools for programmers. In both cases, the search tool was focused mostly on code examples and the code keywords for retrieving relevant information. In contrast, the current work will mainly concentrate on the descriptive text as we believe that text contains information for distinguishing less obvious relevant cases.

In this work we propose a technique for discovering relevant sections of an API tutorial to help programmers find additional related information about API elements in which they are interested. The suggested technique automatically analyses API tutorials to assess the usefulness of the information contained therein using Natural Language Processing and Text Classification methods.

The remainder of this thesis is organized as follows. In Chapter 2 we present a detailed formulation of the problem, experimental corpus, introduction to basic concepts used in the thesis and discussion of related work. In Chapter 3 we present the data preprocessing and detailed solution description. In Chapter 4 we present the methodology and the results of data preparation for experiments. In Chapter 5 we present different experiments and results for the suggested technique. Finally, in Chapter 6 we conclude with the summary of the results and future work.

2

Problem Description

The main problem of the current work is the identification of sections relevant to an API element. An *API element* is generally any member of an API such as a class, a method, a field, etc. For the current work we limit the problem to classes only. The motivation of this is that a single section of an API tutorial usually describes a solution for a programming task by using a set of methods or classes. For example, in the Chat section (Figure 2.1) of the Smack API tutorial, the `Chat.sendMessage()` and `Chat.createMessage()` methods are mentioned. However, none of them separately describes how to create a chat, which is the main task of the section.

We say a section is *relevant* to an API type if it would help a reader unfamiliar with the corresponding API to decide when or how to use the API type to complete a programming task. A section might not be solely about the API type and the API type might not be fully described in the section, but the section might still be considered as relevant.

In the remainder of the chapter, we present the more detailed description of the problem, necessary background for the solution and related work.

2.1 Problem Formulation

For solving the problem of identifying tutorial sections relevant to an API type, we divide the problem into three main sub-problems.

First, for recommending relevant tutorial sections for an API type we decided to consider

Chat

A chat creates a new thread of messages (using a thread ID) between two users. The following code snippet demonstrates how to create a new Chat with a user and then send them a text message:

```
// Assume we've created a Connection name "connection".
ChatManager chatmanager = connection.getChatManager();
Chat newChat = chatmanager.createChat("jsmith@jivesoftware.com", new MessageList
    public void processMessage(Chat chat, Message message) {
        System.out.println("Received message: " + message);
    }
});

try {
    newChat.sendMessage("Howdy!");
}
catch (XMPPException e) {
    System.out.println("Error Delivering block");
}
```

The `Chat.sendMessage(String)` method is a convenience method that creates a Message object, sets the body using the String parameter, then sends the message. In the case that you wish to set additional values on a Message before sending it, use the `Chat.createMessage()` and `Chat.sendMessage(Message)` methods, as in the following code snippet:

```
Message newMessage = new Message();
newMessage.setBody("Howdy!");
message.setProperty("favoriteColor", "red");
newChat.sendMessage(newMessage);
```

Figure 2.1: Smack - Section "Chat"

only API types that are explicitly mentioned in the text of the tutorial. It can be challenging to find where in the text of the tutorial API types are mentioned. API types are not always mentioned using specific HTML tags. For example, titles which mention API types rarely use special HTML tags for marking it as such. Besides, API types can have a name which is also a word in natural language. Even after identifying that a certain word is a code element it still needs to be disambiguated, which means to map it to the exact API type to which it refers. For example, if the found word is `RealVector`, then it should be mapped to `org.apache.commons.math3.linear.RealVector`. The more challenging example is the word `Date`, which should be mapped to `java.util.Date` or `java.sql.Date` as appropriate.

Second, we need to split an API tutorial into sections. The goal is to limit the size

of the recommended tutorial section so that it is not too long. This is done because more concise sections are easier for comprehension and thus would be more useful for programmers. The longer the section is the more effort the programmer needs to invest to find the useful information. We need to split the tutorial in such way that each section will not be too long and will be complete in describing a small task.

The third and the main problem to be solved is to determine if a section is relevant to an API type. The problem of detecting relevance differs from other, apparently similar, problems in NLP such as opinion or sentiment analysis. Relevant features for these problems are explicit expressions of concepts such as agreement, satisfaction, negation. Relevance is often not expressed in a document, rather it is inferred from the document content in combination with user annotations and code examples. For example, a tutorial section can mention more than one API type, however it might only contain useful information for some of them. The main challenge of this sub-problem would be to distinguish between API types for which a section contains or does not contain useful information. This problem is treated as a text classification problem where an API tutorial section–type pair is classified to a "relevant" or "not relevant" category.

The necessary background for the solution of these three sub-problems is presented in Section 2.3.

2.2 Experimental Corpus

Studying how to discover tutorial sections relevant to API types requires a corpus of tutorials. We selected five tutorials covering four different Java APIs. We selected the API tutorials based on several criteria. First, since part of the solution will require access to source code, we chose Java APIs that are open-source. Our solution requires people to look at the tutorial and manually label each section as relevant or not to an API type. That is why we selected APIs so that the application domain of the API is common enough for annotators to complete the task without special training. Besides, tutorials should have acceptable quality,

such as enough coverage of API types, explanation of some complicated parts of the API, basic grammatical quality, etc. Tutorials should be diverse in size, format and origin. For example, the apache.commons.math library tutorial was selected because of its large size, good structure and the complex logic of the API.

Table 2.1 lists all the tutorials selected for this thesis work. It contains the API name, the tutorial, the short name for the tutorial used afterwards in this thesis and the number of words in the tutorial.

Table 2.1: Selected Tutorials for Study

API	Tutorial	Ref. name	N of words
JodaTime API	User guide[1]	JodaTime	4659
apache.commons.math library	User guide[2]	Math Library	28971
Java Collections Framework	Implementations in Java Tutorials[3]	Collections (Official)	23583
Java Collections Framework	Tutorials by Jakob Jenkov[4]	Collections (Jenkov)	12915
Smack API	Documentation[5]	Smack	19075

All of the selected tutorials have different formats and styles. For example, the documentation of the Smack API has a very primitive structure unlike the Math library or Java Tutorials. JodaTime was mainly used for development of thesis work. Math Library was used for testing during the development period, and the other three tutorials were used only for testing purposes.

2.3 Background

2.3.1 Machine Learning

The problem of identifying sections relevant to an API type can be viewed as a text classification problem. Classification is a process that assigns a value from a finite number of discrete categories to each input based on the properties of the input [3, p. 3]. The process can be

[1] http://joda-time.sourceforge.net/userguide.html
[2] http://commons.apache.org/proper/commons-math/userguide/
[3] http://docs.oracle.com/javase/tutorial/collections/
[4] http://tutorials.jenkov.com/java-collections/index.html
[5] http://www.igniterealtime.org/builds/smack/docs/latest/documentation/

based on machine learning algorithms that specify some functions automatically built from the content of a training set with the objective of maximizing classification performance. For the current problem, the input will be the pair formed by an API tutorial section and an API type. The categories to be assigned would be “Relevant” and “Irrelevant”.

There are two types of learning algorithms, namely *supervised* and *unsupervised*. In the former case the training set is annotated with the class labels the algorithms has to automatically generate. In the unsupervised case the learning algorithm attempts to form clusters of input data so that class labels can then be associated with each cluster.

In machine learning categories are usually called *labels*, inputs are called *instances* and the properties of input data are called *features*. The precise *classifier*, which will assign labels to input values, is found during the phase called *learning* or *training* by a supervised machine learning algorithm. During training, the supervised machine learning algorithm goes through examples with correct labels called *training set* and based on the features and the labels of the training set it finds the optimal classifier which will describe the training set as precisely as possible. Afterwards, the trained classifier is tested on new data called *test set* for which labels are unknown to the classifier.

The ability of the trained classifier to correctly classify data samples that are not in the training set is called *generalization*. Since the training set is just a sample from real population of data, the testing set can contain unseen cases of data. For this reason, it is very important to have learning algorithms with a high generalization capability. It is also very important to have a representative training set with good coverage of all the forms in which input data express each classification category. Different machine learning algorithms have different requirements for training data size. Training and testing sets are usually formed by splitting the input data. However it is possible that the most difficult examples or the most easy examples will be in the test set by chance.

To eliminate the chance of unfair splits and for a better evaluation of the generalization of a classifier, *cross validation* is commonly used. *K-fold cross validation* divides the input dataset into k subsets. Afterwards, the model is trained and tested k times, taking one of

the *k* subsets as the test set and the remaining *k-1* subsets as the training set. This way each input data participates in testing the model. The final testing error is computed as the average of all *k* tests. The extreme case of k-fold cross validation is when *k* is equal to the dataset size. That is called *leave-one-out cross validation*. In this case, each single input data is held for testing and the rest is used for training.

There are multiple machine learning algorithms available for text classification tasks. A paper on user reviews classification by Pang et al. [25] uses supervised machine learning techniques for classifying movie reviews. The authors present results for Naive Bayes, maximum entropy (MaxEnt) and support vector machine (SVM) classifiers. In their study the MaxEnt and SVM classifiers showed comparable results while the Naive Bayes classifier performed the worst. Another comparison of supervised machine learning methods for text classification was done by Zhang and Oles [39] which showed similar results.

2.3.2 Natural Language Processing

Natural Language Processing (NLP) is a field of Computer Science which mainly concerns spoken or written language processing and interpretation. Some of the main applications of NLP are information extraction, machine translation, sentiment analysis, question answering. At its foundation are basic operations such as sentence detection, part-of-speech tagging and, tokenization. Some of the low-level operations relevant to this thesis are explained below.

POS tagging Part-Of-Speech tagging is the process of automatically assigning part-of-speech tag to a word. Examples of part-of-speech tags are nouns (NN), verbs (VB), adjectives (ADJ), etc. Besides assigning POS tags, POS tagging tools can also determine additional information such as NNS for plural nouns, VBG for verbs ending with the “-ing” suffix, etc. As an implementation of POS tagging, the Stanford Parser was selected. More information about the tagger can be found in separate publications [31, 32]. Among different tag sets the Stanford Parser uses the Penn Treebank tag set [21].

Stemming Stemming is the process which removes suffixes of the words for transforming related words to a common form. By referring to related words here we mean words which have the same root word but might have different functions. Those words are transformed to the common root, which might not be the morphological root of any individual word. For example, “argue”, “argued”, “argues”, “arguing”, and “argus” reduce to the stem “argu”. For stemming we used an implementation of the commonly used Porter stemming algorithm [26].

Lemmatization In this work, in order to acquire the basic form of the word we used lemmatization. Lemmatization can be viewed as a softer version of stemming, but it uses a more complex process for removing suffixes. Lemmatization uses vocabulary and morphological analysis of words to return a word to its base form or lemma. Words which have the same root will not always be transformed to the same word, because lemmatization uses the POS tag information of a word. For this operation we also used the Stanford Parser.

2.3.3 Information Retrieval

Information Retrieval (IR) is the activity of discovering resources or materials corresponding to the needs of users. IR supports users in browsing and searching activities, supports the task of grouping a large number of documents, and many others. The formal definition [19] of Information Retrieval is as follows

> Information Retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers).

Below the fundamental ideas of IR which are used in this thesis are described in detail.

Vector-Space Model In IR one of the efficient ways of representing a document and the corpus of documents is the vector-space model. For this representation, one vocabulary is built for the document corpus. For each document a vector of the size of the vocabulary is

formed, where each item of the vector corresponds to a word from the vocabulary. If a word exists in the document then the corresponding item in the vector form of the document will have a non-zero value; it will be zero otherwise. There are different ways of calculating the value for the words. For example, the weight can be zero or one indicating the presence or the absence of the word. The weight can be the number of times the document contains a certain word. The most commonly used weight for words is TF-IDF, described in the next paragraph.

Evaluation in IR The most common measures for evaluating the effectiveness of IR techniques are precision, recall, and the measure which averages both, called the F measure [19, p. 155]. Precision is the percentage of relevant items among all retrieved items. In other words, precision shows how noisy the retrieved items are. Recall is the percentage of relevant retrieved items among all relevant items. This shows how effective the algorithm is for finding relevant items. Precision and recall are always in a trade-off. For example if the algorithm retrieves all items then surely all relevant items will be retrieved and recall would be 100%. However, retrieving all items means retrieving not relevant items as well. Therefore, the precision would be low. The opposite will happen if the algorithm retrieves no item. As precision and recall are always in a trade-off, the F measure represents a single measure, the harmonic mean of precision and recall. Various versions of the F measure are possible which can emphasise the importance of precision or recall. However, in this work F1 is considered, which gives equal weight for both recall and precision. The F1 score is calculated according to the following equation:

$$F1 = \frac{2PR}{P+R} \tag{2.1}$$

TF-IDF In information retrieval $TF - IDF$ is a measure of the importance of a term for a certain document. There are multiple ways of calculating the term frequency TF and inverse document frequency IDF [19, p. 118].The most common expression for $TF - IDF$

is as follows:

$$TF - IDF(t,d) = TF_{t,d} * IDF_t = TF_{t,d} * \log \frac{D}{DF_t} \tag{2.2}$$

where $TF_{t,d}$ is the frequency of a term in the current document, IDF_t is the inverse document frequency of a term in the corpus of all documents, D is the size of the corpus and DF_t is the document frequency, which is the number of documents containing the term t. This equation assumes that the importance of the term for document d is inversely proportional to the percentage of documents in which it appears. However, for some problems we used a simplified *TF-IDF* measure which ignores corpus size.

$$TF - IDF(t,d)_{simple} = \frac{TF_{t,d}}{DF_t} \tag{2.3}$$

Cosine Similarity One of the applications of Information Retrieval is the task of assessing the similarity of two documents. As a measure of similarity, the well-known cosine similarity [19, p. 111] is used later in this thesis. The cosine similarity measure assumes a vector-space representation of documents and most commonly uses *TF-IDF* term weighting. The cosine similarity of two documents is calculated as the dot product of the vectors representing the documents.

$$sim(d_1, d_2) = \frac{\vec{V_{d_1}} * \vec{V_{d_2}}}{|\vec{V_{d_1}}||\vec{V_{d_2}}|} \tag{2.4}$$

2.3.4 Maximum Entropy

Maximum entropy (MaxEnt) is a model selection principle for modelling distributions, which we used for obtaining the classifer for our third sub-problem. The MaxEnt approach works in two directions. First, it models a distribution which supposedly generated the training set. Second, it chooses a model with as high *entropy* as possible [20, ch. 16]. The idea of entropy comes from information theory and is defined as

$$H(X) = -\sum_i P(x_i) \log P(x_i) \tag{2.5}$$

Entropy is a measure of uncertainty or the measure of the information content of a variable X. By taking the logarithm in base 2, the entropy will represent the expected number of bits

necessary to encode variable X. It can be shown to have maximum value when the random variable X is distributed uniformly. The maximum entropy principle favours a model which is as uniformly distributed as possible taking into account the training data. Thus, the name of the approach is maximum entropy. The more uniform is the distribution, the more uncertainty is preserved for unseen data.

In general the idea of maximum entropy can be applied to a variety of problems. For text classification we are interested in learning the conditional distribution from training data. One type of maximum entropy model is the *loglinear model* of the following form

$$P(c|d) = \frac{1}{Z} \exp(\sum_i \alpha_i f_i(d,c)) \tag{2.6}$$

where Z is the normalization factor, αs are the learnt feature weights learnt and $f_i(d,c)$ is the value of the ith feature for document d and category c. For text classification, features are usually defined as binary values. Although real-valued features are theoretically possible, binary features are more common because the learning process is more efficient in that case. For example, Nigam et al [23] described the use of the maximum entropy method for text classification with real-valued features. In general, features are defined as

$$f_{i,c'}(d,c) = \begin{cases} 0, & \text{if } c \neq c' \\ w_i, & \text{Otherwise} \end{cases} \tag{2.7}$$

The α weights of the features can be learnt using a variety of optimization algorithms. The literature most commonly mentions iterative scaling [14, ch. 13], generalized iterative scaling [20, ch. 16], and improved iterative scaling [23]. The solution we used by default uses the online limited-memory quasi-Newton BFGS optimization method [24, p. 224], which has been shown to outperform other optimization algorithms for MaxEnt classifiers [17].

In this work features get a category as an argument. In practice, features are created regardless of the category and afterwards, during the classifier training process, the weights of features for each category are calculated during the parameter optimization. For example, suppose we have a set of documents to classify into the “animals” or “politics” categories.

One of the possible features can be the existence of the word "government". During parameter estimation two α weights will be calculated: one for each category. Supposedly, the weight for this feature in the "politics" category would be bigger than for the "animals" category. After all features are created and all feature weights are estimated, then for the classification of a new document, the conditional probability for each category is calculated according to Eq. 2.6. If the new document contains the word "government" then that feature will contribute more to the category "politics" and little for the category "animals". The category with the bigger conditional probability for the document would be chosen as the classification result.

As any other machine learning technique, MaxEnt can suffer from *overfitting*. Overfitting occurs when the classifier chooses a model which adjusts so much to the training set that it performs poorly on the testing set. For example, let us consider a case where a classifier is trained to label data with positive and negative labels. If one of the features occurs once in the training set and the data is positive, then the evidence will suggest for it to have extreme positive weight. However, in general that might be an incorrect assumption. The testing set might contain data with the same feature which are actually negative examples. To overcome this problem usually smoothing or priors are used. For each feature a prior probability is estimated and applied to the gathered evidence of the feature. This will smooth out the extreme values. The solution used for the current problem uses a *Gaussian Prior* which is defined as

$$P(\alpha_i) = \frac{1}{\sqrt{2\pi\sigma_i^2}} \exp\left(-\frac{(\alpha_i - \mu_i)^2}{2\sigma_i^2}\right) \tag{2.8}$$

where σ is usually advised to consider as small (in our implementation it is equal to 1) and μ as 0 [23]. In this particular setting, Gaussian priors helps to keep weights near their mean and move them away when evidence is found, with a rate controlled by σ. For example, the case when σ is infinite corresponds to no smoothing. The more evidence for a feature is found, the more its weight grows. Therefore, the use of priors favours features for which the training set has more evidence. This is logical because common features carry more predictive information. More general features will have higher weight, as opposed to more

specific features. The use of priors smooths weights of the features in order to avoid extreme weights. There is, of course, a trade-off between closely mapping to the observed data, and avoiding extreme values for weights.

One of the issues in classification tasks can be the unbalanced distribution of instances in different classes (e.g., a very high ratio of negative instances, or vice-versa). The trick usually used with MaxEnt to account for that is the introduction of a dummy feature. This is a feature which has constant value for all of the data. As a result, during the training process the feature gets higher weight for the class which is more probable in the training set. Using this approach we computed a $class - feature$ to address the issue of unbalanced negative and positive examples.

2.4 Related Work

2.4.1 Natural Language Processing and Information Retrieval

This section presents some of the foundational works in Natural Language Processing (NLP) and Information Retrieval (IR), ideas which affected the solution of the presented problem. One of the influential papers in Natural Language Processing is Turney's [33] paper on unsupervised classification of reviews as positive or negative. This paper shows that the overall polarity of a review depends on the polarity of some of its phrases. For each phrase a score is calculated, which is the mutual information between the phrase and the terms "excellent" or "poor". The total polarity of the review is the average of the polarities of its phrases. Similarly, in case of our problem, tutorial sections contain information that can be relevant or not relevant for an API type. The same intuition can be used to assess the relevance of the section based on the specific phrases from the section text.

Kolcz et al. [15] suggest using ideas from text summarization as features for text classification. In particular, they propose using the idea of text zones such as title, first paragraph, paragraph with most title words, the first two paragraphs, first and last paragraphs and all

sentences with a minimum number of title words. This idea is also used later in our solution.

Another two more recent papers which used a set of different NLP and IR tricks for automatically extracting knowledge from natural language texts are presented by Fan et al. [7] and Yates et al. [38]. Fan et al. [7] use a set of NLP operations to extract regular linguistic patterns and then based on that infer additional axiomic knowledge using ontology-based generalization. Yates et al. [38] describe the Open IE (OIE) method for automatically extracting a set of relational tuples from a natural language text. The paper also introduces TextRunner, a system which assigns probability to tuples and indexes them. Tuples of related words are automatically extracted from natural language text using machine learning techniques

2.4.2 Natural Language Processing and Information Retrieval in Software Engineering

Software systems have various accompanying natural language resources, such as API documentation, API specifications, requirements, user manuals, bug reports, etc. For building systems to improve software quality and productivity, it is important to analyse and use the information provided in natural language resources. Natural language processing and information retrieval techniques have been shown to help with a variety of software engineering problems. The following papers are examples of successful use of NLP and IR techniques for a variety of Software Engineering problems.

Anvik et al. [1] use NLP techniques to assist the bug assignment task. They convert each bug report to a feature vector indicating the frequency of the words in the report. Afterwards, using the names of assigned developers to the bug as labels, the authors train a text categorization system using SVM.

Wang et al. [35] suggest a duplicate bug detection approach. They use a combination of two features in which one is a similarity measure between new and existing bugs. As a similarity measure, the dot product of the vector representation of two bugs was used.

Ashok et al. [2] create the DebugAdvisor system to assist developers in acquiring the necessary context during a bug fixing task. DebugAdvisor is an information retrieval system that can take long queries and apply them to a search database consisting of different types of documents.

Xiao et al. [37] describe a system called Text2Policy which uses NLP techniques to extract user access control policies from requirements. In general, Text2Policy uses shallow parsing of the document sentences and employs that information to identify and classify access control polices.

Fry et al. [10] suggest a system for reducing software maintenance effort. They argue that using natural language clues helps in understanding the source code. The approach is to extract clues in the form of verbs and direct objects from the source code.

Zhong [40] introduces the Doc2Spec system, which applies NLP techniques to API documentation in order to infer API specification. Zhong uses Named Entity Recognition (NER) along with a machine learning approach to identify action-resource pairs from JavaDocs.

The other approach to Software Engineering problems is to build structured models of concepts of a software project and relationships among them called ontologies. Bontcheva and Sabou [4] propose uniting all software engineering artifacts (e.g. source code, discussion forums, user manuals, etc.) into one ontology. They present an unsupervised technique for learning an ontology from different types of source system artifacts, followed by user validation. For identifying terms and concepts, the authors used several NLP techniques, such as lemmatization, *TF-IDF*, *POS tagging* and word co-occurrence.

2.4.3 Automatically Interpreting API Documentation

We are not aware of any previous study on analyzing API tutorials. However, there are systems that serve the same general purpose of the work described in this thesis. The general purpose of this study is to assist developers while they try to gain more knowledge about a selected API type. Chhetri's master thesis [5] finds and recommends knowledge items from

API reference documentation (e.g. JavaDoc).

Rigby et al. [27]'s work on StackOverflow posts mainly focuses on traceability recovery of code elements. In their work the authors also suggest a method for identifying important code elements. The authors also present a comparative study of traceability systems, showing that Recodoc performs best if the list of elements to link to is limited to certain APIs.

Henß et al. [12] took another approach of making knowledge more accessable to developers. The authors automatically extract FAQs from mailing lists and Q&A online forums. First, they identify question topics using a machine learning technique called Latent Dirichlet Allocation (LDA). LDA models word conditional probabilities as a combination of conditional probabilities of a word given a hidden topic in a fixed size set of topics. Then they select Q&A for each topic, and finally order the selected Q&A within each topic by relevance of the question and the answer to the topic.

Another branch of research tries to improve the search experience for software developers. Assieme [13] is a web search interface that links different sources of information by identifying implicit references of code elements in code examples. Afterwards the authors indexed documents using text around the code example. Mica [29] is another tool meant to assist developers in their search process. Mica is built on the Google web API. It takes Google search results, finds out all keywords of the Java JDK, orders them according to global frequency and total popularity, and if possible, groups them using the hierarchy in the API. The identified keywords in proper order are displayed in the sidebar and serve as a link to a list of all results containing that keyword.

Webar et al. [36] extract tips from Yahoo answers for "How-to"-like questions. First they select forum questions which are in "How-to" form. Afterwards they extract sentences from the answers which correspond to certain patterns for tips, taking into account part-of-speech tagging and grammatical rules, then they transform the questions and extracted sentences into tips. A sample from all extracted tips was evaluated using crowdsourcing and used as a training set for predicting the quality for all extracted tips.

3

Preprocessing and Classification

In this section we present the detailed solutions for sub-problems as well as the preprocessing work necessary for the last sub-problem, which is the classification of API tutorial section–element pairs to relevant or not relevant categories. The first sub-problem is API type identification and is solved by using a special tool. The description of the tool and the detailed solution of this sub-problem is presented in Section 3.1. The second sub-problem is tutorial segmentation. A tutorial is segmented by a two-phase algorithm, where the first phase splits the tutorial as much as possible and the second phase groups those small chunks according to the logic of HTML and the desired length of the sections. A more detailed description of the algorithm and the results for the experimental corpus is presented in Section 3.2. Section 3.3 describes in detail all pre-processing steps applied to the API tutorials before text classification, and the details of the text classification.

3.1 Finding API Elements

Finding API types mentioned in the API tutorial can be also viewed as a traceability problem between an API and its learning resources. For this task we used Recodoc [6], which identifies API types in two phases. First, it finds all words which might be API elements, based on the HTML tags and the title of a section where the word was found. Such words are called *code-like terms*(CLTs). Second, it disambiguates code-like terms. As Dagenais and Robillard [6] state, on average, methods with the same name were declared in 13.5 different types in the

four open source systems they studied. Recodoc was showed to find API elements from API tutorials and mailing lists with 96% precision and recall. This means that around 96% of the identified API elements were correct and 96% of all API elements mentioned in the four studied tutorials were identified by Recodoc. Recodoc not only correctly identifies API elements, but also identifies almost all mentioned API elements.

Recodoc first forms a *codebase* based on the source code of the API. A codebase is the collection of API elements in the API. Recodoc links code-like terms to one or more API elements for which the name matches. As mentioned, different types can contain methods and fields with the same name. Therefore, code-like terms might be ambiguous if multiple matches were found. To overcome this, Recodoc takes into account the context of the code-like term.

Recodoc defines three levels of context. The immediate context of code-like term c includes all code-like terms which are mentioned with c. The local context of c contains all code-like terms in the same section as c. The global context of c contains all code-like terms in the document or tutorial page. For example, consider a section from Smack API tutorial 2.1. In this example, the intermediate context of the API element *createMessage* is *Chat* element, the local context is *Chat*, *sendMessage*, *String*, *Message*. The global context will contain all API elements in the same document as the discussed section, including all API elements from the local context.

Based on these levels of context, a code-like term is considered closer to the other code-like terms if it appears in more specific context. Recodoc also uses the source code of the API to verify the existence the API element. As a result, Recodoc returns all API elements with fully qualified names (FQN).

3.1.1 Data

To identify API elements in the tutorials of the experimental corpus, Recodoc was applied on each of them. The statistics of the results are shown in Table 3.1. For example, Re-

codoc found 901 code-like terms for Math Library, out of which 418 were linked to the apache.commons.math API.

Table 3.1: Recodoc Results for Studied Tutorials

Tutorial	N of CLTs	N of links
JodaTime	136	72
Math Library	901	418
Collections (Official)	711	574
Collections (Jankov)	556	409
Smack	574	273

The large difference between the number of code-like terms and the number of API elements is usually a result of using specific HTML tags for parameter names, formulas, numbers and other words which are not API elements. Recodoc also helps to eliminate noisy cases.

3.2 Tutorial Segmentation

The next step is to split the API tutorial into reasonably small sections. We split the tutorial so that the length of each segment is between a desired minimum and maximum length. For setting maximum and minimum lengths we looked at the small tutorial JodaTime. We divided it only according to the content table and found the lengths overall convenient.

Figure 3.1 shows section lengths for the JodaTime API tutorial. Each bar corresponds to a section and the height corresponds to the number of words in each section. As can be seen from the plot, the average length is around 100 to 150 words. Based on this observation, we selected 100 words as a *min* length and 150 as a *max* length for tutorial sections.

We complete the task of segmentation of a tutorial by using the HTML structure of tutorials. We measure the length of the section by word count, excluding code snippets and HTML tags. As a result, a section which contains only code snippets will have length 0. To split the tutorial, we first split it into as small pieces as possible by taking into account the HTML structure. Afterwards, the pieces are merged back based on continuity and length. The following is the step-by-step description of the algorithm.

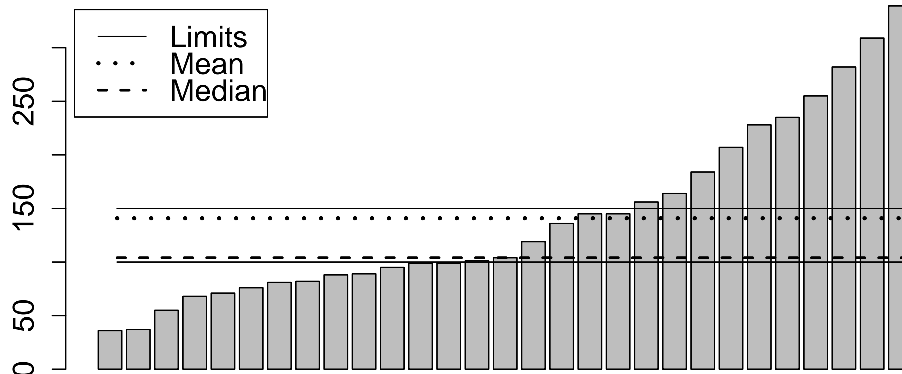


Figure 3.1: Section Lengths for JodaTime API Tutorial

Step 1. Split the tutorial according to content table (if provided) or header tags.

Step 2. Exclude the title of the section from the following steps

Step 3. For all sections, split in step 1, recursively split all HTML elements if they are longer than $max - min$(e.g. longer than 50 words). We try to split HTML elements to be smaller than $max - min$, because if so, later, it would be easier to keep segments within the range of $[min, max]$. We do not split HTML elements even if they are longer than $max - min$, if

a) the HTML element consists of one HTML tag

b) the HTML element is `<P>` HTML tag

c) the HTML element is `<UL>` HTML tag, which is for enumeration

d) the HTML element is `<TABLE>` HTML tag

e) the HTML element is `<DD>` HTML tag

f) the HTML element is `<DT>` HTML tag

Step 4. Merge HTML elements by using the structure of the HTML

a) `<DT>` should be merged with the following `<DD>`s (usually `<DT>` serves as a title for `<DD>`s)

b) `<UL>` should be merged with the previous text (a section cannot start with enumerating something without a previous introduction)

Step 5. Sequentially merge sibling elements of HTML until the length of the resulting element exceeds the *min* length. Afterwards, following siblings with 0 length are also added. Adding a single element might increase the length of a section to exceed the *min* length. However, as elements were split to be smaller than the $max - min$ except for a few exceptions, the section length will be shorter than the *max* length.

Step 6. If a section has been split into subsections in Steps 3,4,5 then the title of the subsections is formed by concatenating the title of the section excluded in Step 2 with the subsection index.

Note that we can still get sections which are longer than the *max* length, because we also want to get sections not only with reasonable size but also with reasonable continuity. By merging first by logic, we might get chunks of the text which after grouping will exceed the *max* length. This is because we do not want to sacrifice the logic of the section because of the length. Also sections can be smaller than the *min* length if the initial length is small.

3.2.1 Data

All tutorials except JodaTime were segmented using the algorithm described above. Table 3.2 presents the results of the algorithm. For example, the Math API tutorial was initially divided into 77 sections according to the content table of the tutorial. Those 77 sections were afterwards divided into 158 sections with an average length of 203 words. This is longer then the desired ideal length (between 100 and 150 words) but as mention previously we prioritize the connectivity of the section above its length. Furthermore, 418 Math API

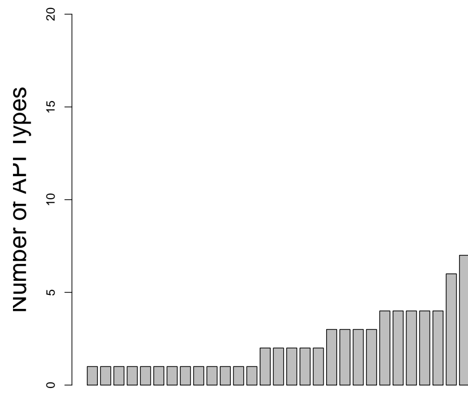
elements, identified by Recodoc, were mapped to 102 tutorial sections. This implies that in the other 46 sections no API element was found. API elements other than classes and interfaces were substituted with the corresponding class. Overall, there were 251 API type - section pairs, out of which 102 were used for the development of the classifier for the third sub-problem.

Table 3.2: Segmentation Results for Studied Tutorials

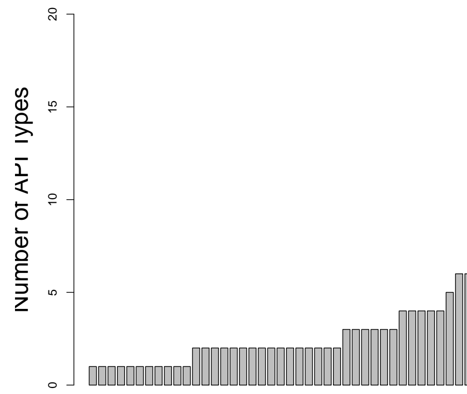
Tutorial	Sections	Segmented Sec.	mean(lengths)	meadian(lengths)	Linked Sec.	Pairs	Annotated
JodaTime	33	33	140	104	29	72	72
Math Library	77	158	203	201	102	251	102
Collections (Official)	56	73	172	163	57	233	233
Collections (Jankov)	70	79	141	132	69	150	150
Smack	60	65	229	212	46	86	86

Figure 3.2 illustrates the density of API types per section for all five studied tutorials. Each bar corresponds to a section of the tutorial and the height of the bar represents the number of distinct API types in the section. From figure 3.2 one can observe that usually the number of API types per section is under 10, however Collections (Official) and Collections (Jenkov) contain a few mega-sections which have more than 10 API types.

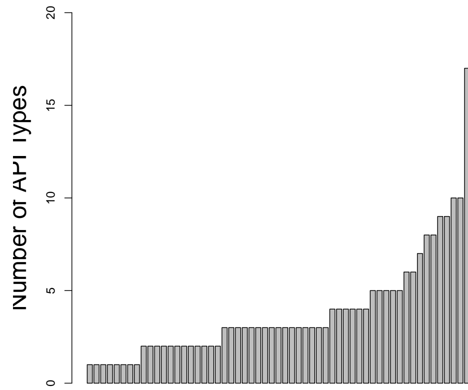
Similarly, Figure 3.3 illustrates the popularity of API types. Each bar corresponds to an API type and the height of the bar represents the number of sections mentioning the API type. For example, in Collections (Jenkov) (Figure 3.3(d)) the popularity of API types is diverse, unlike in JodaTime (Figure 3.3(a)), where one API type (DateTime class) is much more popular than the majority of API types, which appear just once in the whole tutorial.



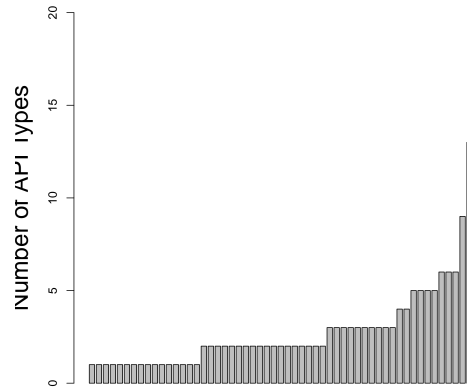
(a) JodaTime



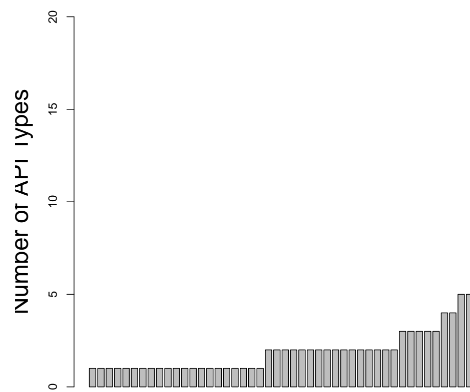
(b) Math Library



(c) Collections (Official)

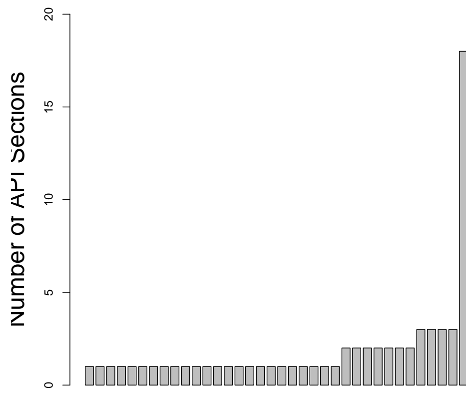


(d) Collections (Jenkov)

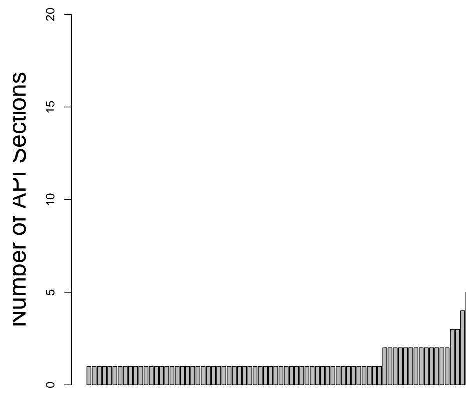


(e) Smack

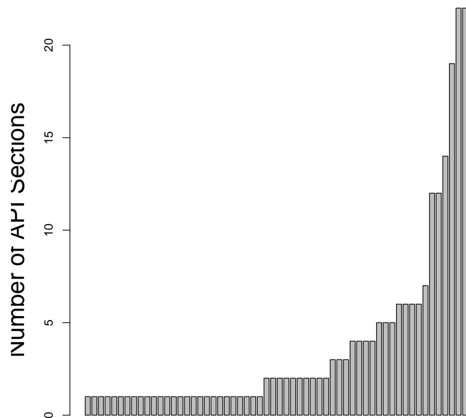
Figure 3.2: Number of API Types per Section



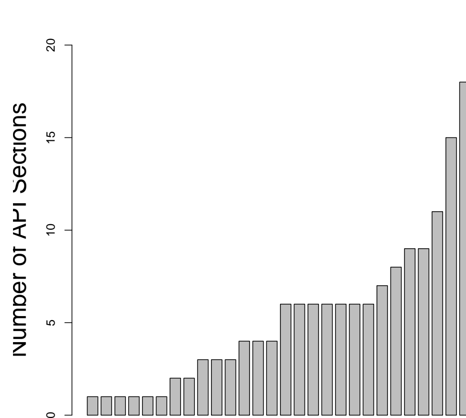
(a) JodaTime



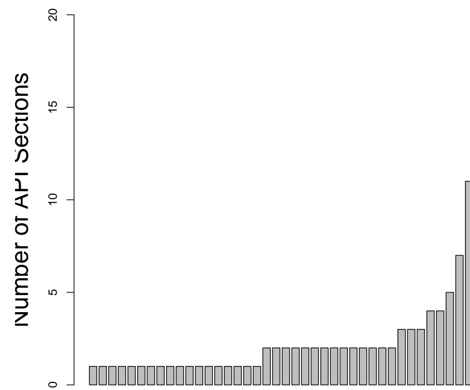
(b) Math Library



(c) Collections (Official)



(d) Collections (Jenkov)



(e) Smack

Figure 3.3: Number of Sections Mentioning API Type

3.3 Relevance Classification

This section will describe how we identify sections relevant to an API type. As described in the previous sections, each API tutorial is processed by RecoDoc, and by the segmentation program. For addressing the third sub-problem we carefully preprocessed API tutorials before applying the main algorithm. Afterwards, for formed API tutorial section-API type pairs, a classifier is created using the features that we designed. Using the experimental corpus and the created classifier we automatically predict whether a section is relevant or not relevant to an API type. The following subsections will describe in detail the necessary preprocessing steps for API tutorials in Section 3.3.1, our choice of classifier in Section 3.3.4, and its features in Sections 3.3.2 and 3.3.3.

3.3.1 Preprocessing

This section describes all of the NLP preprocessing steps applied to the API tutorials. All the preprocessing steps aim to improve the performance of the NLP operation specifically for API tutorials.

Sentence splitting One of the basic NLP operations applied to API tutorials is sentence splitting. For splitting text into sentences, the Stanford Parser[1] was used. It considers the end of the sentence to be found when a sentence-ending character (., !, or ?) is found which is not grouped with the preceding word (such as for an abbreviation or number). However, HTML files, where the text is separated with HTML tags, might not have proper punctuation. That is why, after the HTML file is converted to text, sentences or text blocks (e.g. paragraphs, enumerations, tables, etc.) are merged together. For example, titles never contain punctuation, therefore after HTML tags are removed, the title sticks to the first sentence. Another problematic case is the enumeration, which forms one gigantic sentence if punctuation is omitted. Besides, API tutorials contain code snippets which might be

[1]http://nlp.stanford.edu/software/corenlp.shtml

inserted in the middle of a sentence. For overcoming these problems, two preprocessing steps are introduced and applied on API tutorials: code snippet collapse and HTML-based punctuation.

Collapsing code snippets For processing API tutorials, tutorial texts are cleaned of code snippets. If necessary, the text blocks preceding and following the removed snippets are linked. First, all the code snippets are collapsed into a special keyword holding a unique identifier by which the original code snippet can be referred. The unique identifier has the form of "CODEID=\d" where \d is the number that can be used to retrieve the original code snippet. Afterwards the program determines if the code snippet is part of any sentence or not. For example, Figure 3.4 shows a section of the JodaTime API tutorial in HTML format before collapsing code snippets. When the code collapsing algorithm is applied to it, it transforms

```
1 ... In datetime maths you could say: </p>
2
3 <div class="source">
4       <pre>   instant + duration = instant </pre>
5 </div>
6
7 <p>Currently, there is only one implementation of....
```

Figure 3.4: Case 1: Before Modification

into the result presented in the Figure 3.5. Note that in the modified version, punctuation (e.g. ".") is added after the code snippet, to indicate the end of the current sentence and the beginning of the new one. Now the sentence detector can successfully separate this example

```
1 ... In datetime maths you could say: CODEID=0.
2 Currently, there is only one implementation of...
```

Figure 3.5: Case 1: After Modification

into two sentences where CODEID will be part of the first sentence. Figure 3.6 shows the second case, in which a sentence does not finish with the first code snippet, but continues until the second code snippet. The result of the code collapse algorithm can be seen in 3.7.

Collapsing code segments will not only make sentence detection better and cleaner, but will also create a possibility to exploit the relationship between sentence words and code

```
Similarly, for JDK <tt>Calendar</tt>: </p>

<div class="source">
   <pre>
      // from Joda to JDK
      DateTime dt = new DateTime();
      Calendar jdkCal = dt.toCalendar(Locale.CHINESE);

      // from JDK to Joda
      dt = new DateTime(jdkCal);
  </pre>
</div>

<p>  and JDK <tt>GregorianCalendar</tt>: </p>

<div class="source">
   <pre>
      // from Joda to JDK
      DateTime dt = new DateTime();
      GregorianCalendar jdkGCal = dt.toGregorianCalendar();

      // from JDK to Joda
      dt = new DateTime(jdkGCal);
   </pre>
</div>
```

Figure 3.6: Case 2: Before Modification

```
Similarly, for JDK <tt>Calendar</tt>: CODEID=25 and JDK <tt>GregorianCalendar</tt>:
     CODEID=26.
```

Figure 3.7: Case 2: After Modification

snippets. A code snippet now is just another word in the sentence and as any other word in the sentence, it has relationships with sentence words. As API types are also part of the sentence, now we can identify the relationship between an API type and the code snippet. If a relationship does not exist, at least we can identify whether the API type appears in the same sentence with a code snippet. This is important information because usually important API types of the code snippet are mentioned before it. For example, in Figure 3.7 each code snippet is preceded by the API type which is the main API element of the code snippet.

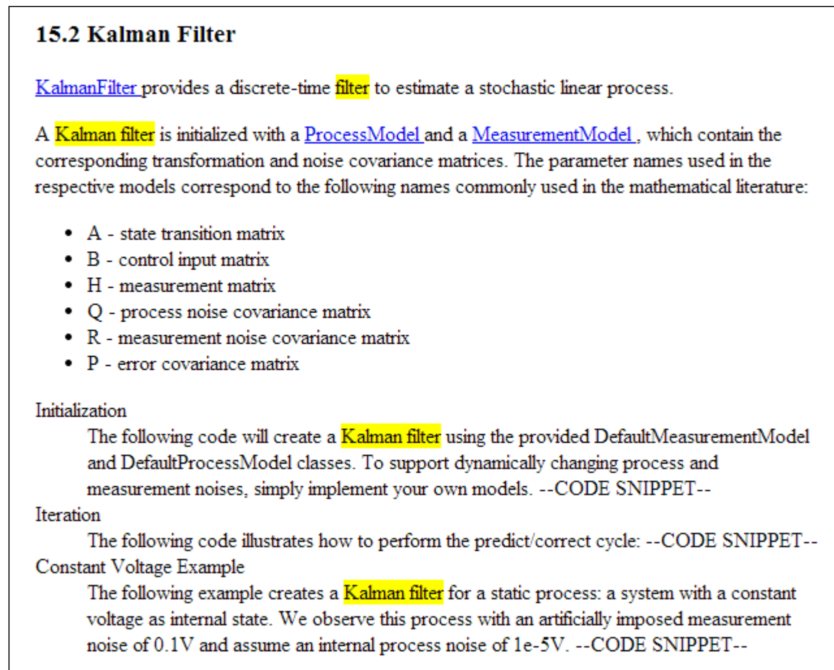
Adding punctuation The second preprocessing step for improving sentence detection is the addition of punctuations based on the HTML of the API tutorial. As tutorial blocks are originally separated by HTML tags, punctuations to indicate the end of the sentence are sometimes omitted. Therefore, when the text of the API tutorial is extracted, the sentences lacking punctuation are merged together. That is why it is necessary to add punctuation where possible before converting HTML to text.

API tutorial blocks like titles, the text in table cells, and the text in enumeration lists, usually lack punctuation in the end and do not start with upper-case. Sometimes paragraphs can miss the dot symbol in the end of the last sentence as well. After converting HTML to text, some of the sentences are merged into one sentence. This leads to undesirable behaviour of the sentence detector. That is why we added punctuation based on the HTML structure before converting the HTML to text. If HTML tags such as `<H1>`, `<H2>`, `<DT>`, `<DD>`, `<P>`, `<TR>` end without punctuation then a dot is added at the end. In case of itemized lists such as `<LI>`, `<TD>`, a comma is added for every item except the last one, and a dot for the last one.

For example, Figure 3.8 shows a section from the Math Library tutorial, which contain a list (which uses the `<LI>` tag) followed by titled paragraphs (uses `<DT>`, `<DD>` tags). If no full stop is added then a sentences will be constructed that includes the phrase preceding the list, all the elements in the list, the title "Initialization", and the sentence following the list ("The following code..."). After adding the punctuation the list would form one sentence with the unfinished sentence preceding it, “Initialization” would be the next sentence, etc.

POS tagging As a POS tagger, the Stanford Parser is used as well. Though this package is very powerful, it was trained on the Wall Street Journal corpus of the Penn Treebank. Therefore, the POS tagger might be inaccurate when applied to technical documentation. Technical documentation contains a lot of unseen words or words in unseen contexts. As a result, Software Engineering specific words and phrases are tagged incorrectly.

For example, the POS tags assigned to the sentence “If this method returns false, the op-



15.2 Kalman Filter

KalmanFilter provides a discrete-time filter to estimate a stochastic linear process.

A Kalman filter is initialized with a ProcessModel and a MeasurementModel , which contain the corresponding transformation and noise covariance matrices. The parameter names used in the respective models correspond to the following names commonly used in the mathematical literature:

- A - state transition matrix
- B - control input matrix
- H - measurement matrix
- Q - process noise covariance matrix
- R - measurement noise covariance matrix
- P - error covariance matrix

Initialization
The following code will create a Kalman filter using the provided DefaultMeasurementModel and DefaultProcessModel classes. To support dynamically changing process and measurement noises, simply implement your own models. --CODE SNIPPET--
Iteration
The following code illustrates how to perform the predict/correct cycle: --CODE SNIPPET--
Constant Voltage Example
The following example creates a Kalman filter for a static process: a system with a constant voltage as internal state. We observe this process with an artificially imposed measurement noise of 0.1V and assume an internal process noise of 1e-5V. --CODE SNIPPET--

Figure 3.8: A Section from Math Library Tutorial: Highlighted for `KalmanFilter` API Element

erating system does not support changing the display mode."[2] are {If/IN, this/DT, method-/NN, **returns/NNS**, **false/JJ**, ,/, ,the/DT, **operating/VBG**, system/NN, does/VBZ, not/RB, support/VB, **changing/VBG**, the/DT, display/NN, mode/NN, ./.}. The existence of the terms "false" and "operating system" confuses the parser, which tags "returns" as plural noun, "false" as adjective and "operating" as verb. In English, false is truly an adjective and operating system is a noun phrase containing a verb, but in technical language those are concepts.

One of the solutions for this problem would be to collect a corpus of technical documentation with correct POS tags and train a new model, as is done in the work of Gimple et al. [11]. The authors described the similar problem for POS tagging of Twitter data. They manually labelled and trained a new model for tagging this corpus. This would be a great solution for technical documents as well, however, it requires a lot of time and resources.

The other way to overcome the mistagging problem would be to distinguish concepts from other words and phrases, and modify the POS tagger accordingly. Quality would be

[2]example taken from from http://docs.oracle.com/javase/tutorial/extra/fullscreen/displaymode.html

compromised, as this would not be a general solution but would cover the cases in which the most common concepts are used. Our solution identifies commonly used multi-word phrases and modifies the POS tagger to force the POS tags for them. The details are presented in the following paragraphs.

Multi-word concepts The Software Engineering field, as all other fields, has its own lexicon. Often Software Engineering terminology contains concepts which are constructed from one or more words. These combinations of words might seem unusual for NLP tools which are trained on a general corpus of data. For example, phrases like “operating system”, “return value”, “scroll pane” or “source code” can be misinterpreted by general NLP tools which tag the words “operating”, “return”, “scroll” as verbs. As the solution of the third sub-problem in this thesis uses POS tags of the words, this behaviour becomes an obstacle and needs a solution.

To overcome mistagging of technical concepts, we used a multi-word term detection algorithm [9]. This method outputs multi-word phrases with corresponding scores. With the use of this algorithm, we extracted more than 30000 multi-word phrases from official Java Tutorials[3]. Around 20000 phrases had a score bellow two. After manual inspection of the output, we manually selected 10 as a threshold, which filtered out 1037 multi-word concepts (Appendix B). The selection of those 1037 concepts is based solely on the manually selected threshold. Therefore, the selection contains some noise. Some of the selected multi-word concepts are not real concepts, but reoccurring word combinations (e.g. new connection, current JDK, etc.). Those cases are not manually removed, because they do not have a negative effect on the following steps of our solution. Besides, in some cases it is questionable whether a certain phrase represents a concept or not. For example, in the case of “last element”, “default layout” it is not obvious whether they are concepts or not. Therefore, for the cleaning concept list we would need multiple annotations to avoid subjectiveness. The extracted concepts are not representing all Computer Science fields but are biased towards

[3]http://docs.oracle.com/javase/tutorial/

Java terminology.

After obtaining the list of concepts, words in these phrases were concatenated so that from here on they would be treated as one token, and the POS tagger was forced to tag the phrases as a noun.

Group noun phrases During multi-word concept extraction we observed that usually an API type is used in the sentence as part of a noun phrase. For example, in the sentence from JodaTime user guide[4]: "Within Joda-Time an instant is represented by the ReadableInstant interface", ReadableInstant is an API type and the noun phrase it is part of is "the ReadableInstant interface". According to the Stanford Parser, "interface" is the main noun and ReadableInstant is a descriptive word for it. This means that in the above example, "interface" is the object which "is represented by ...". Later, in the solution of the third subproblem, when we explore the relationships of API types with other words in the sentence, it is important to map relationships to the API type "ReadableInstant" instead of "interface".

To overcome this problem, we replaced all noun phrases which contain an API type with the API type itself. For example, in our previous case "Within Joda-Time an instant is represented by ReadableInstant".

POS tagger modification For tagging concepts and API types we needed to force the tagger to tag them as nouns. It is important to do this before tagging the whole sentence, because one mistagged word can lead to the wrong tagging of the whole sentence. That is why it is not enough to overwrite tags for concept words and API types after the tagging task. It is important to modify the tagger so that from the beginning, it will have proper tags for concepts and API types. For that, we added one more step before calling the Stanford Parser which pretags concepts and API types as nouns. Only after that, the parser will be called on the partially tagged sentence.

[4]http://joda-time.sourceforge.net/userguide.html

3.3.2 Feature Design

For constructing features, the JodaTime tutorial was used as a development set. The Math Library tutorial was used to validate our initial intuitions. The other tutorials were not used during the feature design and were reserved for the evaluation. Both linguistic and structural properties of the text were taken into account. Some of the features are real-valued, others are binary. All used features were divided into five groups, based on the level of detail, and are presented in Tables 3.4, 3.5, 3.6, 3.7.

Real-Valued Features

Table 3.3 lists real-valued features used for classification.

Table 3.3: Real-Valued Features

Feature	Short Description
freq	Feature showing how important is an API type for an API tutorial section, according to the $TF-IDF_{simple}$ formula
wordNum	Feature describing cases when an API type or part of it are mentioned in the text as simple words, not as code words
substituteNum	Feature describing cases when an API type is a substitute for its methods and fields

freq As mentioned in Section 2.3, $TF-IDF$ is a measure widely used in IR to describe the role of a term in a certain document. For feature *freq*, we chose the simple version of *TF-IDF* over the version normalized with the corpus size. Although normalizing the features is preferable for the MaxEnt classifier, we decided not to be dependent on the corpus size which in this case should have been the number of sections in the tutorial. Intuitively, penalizing this feature for the section from longer tutorials is not appropriate. The most common expression for $TF-IDF$ is as follows:

$$TF-IDF(c,s)_{simple} = \frac{TF_{c,s}}{DF_c} \tag{3.1}$$

where $TF_{c,s}$ is the frequency of the API type in the section and DF_c is the section frequency, which is number of sections containing the API type c.

wordNum This feature represents how many times an API type or part of it is mentioned in the text of the section. Some of the API types or part of API type names can be used in the text of the section as simple words and not as a code term. This feature counts those occurrences and calculates a weight, which we call the *wordNum* feature. For calculating the *wordNum* feature, first the unnormalized weight is calculated according to following formula:

$$w(c,s) = \sum_{lw \in LW(s)} AreAlike(c, lw) \tag{3.2}$$

where $LW(s)$ is the set of lemmas of all words in the section s. $AreAlike$ is a function which returns 1 if lw matches fully the API type c name and 0.5 if lw matches c partially. For detecting a partial match, c is split by CamelCase and each part is compared with lw.

It is known that unnormalized features might cause problems for classification [23]. As Nigam et al [23] observed, unscaled frequencies of the terms, negatively affected the accuracy of the MaxEnt classifier. During the initial experiments we observed the same behaviour, and for that reason we decided to normalize the *wordNum* feature. As there is no theoretical upper limit for w, we used an approximate upper limit. We chose 5 to be the upper limit and calculated the final value for the feature as follows

$$wordNum(c,s) = \begin{cases} 1, & \text{if } w(c,s) \geq 5 \\ \frac{w(c,s)}{5}, & \text{Otherwise} \end{cases} \tag{3.3}$$

For example, Figure 3.8 shows a section from the Math Library tutorial. For the displayed section and the `KalmanFilter` class, the *wordNum* feature value is calculated to be 0.7. No full match and 7 partial matches are found. As each partial match is equal to 0.5, the unnormalized weight would be equal to 3.5 according to Equation 3.2. After normalization, the value we get is 0.7. For the *wordNum* feature, the title is not considered, because it is considered as part of the *inTitle* and *inParentTitle* features discussed later in this section.

substituteNum This feature is calculated as follows:

$$substituteNum(e, s) = \frac{Count_{substitute}(e, s)}{Count(e, s)} \tag{3.4}$$

As mentioned in Chapter 2, the decision was made to substitute all methods and fields with their declaring class. The *substituteNum* feature is used to quantify how often classes are introduced to substitute methods or fields, opposed to when classes are explicitly mentioned.

Tutorial Level Features

Table 3.4 lists tutorial level features. These are calculated on the basis of the whole tutorial and have binary values.

Table 3.4: Tutorial Level Features

Feature	Description
$inParentTitle$	TRUE for $\langle s, c\rangle$ if s is a subsection of another section S and c is present in the title of S
isOnlyOne	TRUE for $\langle s, c\rangle$ if s is the only section of the tutorial which mentions c

inParentTitle The sections of some tutorials have hierarchical structure; however, in this work the hierarchical structure is ignored, except for the *inParentTitle* feature. If a section has a parent section, then the API type is compared with the title of the parent section. A match is found if the API element or part of it is mentioned in the title.

isOnlyOne The *isOnlyOne* feature is a binary feature which describes whether the API type is common or not across the tutorial. Intuitively, if the section is the only section mentioning the API type, then there are higher chances that the section is focused on the API type.

Section Level Features

Table 3.5 lists section level features, which are calculated based on the information of a single section of the tutorial.

Table 3.5: Section Level Features

Feature	Description
inCode	TRUE for $\langle s, c\rangle$ if s contains a code snippet and the code snippet contains c
notInCode	TRUE for $\langle s, c\rangle$ if s contains a code snippet and the code snippet does not contain c
moreThanOnce	TRUE for $\langle s, c\rangle$ if c is mentioned in s as a code term more than once
once	TRUE for $\langle s, c\rangle$ if c is mentioned in s as a code term only once
inTitle	TRUE for $\langle s, c\rangle$ if c is present in the title of s
inFirstSent	TRUE for $\langle s, c\rangle$ if the first sentence of s contains c as a simple word or as a code word

inCode and notInCode These features are one of the few cases when the information from code snippets is used. Both these features appeared to be very effective, as important API types used in the code snippets are usually discussed in the text and vice versa.

moreThanOnce and once These features are also very effective, as important API types are usually mentioned more than once in the text. If an API type is mentioned only once in the section, then more evidence is necessary to prove the importance of the API type.

inTitle This feature is similar to the *inParentTitle* feature, but here only the title of the current section is considered. If an API type is mentioned in the title of the section, this is important evidence that the API type is the focus of the section.

inFirstSent This feature describes the cases in which the text of the section begins by mentioning the API type. Intuitively, if the text of the section starts with the discussion of an API type, then the API type should have an important role in the section.

Sentence Level Features

Table 3.6 lists sentence level features, whose values are calculated based on the individual sentences of a section.

Table 3.6: Sentence Level Features

Feature	Description
isExample	TRUE for $\langle s, c\rangle$ if any sentence of s mentions c as an example
isInParentheses	TRUE for $\langle s, c\rangle$ if any sentence of s mentions c in a phrase surrounded with parentheses
withCode	TRUE for $\langle s, c\rangle$ if any sentence of s mentioning c contains a code snippet
importantSentence	TRUE for $\langle s, c\rangle$ if any sentence of s mentioning c is considers as "`important`"
modal	TRUE for $\langle s, c\rangle$ if in any sentence of s mentioning c, a verb applied to c has modal verb
negation	TRUE for $\langle s, c\rangle$ if in any sentence of s mentioning c, negation is applied to the verb connected to c
inEnum	TRUE for $\langle s, c\rangle$ if in any sentence of s, c is enumerated with more than one other API types, or connected to other API type with "or" conjugation.

isExample This feature describes the cases in which an API type is explicitly mentioned as an example. The *isExample* feature is TRUE if the name of the API type is preceded, in the same sentence, by one of the following phrases: "such as", "for example", or "for instance".

isInParentheses This feature has the same motivation as the *isExample* feature. When an API type is mentioned in parentheses, it also can be an example and carries a secondary

function for the text. The only exception we considered is when the text in parentheses starts with the word "note". When a phrase starts with the word "note", it can contain important information, therefore we made the exception for that case.

withCode As mentioned in the Section 3.3.1, after collapsing code snippets, they can become part of the sentences. This usually happens when an API type is one of the main components of the code snippet. Based on this observation, we introduced the *withCode* feature, which is *TRUE* if an API type appears in the same sentence with the code snippet.

importantSentence A sentence is considered important if it is in imperative mood or it starts with instructive words (to, when, by, try, note, in order). We consider a sentence to be in the imperative mood if any verb of the sentence does not have a subject or its subject is "you" (e.g. in cases where the tutorial directly addresses a reader).

modal and negation These features are identified by parsing the sentence which contains an API type. By using the Stanford Parser, all dependencies between the words are identified. If an API type is the subject or the object of some verb, and that verb is also connected to a modal verb, then the *modal* feature is *TRUE*. If the verb is connected to the word "not" then the *negation* feature is *TRUE*.

inEnum This feature describes the case in which case an API type is mentioned within a list of items. Intuitively, if a sentence contains an enumeration, one element of which is the API type, then the sentence will not contain important information for that particular API type. We consider an API type to be in enumeration if API type is mentioned along with other nouns separated by the word "or", punctuation, or by the word "and" if the number of enumerated words is more than two. The difference for the words "or" and "and" is based on observations we made during the development period.

Dependency-based Features

Table 3.7 presents two features that are calculated with a more detailed analysis of the linguistic properties of API tutorials. The table mentions brief descriptions of the features. The more detailed description can be found in Section 3.3.3.

Table 3.7: Dependency-based Features

Feature	**Description**
depScore	a score representing how relevant or not relevant is *c* for *s* based on the dependencies of *c*. Dependencies of *c* are all phrases in *s* containing *c*. The total score is calculated as the average of scores per dependency.
relScore	a score representing how relevant or not relevant is *c* for *s*, based on the relation types of dependencies of *c*. Relation types of dependencies of *c* are the linguistic relations that connect words to *c* in the phrases involving *c*. The total score is calculated as the average of scores per relation type of a dependency

3.3.3 Dependency-based Features

In this section, typed dependencies and their use are discussed. Typed dependencies are also known as grammatical relations. Words in the sentences relate to each other through syntactical relations (such as subject, object, preposition, etc) whereby the governor is the word which possesses the relation and the dependent is the word to which the relation is applied. For example, in the sentence “Cats eat fish” there are two main dependencies. “Cat” is the subject of “eat” and “fish” is the object of eat. Typed dependencies between words sometimes carry very important information about the role of the phrase and constituent words. Certain dependencies might be indicative of the relevance of a section to a class or not. For example, when a class is the subject of a verb, then it is very likely that class is the main focus of the sentence. We decided to exploit this information for improving the classification results. API tutorials are particularly suitable for exploiting the regularities of

text because they contain repetitive grammatical structures. For examples, three sections in JodaTime tutorial start with a similar phrase (Figure 3.9).

Intervals

An *interval* in Joda-Time represents an interval of time from one instant to another instant. Both instants are fully specified instants in the datetime continuum, complete with time zone.

Intervals are implemented as *half-open*, which is to say that the start instant is inclusive but the end instant is exclusive. The end is always greater than or equal to the start. Both end-points are restricted to having the same chronology and the same time zone.

Two implementations are provided, `Interval` and `MutableInterval`, both are specializations of `ReadableInterval`.

Durations

A *duration* in Joda-Time represents a duration of time measured in milliseconds. The duration is often obtained from an interval.

Durations are a very simple concept, and the implementation is also simple. They have no chronology or time zone, and consist solely of the millisecond duration.

Durations can be added to an instant, or to either end of an interval to change those objects. In datetime maths you could say:

```
instant  +  duration  =  instant
```

Currently, there is only one implementation of the `ReadableDuration` interface: `Duration`.

Periods

A *period* in Joda-Time represents a period of time defined in terms of fields, for example, 3 years 5 months 2 days and 7 hours. This differs from a duration in that it is inexact in terms of milliseconds. A period can only be resolved to an exact number of milliseconds by specifying the instant (including chronology and time zone) it is relative to.

For example, consider a period of 1 month. If you add this period to the 1st February (ISO) then you will get the 1st March. If you add the same period to the 1st March you will get the 1st April. But the duration added (in milliseconds) in these two cases is very different.

Figure 3.9: An Example from JodaTime API Tutorial

To use the idea of typed dependencies in the classifier, first we had to identify positive and negative dependencies and create a database of such dependencies with corresponding weights. We used Java Tutorials to create the database; which contains one of the tutorials from the experimental corpus. Around 1785 dependencies were extracted in which either the governor or the dependent was the code-like term. Afterwards, we annotated each dependency as positive, negative or not useful, depending on the context. Here, a single sentence where the code-like term appeared was considered as context. A typed dependency was labelled as positive or negative if it contributed to the relevance or non-relevance of the code-like term.

Otherwise, if the typed dependency had no contribution for relevance or non-relevance, it was labelled as not useful. The definition of relevance here is the same as during the annotation process discussed in Chapter 4. From 1785 extracted typed dependencies, 841 were classified as positive, 752 as negative, and 615 as not useful. Those 615 triples marked as not useful were ignored and were not used in following steps. The useful dependencies overall contained 246 distinct typed dependencies and 39 distinct relations. As a reminder, typed dependencies are the triples of governor-relation type-dependency such as (cat-subject-eat), while relation type is only the linguistic relation connecting words such as subject, object, adjective, etc.

For each kind of dependency, a weight was calculated based on the annotation results. The intuition for the weight calculation was that the more times a dependency appeared as positive, the larger the weight should be. However the popularity of the dependency should also have a role. The more popular the dependency, the more it should contribute to the final weight, as we can be more confident if we have more evidence. Taking into account this intuition we decided to use the $z - score$ to calculate the weight for each dependency. The $z - score$ is defined as in Eq. 3.5. It measures how many standard deviations away a piece of data is from its mean.

$$Z = \frac{x - \mu}{\sigma} \tag{3.5}$$

To use the $z - score$ as a feature for classification, the normalized version of all scores was calculated. The total score for the section was calculated by taking all dependencies which contain an API type and averaging them.

Table 3.8 contains a few examples of dependencies. The third and fourth lines contain dependencies in which all instances were marked as negative. That means that as percentage of negative cases, those cases are equivalent; however the $z - score$ takes into account the number of occurrence and, therefore, distinguishes these two cases. The $z - score$ for third line is -1.81 as opposed to fourth line, which is -4.42. This is because the dependency which appears 2 times, as in the third line, gives less confidence than the one in the forth line which appears 12 times.

[5] dobjMDneg - object of a verb, where verb is preceded by modal verb and negation word - not

Table 3.8: A Few Examples of Dependencies

Governor	Relation	Dependant	Total N	Positive N	Z-score	Norm.
use	dobjMDneg[5]	clt[6]	5	0	-2.85	0.32
clt	nsubj[7]	specify	11	11	2.6	0.93
catch	dobjMD[8]	clt	2	0	-1.81	0.44
define	prepIN[9]	clt	12	0	-4.42	0.19
use	dobj[10]	clt	88	61	1.42	0.79

However, the percentage of positive or negative cases is also important. For example, the fifth line of the table contains a (use-object-clt) dependency. An example of the sentence containing this dependency can be "If you want to improve the speed of the search operation use HashMap". This dependency appeared in 88 sentences of the JavaTutorial and in 69% of the cases has been marked as positive and in 31% as negative. The $z-score$ for this case would be just 1.42, which is less than the $z-score$ in second line. Although the number of occurrences in the second line is much smaller than in line five, all occurrences of line two were marked as positive, so it has a higher $z-score$.

3.3.4 MaxEnt as a Choice of Classifier

As discussed in Section 2.3.1, MaxEnt and SVM are classification algorithms which were proven successful at a variety of tasks, including text classification. As our choice for classifier, we selected the MaxEnt classifier as it is especially recommended for text classification [23], [18]. In text classification, the number of features can easily exceed the number of data items, leading to sparse features. Nigam et al. [23] showed that for text classification, a basic MaxEnt classifier has poor feature selection but a MaxEnt classifier with priors performs the best. As our experimental corpus is not large we expect the features to be sparse. However, MaxEnt is commonly used with binary features and we have a mix of binary and real-valued features. To check the possible drawbacks of using real-valued features and also

[6] clt - code-like term
[7] nsubj - subject of a verb
[8] dobjMD - object of a verb, where verb is preceded by modal verb
[9] prepIN - prepositional phrase with preposition IN
[10] dobj - object of a verb

perform a basic comparative evaluation of the MaxEnt classifier, we decided to experiment with a classifier other than MaxEnt. The second option for a classifier was SVM, which is not restricted to binary features. SVM was not our first choice of classifier because we have small dataset and Manning et al. [19, p. 308] advise to apply SVM for larger datasets. We used the implementation of SVM provided by the SVMlight package which is called internally through the Stanford Core NLP package. The default configuration was used and no special parameter tuning was performed to adapt the classifier to the observed data.

As real-valued features were one of the questionable cases, we ran the experiment twice: once without real-valued features and once with all the features. For assessing the effectiveness of the classifiers for this experiment and for the following ones the precision, recall and F1 scores are calculated using Leave One Out Cross Validation. As presented in Table 3.9, the F1 score is always better or the same for the MaxEnt classifier. For a few cases, the precision of the SVM classifier is slightly better than MaxEnt but the lower recall makes it less desirable.

This experiment allowed us confirm that MaxEnt is a reasonable choice of classifier by comparing it to an off-the-shelf classifier.

Table 3.9: Comparative results for SVM and MaxEnt classifiers

Tutorial	SVM(All features)			SVM(without RV)			MaxEnt(All features)			MaxEnt(without RV)		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1
Jodatime	0.81	0.57	0.67	0.81	0.57	0.67	0.88	0.73	0.80	0.75	0.70	0.72
Math Library	0.70	0.74	0.72	0.69	0.69	0.69	0.71	0.78	0.74	0.69	0.69	0.69
Collections(Official)	0.77	0.30	0.44	0.50	0.07	0.12	0.68	0.61	0.64	0.69	0.55	0.61
Collections(Jenkov)	0.67	0.05	0.09	0.20	0.02	0.04	0.78	0.76	0.77	0.69	0.69	0.69
Smack	0.75	0.89	0.81	0.74	0.88	0.80	0.84	0.88	0.86	0.82	0.80	0.81

4

Annotating The Experimental Corpus

Annotation is the process of assigning specific labels to data. The process includes a subjective judgement of the data. Therefore, annotation is usually done by multiple annotators. To ensure that multiple annotators understand the task the same way, an annotation guide was developed (Appendix A) which for our case contains the definitions and the examples for "relevant" and "not relevant" cases. The data items to annotate in this case are the pairs formed by an API type and an API tutorial section in which the class appears. The labels to be assigned to data are "relevant" and "not relevant", indicating whether the tutorial section helps to explain the usage of the API type or not. Annotators were asked to mark each appearance of an API type in an API tutorial section as relevant or not relevant according to the annotation guide.

The following sections discuss the annotation tool we developed to support the annotation task, the annotation process of the experimental corpus, and the annotation results.

4.1 Annotation Tool

For annotating the experimental corpus, we created a special annotation tool to assist annotators in their task. As can be seen in Figure 4.1 the annotation tool has the following functionality. It:

- displays each section or subsection with HTML formatting;
- one by one highlights all API types found in the section;

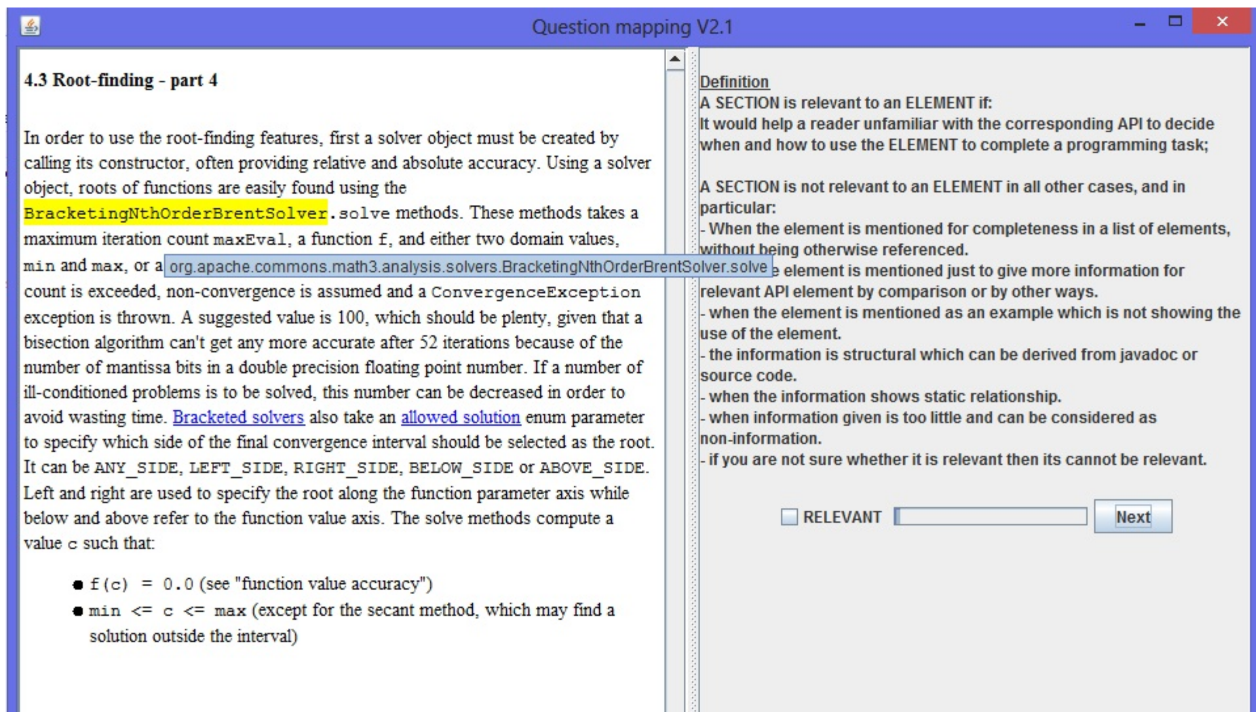


Figure 4.1: Annotation Tool

- displays the fully qualified name of the currently highlighted API type when the user hovers over a tutorial section;
- displays the current progress of the entire annotation task on the progress bar;
- displays in concise form the annotation guide, to help annotators in their decisions;
- provides functionality to save the current state of the annotation and resume after restarting the application;
- does not include a “previous” button on purpose, so that the annotators must follow their first impression and not overthink the task.

4.2 Annotation Process

The annotation process of each portion of data is an *annotation session*. During an annotation session, two participants annotate the same data and afterwards discuss the dis-

agreements. In the beginning of each annotation session, two participants were introduced to the annotation guide and annotation tool. The annotation guide presented in Appendix A contains a definition of relevance and examples of common cases of non-relevance. For developing a detailed annotation guide, first we annotated the JodaTime tutorial. Through the discussion of results and the experience with collaborators, we developed the annotation guide. Each session of annotation began with 10 warm-up examples so that the annotators could get used to the task. This warm-up examples were followed by actual data. The amount of work was divided so that it would not take more than one hour to complete an annotation task. Within one hour after the annotation task, the two annotators discussed their disagreements to reach a common decision for each data item. The disagreement discussion was done in the presence of the author who took notes of the decisions and made sure that those complied with the annotation guide. The initial disagreement was measured with the Cohen-kappa coefficient [34] and is presented in Section 4.3.

4.3 Annotation Results

Each of the five tutorials listed in Chapter 2 was annotated according to the process described above. The annotation of each tutorial was a separate session except Collection(official). As Collection(official) was too big for one session of annotation, it was divided into two parts and was annotated in two sessions. The annotators were the author, her supervisor of current work, a post-doctoral fellow, a PhD student, a Master's student and an intern of the same lab, and also 2 Master's students from two different groups at McGill University.

Table 4.1 presents the size and the initial disagreement Cohen's kappa coefficient of each annotation session. Cohen's kappa coefficient is used to asses the inter-rater agreement. According to Fleiss's [8, Chapter 18] *kappa* over 0.75 can be considered as excellent, 0.40 to 0.75 as fair to good, and below 0.40 as poor agreement beyond chance. In our case results for two annotation sessions are below 0.4. For JodaTime the *kappa* score is 0.39 and for the first session of the Collections(official) it is 0.29. The JodaTime tutorial was used for

Table 4.1: Annotation Results

Tutorial	**N of pairs**	***kappa***
JodaTime	72	0.39
Math library	98	0.51
Collections (Official)	107 113	0.29 0.61
Collections (Jenkov)	150	0.57
Smack	86	0.63

developing the annotation guide and for clarifying the definition for relevance. That is why we accepted the annotation results for JodaTime, though the *kappa* score is a little below the fair agreement.

According to table 4.1, the *kappa* score for Collections(Official) is much below the acceptable agreement score. The main reason was that the annotators initially held radically different views on doubtful cases. This session contained sections describing the `Collections` class, which is a class of static methods. The annotation guide did not include any specific instructions for this case and annotators had to make their subjective decisions. Annotators made well-argumented but still radically different decisions. The other session for Collections(Official) and the session for Collections(Jenkov) contained sections describing the same problematic class `Collections`, but the annotators were more similar in their understanding of the case. This means that the reason of low agreement score was the subjective view of the annotators. The disagreement discussion for the first session of Collections(Official) was extensive. It lasted more than an hour, unlike other sessions, for which it took around 15 minutes. Through long and detailed discussion, the annotators and the author made sure that the best possible decisions were made for each case. Because of the rigorous discussion, we accepted the results despite the low initial agreement score.

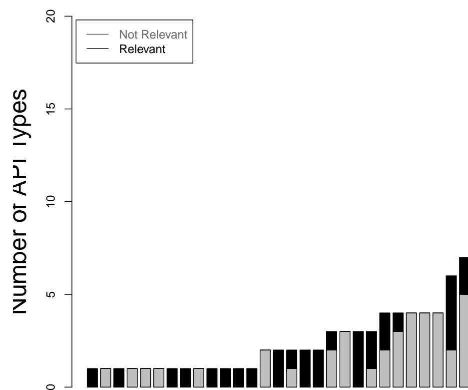
Table 4.2 contains some of the descriptive statistics for the selected tutorials. The second column of the table contains the total number of API tutorial sections and API type pairs preceded by the number of relevant pairs. The third column contains the number of distinct

API types preceded by the number of relevant distinct API types. For example, Collections (Official) has 219 pairs from which only 56 were annotated as relevant. That is, only 25% was found to be relevant. However, the third column shows that of the 58 distinct API types mentioned in the tutorial, 31 had at least one matching relevant section. This means that the tutorial covered 53% of the mentioned API types with useful information.

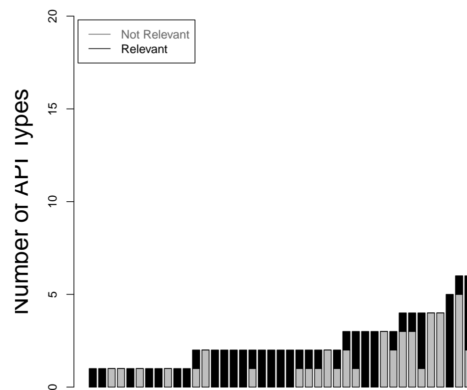
Table 4.2: Tutorial Statistics

Tutorial	Section-Element (Relevant/Total)	Unique API Types (Relevant/Total)
JodaTime	30/68	21/36
Math Library	54/98	45/74
Collections (Official)	56/219	31/58
Collections (Jenkov)	42/150	21/28
Smack	56/86	29/40

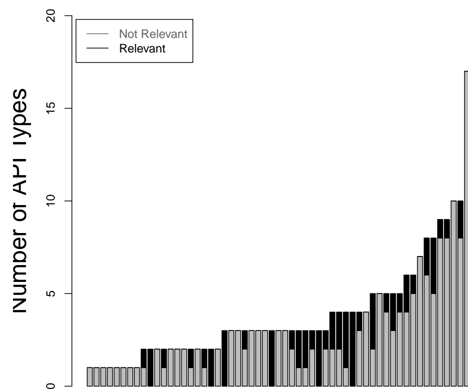
The detailed results of the annotation process can be found in Figure 4.2 and Figure 4.3. Figure 4.2 contains a graph for each tutorial describing the density of relevant and not relevant API types in the section, where each section is represented by a bar in the graph. In Figure 4.3 each bar shows the number of sections containing an API type, the darker part corresponds to the number of relevant cases and the lighter part corresponds to not relevant cases.



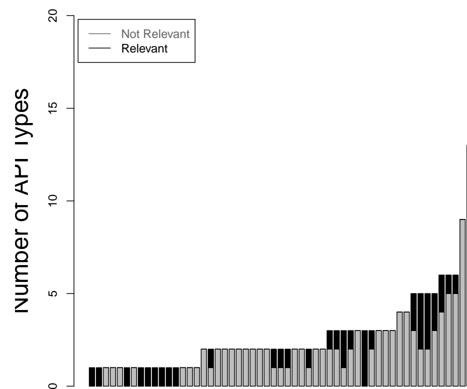
(a) JodaTime



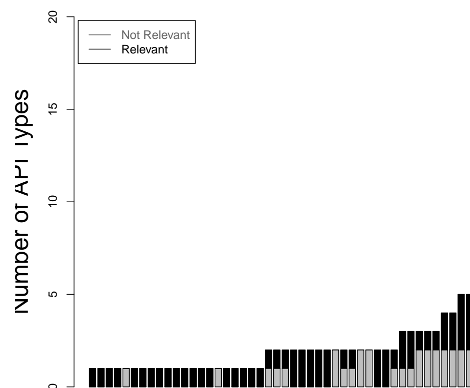
(b) Math Library



(c) Collections (Official)

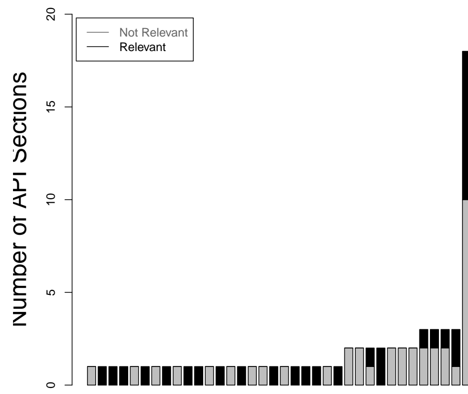


(d) Collections (Jenkov)

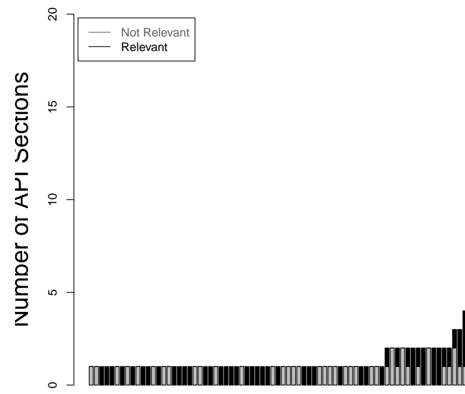


(e) Smack

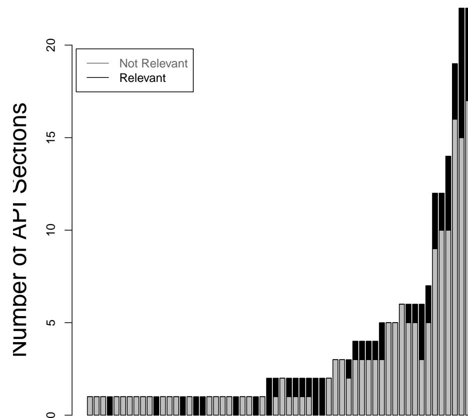
Figure 4.2: Number of Relevant API Types per Section



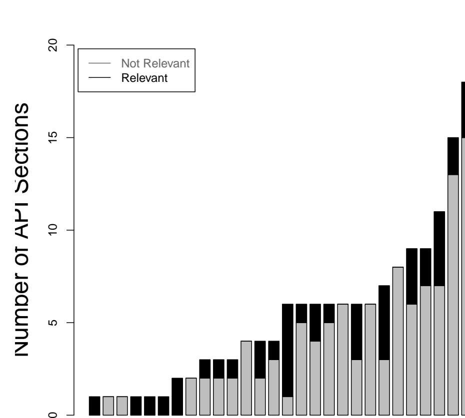
(a) JodaTime



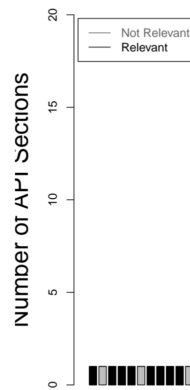
(b) Math Library



(c) Collections (Official)



(d) Collections (Jenkov)



(e) Smack

Figure 4.3: Number of Sections with Relevant API Type

5

Classification Results and Experiments

5.1 Classification Results

For evaluating the performance of the system for each tutorial, we performed leave-one-out cross validation (LOOCV). For each API tutorial section-class pair, the classifier was trained on the rest of the tutorial and tested on the held-out pair. Results are accumulated for the tutorial, and precision, recall, and F1 scores are calculated. The results are presented in Table 5.1.

Table 5.1: LOOCV Results for All Tutorials

Tutorial	**P**	**R**	**F1**
JodaTime	0.81	0.73	0.77
Math Library	0.69	0.74	0.71
Collections (Official)	0.71	0.62	0.67
Collections (Jenkov)	0.84	0.76	0.80
Smack	0.87	0.80	0.83

From the table we can see that the Math Libary and Collections(offical) tutorials are the hardest ones to classify. For the Math Library tutorial, out of 31 incorrect classifications (false positives + false negatives), 18 cases had very few features and probabilities for both "relevant" and "not relevant" categories were between 40-60. That means that the probability of both categories was close to 50%. To overcome this problem, more features can be introduced to better distinguish between the "relevant" and "not relevant" categories.

In the case of Collections(Official), the cases with few features were only 5 out of 35 incorrectly classified cases. Collections(Official) differs from other tutorials through its large ratio of negative cases. In this case, $class - feature$ (described in Section 2.3.4) has a large negative value. Therefore, more positive evidence is needed to classify the data into the positive category. That explains the low recall value for Collections(Official).

It might have been possible to improve performance of the classifier in case of unbalanced datasets if we sample it to have at least a 2:1 ratio of positive and negative examples. However, taking into account the scarcity of the data we decided to leave it for future work when bigger datasets will be available.

The other problem common for Math library and Collections(Official) is that some of the sentence level features had very little weight. For the MaxEnt classifier, the weights of the features are learned based on number of times they occur for "relevant" or "not relevant" data items. If a feature occurs a few times, then it is not going to have a large weight. In other cases, features like $importantSentence$ can a have small but negative weight, while a $negation$ feature may have a positive weight. Surely, a large number of negative cases can decrease the weight for intuitively positive features, like $importantSentence$. Also, having a bigger dataset with more occurrences of features would improve and stabilize the weights of the features. However, the problem here is that for sentence level features it is not obvious how those should be generalized for a section.

For example, if the same section contains an API type in an enumeration (which enables the $inEnum$ feature) and also mentions the API type twice with a modal verb (which enables $modal$ feature) then should both features be present, or should the features be combined? The question is how to accumulate and average binary features for a section. Moreover, the longer the section gets, the higher is the chance that sentence level features will appear. Math Library and Collections(Official) are the tutorials with the biggest average section lengths.

A somewhat similar problem was mentioned by Turney [33]. The author estimated the total polarity of reviews as an average of the polarities of a review's phrases. This idea worked for product and car reviews, but failed for movie reviews. Movie reviews appeared

to contain contradictory information, irony, etc. As in the case of movie reviews, here also it is not clear how negative evidence and positive evidence should be combined.

A further analysis of the effects of group of features can be found in Section 5.2

It is also interesting to look at the results per API type. Table 5.2 presents coverage information per API type. The second column of the table shows the number of distinct API types mentioned in the tutorial. The third column shows the distinct number of API types that had at least one relevant section according to the annotation. The number of distinct API types which had both an annotated relevant section and have been recommended is presented in the fourth column. The last column shows the distinct number of API types which did not have any relevant sections but were recommended by the classifier.

Table 5.2: Coverage Provided by Classification per API Element

Tutorial	N of API T.	Relevant	Recommended	Falsely Recommended
Joda	36	21	17	3
Math	74	45	30	17
Collections (Offical)	58	31	22	7
Collections (Jenkov)	28	21	16	5
Smack	40	29	22	5

The precision and recall per API type seems an interesting information as well. However, those results are skipped because as API types have low frequency, precision and recall are not that informative.

5.2 Results for Different Set of Features

Features for the classifier can be divided into 4 groups (tutorial level(T), section level(SC), sentence level(ST) and real-valued(RV) features) plus two additional features (dependency(D) and relation(R)-based features), which we considered as separate groups of features. In this section we explore the effect of each group of features on the classification results.

Table 5.3 presents the results of LOOCV for each tutorial by different sets of features. The first six rows are the classification results for separate feature groups. Afterwards, features are added group by group, in decreasing order of detail. First, tutorial level features and real-valued features, then section level, sentence level, dependency features, relation features and then all features together.

Table 5.3: Classification Results for Different Set of Features

Tutorial	JodaTime			Math Library			Collections (Official)			Collections (Jenkov)			Smack		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
R	0.70	0.70	0.70	0.65	0.78	0.71	0.72	0.32	0.44	0.65	0.48	0.55	0.70	0.96	0.81
D	0.67	0.67	0.67	0.67	0.69	0.68	0.40	0.07	0.12	0.68	0.40	0.51	0.70	0.96	0.81
ST	0.76	0.63	0.69	0.76	0.54	0.63	0.55	0.20	0.29	0.61	0.83	0.71	0.69	0.91	0.78
SC	0.66	0.63	0.64	0.65	0.81	0.72	0.62	0.43	0.51	0.56	0.45	0.50	0.91	0.77	0.83
T	0.72	0.43	0.54	0.61	0.89	0.72	0.33	0.04	0.06	0.62	0.31	0.41	0.70	0.96	0.81
RV	0.77	0.77	0.77	0.58	0.78	0.67	0.50	0.21	0.30	0.79	0.52	0.63	0.68	0.91	0.78
RV,T	0.84	0.87	0.85	0.56	0.65	0.60	0.62	0.23	0.34	0.84	0.62	0.71	0.73	0.73	0.73
RV,T,SC	0.82	0.77	0.79	0.62	0.74	0.68	0.66	0.48	0.56	0.83	0.81	0.82	0.85	0.79	0.81
RV,T,SC,ST	0.81	0.70	0.75	0.70	0.69	0.69	0.70	0.55	0.62	0.85	0.79	0.81	0.87	0.80	0.83
RV,T,SC,ST,D	0.81	0.70	0.75	0.68	0.74	0.71	0.68	0.54	0.60	0.82	0.79	0.80	0.86	0.79	0.82
RV,T,SC,ST,R	0.81	0.73	0.77	0.73	0.76	0.75	0.68	0.61	0.64	0.84	0.76	0.80	0.85	0.79	0.81
All	0.81	0.73	0.77	0.69	0.74	0.71	0.71	0.62	0.67	0.84	0.76	0.80	0.87	0.80	0.83

According to these results, one of the weakest groups of features is the group of tutorial level features. This can be explained by the small number of features in the group(only two binary features). However, real-valued features combined with tutorial level features already shows improvement for the majority of tutorials.

Collections (Official) is the most difficult task according not only to the performance for all features, but also for separate groups of features. The same relation can be observed for the Smack tutorial, which has the overall highest performance, and accordingly, separate groups of features have the highest performance among other tutorials. In other words,there is no dominant or weak feature group. If a tutorial is well-served by our choice of features then all features work well, and vice versa. It is also worth mentioning that adding features based on their level of detail usually improves the performance. The only level of features which causes problems, for example for JodaTime, is sentence level features. The problem with sentence level features is discussed in the previous section.

Another interesting observation is the effect of the dependency and relation features. Including dependency features, in the case of JodaTime and Math Library, improves recall and pulls down the precision. In the case of the other three tutorials, both scores go down. In contrast, the addition of the relation feature always improves or does not change precision and recall. Surprisingly, the combination of these two always improves or does not change the performance. For example, for Collections (Official) the addition of the dependency feature brings down both precision and recall. However, the dependency feature combined with the relation feature improves performance, compared to using the feature sets without dependency.

In this experiment we asses the performance of the classifier by increasingly adding more and more features without changing the training size. In this experimental setting there is a potential for overfitting. However based on the results presented in the Table 5.3 we can see that it is not the case. The MaxEnt classifier with the proper configuration is able to avoid overfitting.

5.3 Cross-Tutorial Testing

One of the research questions was whether features can generalize across the five tutorials we studied. To answer this question, we set up an experiment to train a model on four tutorials and test on the fifth one. Table 5.4 presents the results of this experiment, where each line corresponds to the case in which a tutorial was used as testing and the other tutorials were used for training. Precision, recall and F1 score were calculated by using the accumulated false positives, true positives, false negatives, true negatives for all API tutorial section-class pairs from a given tutorial.

The main tendency that can be noticed is that recall is overall lower if we compare with recall for the LOOCV results. However, for JodaTime, Math library, Collections (Official) precision is improved and for Smack it stayed the same. One of the main reasons for the observed changes was the difference in positive and negative examples ratios between training

Table 5.4: Cross Tutorial Testing Results

Test Tutorial	Precision	recall	F1
Jodatime	0.94	0.57	0.71
Math Library	0.87	0.48	0.62
Collections (Official)	0.74	0.76	0.75
Collections (Jenkov)	0.80	0.68	0.73
Smack	0.87	0.64	0.74

and testing sets. For example, for JodaTime the number of positive examples is almost the same as the number of negative examples, but in the training set for JodaTime the percentage of positive examples is around 35%. As a result the negative class has a slightly higher weight. Therefore, fewer sections are classified as relevant, which lowers the recall, but improves the precision. One possible reason for the observed changes can also be the different training set sizes for each of the tutorial, but this hypothesis needs more exploration.

5.4 Learning Curves

To better understand the limitation of a small corpus size and its effect on the classification results, we conducted a separate experiment. All tutorials were merged together forming one corpus of 621 API tutorial section-class pairs. The corpus was shuffled and split into 120 test pairs and 501 training pairs. Afterwards, the experiment was run for different sizes of training set from 1 to 521. For each experiment, five performance measures were calculated for both the test set of 120 pairs and the used training set. The resulting lines are called learning curves and are presented in Figure 5.1.

Figure 5.1(a) shows the percentage of wrongly classified pairs for training and testing sets, for different size training sets. As the training set grows the overfitting decreases, taking the testing error down and training error up. The straight line represents an arbitrary performance threshold of 0.80 for the classifier. Although there is a gap between the training and testing errors, the performance is stable for the last 100 examples. This means that most probably adding one or two tutorials to the corpus will not improve classifier results

significantly.

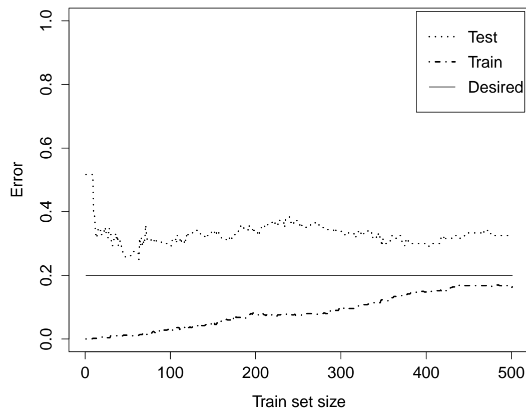
For better diagnosing the system, learning curves for precision, recall, F1, and false positive rate (FPR) are presented. Figure 5.1(b) presents the precision change for training and testing, sets for various training set sizes. Approximately after the training set size exceeded 350 examples, the precision for both testing and training sets reached the desired performance and remained constant. Figure 5.1(d) presents similar results for a different measure called false positive rate (FPR), also known as false alarm ratio. This is a measure for describing cases which were wrongly classified as relevant. The FPR is slightly above 0.1, which means that only 10% of pairs that are not relevant were mistakenly classified as relevant. These are good results showing that the final classifier labelings are not very noisy. However, Figure 5.1(c) shows low results for recall. Recall is the percentage of right classifications among all relevant examples. Low recall means that there are more relevant examples which were not classified as relevant. The recall for both training and testing sets are lower than desired. However, note that for our particular application, precision is always more important than recall. It is more important to have a low percentage of noise in the classification results, even if this implies missing potentially useful sections.

Based on this experiment, we conclude that for improving the performance of our system, increasing the corpus size might not help considerably. Instead, as a first step for improving the system, more or better features are necessary.

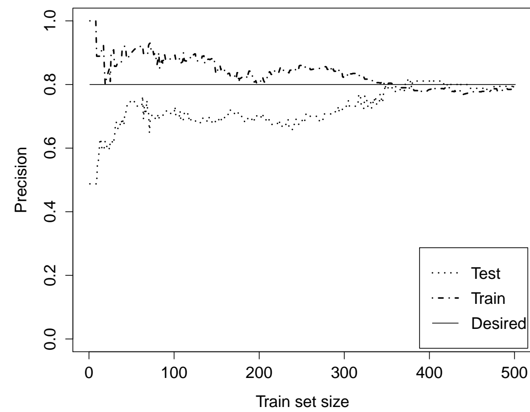
5.5 IR Comparison

We use a MaxEnt classifier for determining relevant sections of an API tutorial for a particular API type. However, it remains an open question whether it is possible to achieve similar results by using existing IR techniques. To explore this question, we conducted the following experiment. Intuitively, the more common words exists between a section and an API type description (e.g. JavaDoc), the more similar is the section to the API type description, and thus, the more focused is the section on the API type. Therefore, the similarity measure

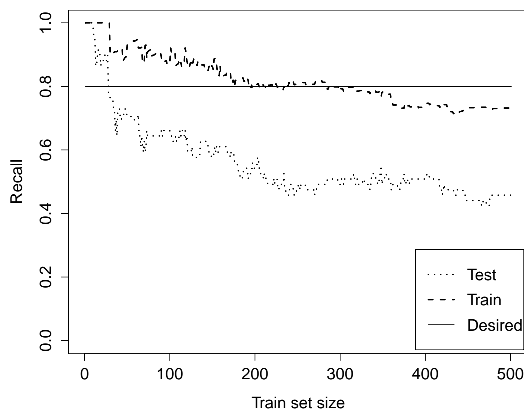
(a) Percentage of wrongly classified cases of training and testing sets



(b) Precision for train and test sets



(c) Recall for train and test sets



(d) FPR for train and test sets

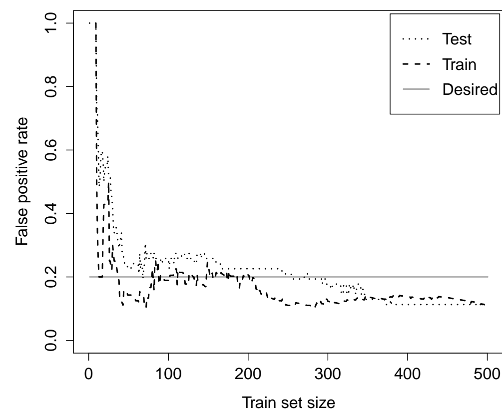


Figure 5.1: Learning Curves

between an API tutorial section and an API type description is calculated. As a measure of similarity, the well-known cosine similarity described in Section 2.3 is used. API tutorial sections and API documentations are considered as a document corpus. For each API tutorial section, the similarity between the section text (excluding code snippets) and the API type documentation is calculated.

We consider an API type relevant if the similarity value is higher than a certain threshold. The algorithm is based on threshold and is not top-n-based because the number of relevant API types per section varies a lot and can even be 0. For each tutorial, a threshold is

Table 5.5: MaxEnt vs. CosSim

Tutorial	MaxEnt			CosSim		
	P	R	F1	P	R	F1
JodaTime	0.81	0.73	0.77	0.73	0.73	0.73
Math	0.69	0.74	0.71	0.67	0.65	0.66
Collections (Offical)	0.71	0.62	0.67	0.30	0.94	0.45
Collections (Jenkov)	0.84	0.76	0.80	0.33	0.88	0.48
Smack	0.87	0.80	0.83	0.74	0.52	0.61

calculated according to the following description. For each API type c_i, the top n_i most similar sections were retrieved, where n_i is the number of relevant sections for c_i according to the annotation results. The smallest similarity value of the retrieved sections for all API types is averaged and considered as a threshold afterwards. Based on the calculated threshold, sections were selected for each API type. As retrieval was done per API type, precision, recall, and F1 scores are calculated for each API type. Afterwards, calculated per API element precision, recall, and F1 scores are averaged and are presented in Table 5.5.

As can be seen, the similarity technique fails, especially in the case of many not relevant examples, such as in the Collections (official) and Collections (Jenkov) tutorials. However, generally the performance of similarity measure is not that bad. Surely it cannot be used as an independent solution for our problem but the results can be used as part of some solution.

6

Conclusion and Future Work

6.1 Conclusion

In this work, we suggested a three-step technique for discovering API tutorial sections that may help explain API types. The experimental corpus contained five tutorials, where two tutorials were used for development and three were used for testing. Moreover, we showed that it is possible to get meaningful results having just small amount og data from which to learn. When a classifier was trained and tested on the same tutorial using One Leave Out Cross validation, the achieved precision was 0.69-0.81 and recall 0.62-0.80. In case when a classifier was trained on four out of five experimental tutorials and tested on the fifth tutorial, precision varied between 0.74-0.94, and recall was 0.48-0.76. The results are promising and show good generalization for unseen tutorials. However, it is not possible to claim that the technique will have comparable results for any unseen tutorial. The results show that the Math Library tutorial and Collections(Official) tutorial are harder to classify because of lack of positive features.

The main goal of this work was to show that NLP features can be used along with structural properties of the text for discovering tutorial sections relevant to an API type. The experiment with different groups of features showed the success of NLP features, as by adding more detailed features, the performance of the classifier increased.

A special remark needs to be given also for dependency features. Although a relatively large corpus was used for acquiring the dependencies, these were taken only from one source

and were annotated only by the author. That is why the database of dependencies is not complete and the quality is yet to be verified. However, the technique of using dependencies was overall successful. The experiments showed improvement of the classification results after dependency and relation features were integrated.

One of the main limitations of the proposed system is the quality of API tutorials. However, the suggested technique can be used to combine and link multiple resources of the same API. The third and the main step of the solution can be used for classifying different resources containing textual information.

6.2 Future Work

In order to keep the problem tractable, some assumptions had to be made about the granularity of API types, the role of linguistic properties of the text, tutorial structure, and the knowledge of the user. In future work, we would like to better understand the effect of these decisions for the improvement of our system.

Considering the class instead of its methods and fields was one the simplifications. It was justified by the fact that usually not a method, but rather a group of them solves a programming task. For the case of static methods, this assumption might not hold. However, handling this problem by decreasing the granularity level to methods and fields would not be right either. In the future, it would be better if a distinction is made between task-solving methods and assisting methods. Afterwards, this information could be used by the classifier. This is a complex problem and it would deserve separate research.

One of the paths taken in this study was to explore the information in the text of the tutorial and to exploit linguistic properties of the text. That is why code snippets were not considered in depth. However, we believe that code snippets contain valuable information which, if added to a classifier, would improve its performance.

Another assumption made in this work was that tutorial sections are independent. That is clearly not true. Sections are logical continuations of each other. Therefore, it would

be interesting to use that information and measure its effect on the classification results. Moreover, the pairs of API tutorial sections and API types were considered independent as well. If not,the information of previously classified pairs could be used by a classifier when making a decision for the current one. This could be done similarly to sequential models used in NLP [30], but the effectiveness of it is a new research question.

The system proposed in this work can recommend tutorial sections for API types explicitly selected by the programmer. However, the programmer might not know the exact API type needed to solve the programming task. We believe that after collecting a reasonable corpus of API resources, a natural language query can be used to find a section addressing the programming task.

Bibliography

[1] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? In *Proceedings of the 28th International Conference on Software Engineering*, pages 361–370, 2006.

[2] B. Ashok, Joseph Joy, Hongkang Liang, Sriram K. Rajamani, Gopal Srinivasa, and Vipindeep Vangala. DebugAdvisor: a recommender system for debugging. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 373–382, 2009.

[3] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*. Springer New York, 2006.

[4] Kalina Bontcheva and Marta Sabou. Learning ontologies from software artifacts: Exploring and combining multiple sources. In *Proceedings of Workshop on Semantic Web Enabled Software Engineering*, 2006.

[5] Yam Bahadur Chhetri. Classifying and recommending knowledge in reference documentation to improve API usability. Master's thesis, McGill University, 2012.

[6] Barthélémy Dagenais and Martin P Robillard. Recovering traceability links between an API and its learning resources. In *Proceedings of the 34th IEEE/ACM International Conference on Software Engineering*, pages 47–57, 2012.

[7] James Fan, Aditya Kalyanpur, David Gondek, and David Ferrucci. Automatic knowledge extraction from documents. *IBM Journal of Research and Development*, 56(3.4):5–1, 2012.

[8] Joseph L. Fleiss, Bruce Levin, and Myunghee Cho Paik. *Statistical Methods for Rates and Proportions*. Wiley Series in Probability and Statistics. Wiley, 2013.

[9] Katerina T. Frantzi, Sophia Ananiadou, and Jun-ichi Tsujii. The C-value/NC-value Method of Automatic Recognition for Multi-Word Terms. In *Proceedings of the 2nd European Conference on Research and Advanced Technology for Digital Libraries*, pages 585–604, 1998.

[10] Zachary P. Fry, David Shepherd, Emily Hill, Lori Pollock, and Vijay K. Shanker. Analysing source code: looking for useful verb-direct object pairs in all the right places. *Software, IET*, 2(1):27–36, 2008.

[11] Kevin Gimpel, Nathan Schneider, Brendan O'Connor, Dipanjan Das, Daniel Mills, Jacob Eisenstein, Michael Heilman, Dani Yogatama, Jeffrey Flanigan, and Noah A. Smith. Part-of-speech tagging for twitter: annotation, features, and experiments. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics*, pages 42–47, 2011.

[12] Stefan Henß, Martin Monperrus, and Mira Mezini. Semi-automatically extracting FAQs to improve accessibility of software development knowledge. In *Proceedings of the 34th International Conference on Software Engineering*, pages 793–803, 2012.

[13] Raphael Hoffmann, James Fogarty, and Daniel S Weld. Assieme: finding and leveraging implicit references in a web search interface for programmers. In *Proceedings of the 20th ACM Symposium on User Interface Software and Technology*, pages 13–22, 2007.

[14] Frederick Jelinek. *Statistical methods for speech recognition*. MIT press, 1997.

[15] Aleksander Kolcz, Vidya Prabakarmurthi, and Jugal Kalita. Summarization as feature selection for text categorization. In *Proceedings of the 10th international conference on Information and knowledge management*.

[16] Walid Maalej and Martin P Robillard. Patterns of Knowledge in API Reference Documentation. *IEEE Transactions on Software Engineering*, 39:1264–1282, 2013.

[17] Robert Malouf. A comparison of algorithms for maximum entropy parameter estimation. In *Proceedings of the 6th Conference on Computational Natural Language Learning*, pages 49–55, 2002.

[18] Christopher Manning and Dan Klein. Optimization, maxent models, and conditional estimation without magic. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology: Tutorials*, pages 8–8, 2003.

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

[20] Christopher D. Manning and Hinrich Schütze. *Foundations of statistical natural language processing*. MIT Press, 1999.

[21] Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a large annotated corpus of English: the penn treebank. *Computational Linguistics*, 19(2):313–330, 1993.

[22] Seyed Mehdi Nasehi, Jonathan Sillito, Frank Maurer, and Chris Burns. What makes a good code example?: A study of programming Q&A in StackOverflow. In *Proceedings of the 28th IEEE International Conference on Software Maintenance*, pages 25–34, 2012.

[23] Kamal Nigam, John Lafferty, and Andrew McCallum. Using maximum entropy for text classification. In *IJCAI-99 workshop on machine learning for information filtering*, volume 1, pages 61–67, 1999.

[24] Jorge Nocedal and Stephen J Wright. *Numerical optimization*, volume 2. Springer New York, 1999.

[25] Bo Pang, Lillian Lee, and Shivakumar Vaithyanathan. Thumbs up?: sentiment classification using machine learning techniques. In *Proceedings of the ACL-02 Conference on Empirical Methods in Natural Language Processing*, pages 79–86, 2002.

[26] Martin F. Porter. An algorithm for suffix stripping. In Karen Sparck Jones and Peter Willett, editors, *Readings in information retrieval*. 1997.

[27] Peter C. Rigby and Martin P. Robillard. Discovering essential code elements in informal documentation. In *Proceedings of the 25th International Conference on Software Engineering*, pages 832–841, 2013.

[28] Martin P. Robillard. What makes APIs hard to learn? Answers from developers. *IEEE Software - Special Issue on the Collaborative and Human Aspects of Software Engineering*, 26(6):27–34, 2009.

[29] Jeffrey Stylos and Brad A Myers. Mica: A web-search tool for finding API components and examples. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 195–202, 2006.

[30] Charles Sutton and Andrew McCallum. An introduction to conditional random fields for relational learning. *Introduction to Statistical Relation Learning*, pages 142–146, 2007.

[31] Kristina Toutanova, Dan Klein, Christopher D. Manning, and Yoram Singer. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology*, pages 173–180, 2003.

[32] Kristina Toutanova and Christopher D. Manning. Enriching the knowledge sources used in a maximum entropy part-of-speech tagger. In *Proceedings of the Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora:*

held in conjunction with the 38th Annual Meeting of the Association for Computational Linguistics, pages 63–70, 2000.

[33] Peter D. Turney. Thumbs up or thumbs down?: semantic orientation applied to unsupervised classification of reviews. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, pages 417–424, 2002.

[34] Uchila N. Umesh, Robert A. Peterson, and Matthew H. Sauber. Interjudge Agreement and the Maximum Value of Kappa. *Educational and Psychological Measurement*, 49:835–850, 1989.

[35] Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *Proceedings of the 30th International Conference on Software Engineering*, pages 461–470, 2008.

[36] Ingmar Weber, Antti Ukkonen, and Aris Gionis. Answers, not links: extracting tips from yahoo! answers to address how-to web queries. In *Proceedings of the 5th ACM International Conference on Web Search and Data Mining*, pages 613–622, 2012.

[37] Xusheng Xiao, Amit Paradkar, Suresh Thummalapenta, and Tao Xie. Automated extraction of security policies from natural-language software documents. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 12:1–12:11, 2012.

[38] Alexander Yates, Michael Cafarella, Michele Banko, Oren Etzioni, Matthew Broadhead, and Stephen Soderland. TextRunner: open information extraction on the web. In *Proceedings of Human Language Technologies: The Annual Conference of the North American Chapter of the Association for Computational Linguistics: Demonstrations*, pages 25–26, 2007.

[39] Tong Zhang and Frank J. Oles. Text categorization based on regularized linear classification methods. *Information Retrieval*, 4(1):5–31, 2001.

[40] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. Inferring resource specifications from natural language API documentation. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 307–318, 2009.

A

Annotation Guide

RELEVANCE: A SECTION is relevant to an API ELEMENT if: It would help a reader unfamiliar with the corresponding API to decide when or how to use the ELEMENT to complete a programming task;

NON RELEVANCE:

1. When the element is mentioned for **completeness in a list of elements**, without being otherwise referenced. Example:

   Introduction

   ... In particular, we cover the usage of the key DateTime, Interval, Duration and Period classes...

2. When the element is mentioned just to **give more information/supplement for relevant** API type by comparison or by other ways. Example:

   Accessing Fields

   ... For a complete reference, see the documentation for the base class AbstractReadableInstantFieldProperty....

3. when the element is mentioned as an example which is not showing the use of the element. Example:

   Fields

. . . So, for instance, the 'day-of-year' calendar field would be retrieved by calling the getDayOfYear() method...

4. The information is **structural** which can be derived from Javadoc or source code. Example:

   Querying DateTimes

   int iDoW = dt.getDayOfWeek(); where iDoW can take the values (from class DateTime-Constants). *public static final int MONDAY = 1; public static final int TUESDAY = 2; ...*

5. when the information shows a **static relationship**. Example:

   Date fields

   The DateTime implementation provides a complete list of standard calendar fields: dt.getEra(), dt.getYear(), dt.getWeekyear() ...

6. when too little information is given which is can be considered as **non-information**. Example:

   Querying DateTimes

   ...For instance, the direct way to get the day of week for a particular DateTime, involves calling the method int iDoW = dt.getDayOfWeek(),

7. if **not sure** whether it is relevant, then it cannot be relevant. Example:

   Instants

   ..Within Joda-Time an instant is represented by the ReadableInstant interface...Other implementations of ReadableInstant include Instant and DateMidnight

B

Multi-word concepts

jar file	java se	java web start	java programming language
java sound api	text field	example index	java web start application
web start	launch button	java platform	scroll pane
java web	java se development	text area	java sound
java programming	combo box	java virtual machine	internal frame
source code	method purpose	code snippet	split pane
jar files	jnlp file	swing components	layout manager
start application	virtual machine	java se development kit	programming language
java db	data source	text component	audio data
xml document	layered pane	application program	key value
web page	file system	command line	content pane
described notes	netbeans ide	deployment toolkit	xml schema
source file	menu item	progress bar	programming language keyword
file chooser	data type	main method	text pane
text fields	policy file	check box	color chooser
remote object	regular expression	class path	root pane
development kit	xml file	service provider	tool bar
tool tip	java network launch protocol	full-screen exclusive mode	security manager
radio buttons	java compiler	java 2d	java language
public key	javascript code	java network launch	action listener
formatted text	inner class	class name	tool tips
bidirectional text	internal frames	java runtime environment	menu bar
dialog box	java plug-in	progress indicator	lambda expression
java runtime	layout managers	text components	class files
first argument	xml data	coffee break	midi messages
splash screen	input string	applet tag	type parameters
try block	java plug-in software	progress monitor	menu items
java web start applications	java look	sound file	type parameter

formal parameter	anonymous classes	public key certificate	sample program
microsoft windows	current directory	list selection	java class
nested classes	lambda expressions	code fragment	key bindings
dynamic tree	jdbcrowset object	solaris os	try-with-resources statement
property change listener	inner classes	class file	glass pane
java network	popup menu	anonymous class	standard midi
ldap server	abstract class	assistive technologies	java application
regular expressions	symbolic link	standard output	system properties
top-level container	focus traversal	return type	midi file
default settings	formatted text fields	jdbc driver	change listener
editor pane	ldap v3	default value	launch protocol
combo boxes	event listeners	path environment variable	full path
javascript interpreter	properties files	mouse wheel	user interface
code example	exclusive mode	ldap service	loading progress indicator
midi specification	local variables	byte streams	primitive type
window decorations	control flow	operating system	start applications
string argument	manifest file	different types	rich internet
path environment	tool tip text	file name	runtime environment
java applet	security properties	corresponding adapter class	parameterized type
source files	properties object	error message	mouse events
list data	api documentation	java tutorials	radio button
customized loading progress indicator	java web start software	containment hierarchy	system administrator
jndi tutorial	local classes	structured type	standard midi file
memory consistency errors	tree expansion	image icon	security properties file
java 2d api	standard edition	port number	tree model
sample code	keyboard focus	active directory	broken links
sax parser	terminal window	java applets	default setting
resource bundle	print dialog	window listener	document object model
deployment tool	complete code	miscellaneous fixes	default button
output stream	midi data	exception handler	sql statements
java content tree	java se platform	file format	input stream
text file	mouse button	scroll bar	focus traversal policy
primitive types	control flow statements	jdbc api	builder tool
remote objects	swing trail	return value	symbolic links
key bytes	class variables	check boxes	generic type
text editor	jre software	layout management	enum type
data format	configuration file	property editor	xml content
web server	new connection dialog	java collections framework	upper left corner
modal dialog	logical name	current position	user input

example running
static method
java content
background task
scroll panes
rich internet applications
data types
public key bytes
java program
next generation java plug-in
coffees table
sound api
compile-time error
java collections
undoable edit listener
list selection listener
tree expansion listener
public string
editable combo box
base name
full path name
main class
double d
demo application
focus owner
standard midi files
unicode code
column headers
document filter
abstract classes
previous match
private key
oracle directory server
modeless dialog box
bicycle class
common problems
midi message
item event
new material
policy entry
time stamps
loading progress
example program
data model
java development
focus cycle
memory consistency
primitive data types
reference implementation
no-argument constructor
tool bars
nested class
rectangle class
instance variables
selection mode
editor kit
when input
generic method
locale object
generic types
midi wire protocol
invisible components
abstract methods
remote interface
java tutorial
root node
custom renderer
java console
root element
catch block
graphical user
tree-will-expand listener
focus traversal keys
method invocation
object class
jdbcrowset objects
log file
cancel button
directory structure
initial value
display mode
type map
target data
special characters
xslt examples
properties file
java development kit
internal frame listener
mouse listener
jtextcomponent class
maximum values
java control
javafx ui
boolean argument
default file
statement object
background color
useful methods
password field
event handler
other objects
auto-commit mode
more details
string value
document listener
static fields
swing text
command-line arguments
new connection
same directory
single character
class declaration
embedded flag expression
object reference
java technology
other types
default values
default behavior
root panes
left corner
local class
time stamp
white space
first line
scroll bars
unicode code point
grouplayout layout manager
string builder
caret listener
end user
accessible object
editor panes
default constructor
text panes
sql statement
instance method
display area
byte array
custom mapping
default file system
maximum sizes
ldap provider
pooled connection
start software
apache ant
unicode characters
html page
jlayer class
connection object
directory server
event handlers
datasource class
internal frame events
enum types
different kinds
system property
home directory
class hierarchy
first time
first row

supplementary character
remote method
sqlxml object
mbean server
default properties
selection model
java objects
further information
class-path header
particular type
preview panel
stylizer sample
key certificate
javafx ui controls
unicode locale
rich internet application
adapter class
printable object
datasource objects
editable text
object-oriented programming concepts
java code
text areas
physical fonts
host name
functional interface
formal parameters
uneditable combo
audio data format
garbage collection
midi channel
method description
upper bounded wildcard
task interface
box class
files class
java type
select statement
sound files
first column
new data
component class
toggle button
input map
text layout
java applications
deque instance
default locale
additional information
resize weight
init method
jaxb binding
tree model listener
jdbc tutorial
value pairs
preparedstatement object
custom icon
command-line argument
data file
method declaration
modeless dialog
custom painting
editable combo
character streams
focus cycle root
new project
jaxb annotations
string s
new row
data line
midi input port
java console log
format method
listener api
tray icon
happens-before relationship
following steps
last name
xml parser
watch service
action object
static nested class
ui delegate
remote interfaces
import javafx
simple example
paintcomponent method
new line
embedded flag
supplementary characters
java cache viewer
java control panel
box layout
file type
midi input
audio input
application programs
column header
numeric value
java api
item events
carriage return
input verifier
new service
variable name
context instance
image file
intrinsic lock
web site
jaxb binding compiler
static factory method
dictionary service
unicode character
compound messages
mouse-motion listener
search criteria
first step
local variable
schema definition
package name
stored procedures
example code
midi wire
easiest way
join framework
latest version
accessory component
worker thread
object-oriented programming
extensible stylesheet language
list selection events
file store
parent class
return values
mouse motion
split panes
class method
new operator
primary key
per-pixel translucency
ldap entry
following step
demo applet
factory method
value value
reference type
enum constants
source object
singleton design pattern
windows active directory
new page
parent component
components lesson
security environment
data directory
other kinds

oracle directory
type inference
last row
first application
last character
data transfer
raw type
network interface
joinrowset object
stringbuilder class
database connection
source data
authentication information
language code
throwable class
selection changes
thread interference
user types
first character
different ways
layered panes
result sets
java application launcher
formal type parameter
integer arguments
native methods
deployment options
graphical user interface
simple name
table lists
tree nodes
button text
general information
audiosystem class
try statement
start button
loan amount
test harness

non-unicode text
option pane
java vm
table coffees
runtime exception
static field
property-change listener
tree node
int arguments
generic methods
static factory
mac os
x alignment
chooser panel
schema type
event example
member variables
xslt directory
collection views
rectanglearea class
authentication mechanism
static string
java secure socket
java language specification
standard mbean
client program
complete list
key binding
code excerpt
mouse event
jnlp api
window system
character stream
pooled connections
color selection
time limit
accessibility api
type arguments

primary surface
cell renderer
layerui subclass
count program
static int
accessible context
other swing
tostring method
entire input
list data listener
relative path
distinct data
spring layout
value pair
text nodes
second column
default implementation
dateformat class
pricelist object
same type
text node
tree selection
java language trail
progress indicator class
font object
int value
type java
decimal point
output ports
attribute value
java virtual machines
java runtime environment software
mixer interface
left mouse
com pareto
connectionpooldatasource object
multiple components
legacy code

layout code
key events
low-level events
xjc binding
parameterized types
policy tool
parallel group
implementation type
while loop
swing component
code sample
coordinate system
logical fonts
total number
saxlocalnamecount program
third argument
pop-up menu
style element
key-value pairs
attribute values
early access
xml schema definition
vertical scroll bar
html file
new value
audio system
java programs
minimum version
relational database
back buffer
unicode locale extension
fields of type parameters
preferred width
document listeners
current state
unchecked exceptions
start method
input parameters

parent web
same width
other applets
unmarshal validate
color choosers
line breaks
check box menu
info object
applet class
schema validation
system class
environment properties
large object
character sequence
input file
filteredrowset object
tool bar buttons
random access files
file systems
desktop pane
sortedset interface
schema file
specify requirement
first method
square brackets
concrete subclass
input sequence
chained exceptions
window events
same object
digital signature
getsource method
ldap name
desktop api
object name
same signature
new thread
java classes
background thread
window listeners
above example
dom tree
close method
many examples
unchecked warning
tree selection listener
xpath expression
vertical scroll
specific file
advanced data
focus listener
component listener
separate thread
member variable
zip file
anonymous class expression
swing text components
drivermanager class
xml processing
java naming
immutable objects
most applications
sound data
dictionaryservicedemo sample
raw types
element type
file choosers
compound message
single method
new integer
midi system
hard link
detailed information
top jpanel
third row
runapplet function
set interface
public key certificate file
key listener
return statement
design area
second method
file permissions
key-released event
service provider interface
item listener
system resources
other java
more statements
default set
sql type
formatter factory
column values
key code
right arrow key
midi output port
version note
images directory
window-closing events
math class
footer text
directory interface
content panes
swing api
simple program
base direction
uppercase letter
serviceloader class
drop-down list
action events
minimum sizes
lead selection
writelist method
highscore class
several methods
variable names
invisible component
minimum value
end users
search filter
error handling
tabbed panes
info objects
instance variable
list selection model
java cache
coffee houses
singleton design
byte stream
while statement
collection interface
default focus
factory methods
xjc binding compiler
security environment property
instance methods
same data
same file
word boundaries
directory services
unchecked exception
entire code
other threads
current node
unicode standard
absolute positioning
property editors
structured types
single argument
double value
static import
most cases
sqlxml objects
event-dispatching thread
xml element
top-to-bottom box

extensible stylesheet
tip text
key strokes
mutable objects
wildcard use
null value
table printing
other thread
empty string
environment variables
question mark
class members
mxbean interface
single line
progress dialog
sax events
particular file
other things
language transformations
menu layout
new object
runnable object
midi files
access modifiers
new jdbcrowset
many methods
multiple lines
accessible contexts
method declarations
social networking
first thing
method references
simple type
icon argument
stringbuilder append
path variable
read timeout

filteredrowset objects
other data
listener interface
many times
standard swing
right arrow
formal type
synth xml
default editor
class loader
primitive data type
related classes
line argument
first statement
serializable interface
main-class header
stringbuilder insert
branch nodes
shaped windows
rmi registry
ext directory
empty table
example xml
initial program
integer object
ldap operations
interface java
value property
certificate signing
xmlaccessororder annotation
particular object
next figure
server-side application
default layout
string regex
last element

dual carets
data structure
custom cell
bridge method
midi output
alpha value
required permission
resource bundles
various types
data flavor
public interface
file formats
transaction isolation level
bulk operations
custom class
essential java
count limit
visible area
signature object
random access
code base
further details
jtable class
dialog owner
semantic events
web browser
more components
windows xp
accessor methods
close button
new services
pattern class
path instance
file owner
other information
collection view

component orientation
gui builder
int argument
xml stream
gui components
statement objects
list model
element node
other file
how-to page
map interface
distinguished name
standard midi files specification
name-value pairs
resultset interface
boolean value
integer value
potential line
setter methods
state element
array objects
api examples
new policy
right side
stop method
new string
overloaded methods
data streams
glob syntax
support assistive
same class
numeric values
files specification
decimal separator
code examples
current jdk

C

Tutorial Statistics

Table C.1: JodaTime Statistics per Section

Parent Section Title	Section Title	Rel	Total
Section13-Manipulating DateTimes	SubSection13.2-DateTime methods	0	1
Section16-Input and Output	SubSection16.5-Direct access	0	1
Section16-Input and Output	SubSection16.1-Formatters	0	1
Section17-Advanced features	Section17.2-Converters	0	1
Section12-Querying DateTimes	SubSection12.1-Accessing fields	0	1
Section13-Manipulating DateTimes	Section13-Manipulating DateTimes	1	1
Section11-Working with DateTime	SubSection11.1-Construction	1	1
Section12-Querying DateTimes	SubSection12.3-Time fields	1	1
Section16-Input and Output	SubSection16.2-Standard Formatters	1	1
Section11-Working with DateTime	SubSection11.2-JDK Interoperability	1	1
Section13-Manipulating DateTimes	SubSection13.3-Using a MutableDateTime	1	1
Section17-Advanced features	SubSection17.1-Change the Current Time	1	1
Section17-Advanced features	Section17.3-Security	1	1
Section3-Instants	Subsection3.1-Fields	0	2
Section5-Durations	Section5-Durations	1	2
Section16-Input and Output	SubSection16.3-Custom Formatters	2	2
Section16-Input and Output	SubSection16.4-Freaky Formatters	2	2

Section14-Changing TimeZone	Section14-Changing TimeZone	2	2
Section9-Interface usage	Section9-Interface usage	0	3
Section4-Intervals	Section4-Intervals	1	3
Section15-Changing Chronology	Section15-Changing Chronology	2	3
Section13-Manipulating DateTimes	SubSection13.1-Modifying fields	3	3
Section12-Querying DateTimes	Section12-Querying DateTimes	0	4
Section1-Intorduction	Section1-Intorduction	0	4
Section12-Querying DateTimes	SubSection12.2-Date fields	0	4
Section3-Instants	SubSection3.2-Properties	1	4
Section6-Periods	Section6-Periods	2	4
Section3-Instants	Section3-Instants	4	6
Section7-Chronology	Section7-Chronology	2	7

Table C.2: JodaTime Statistics per API type

API type	**Rel**	**Total**
AbstractReadableInstantFieldProperty	0	1
DateTimeConstants	0	1
MutablePeriod	0	1
ReadableDuration	0	1
DateMidnight	0	1
DurationField	0	1
MutableInterval	0	1
DateTime.Property	0	1
appendTwoDigitYear	1	1
DateTimeUtils	1	1
PeriodType	1	1
DateTimeFormat	1	1

ISODateTimeFormat	1	1
DateTimeZone	1	1
BuddhistChronology	1	1
Instant	1	1
ISOChronology	1	1
JodaTimePermission	1	1
DateTimeFormatter	1	1
DateTimeField	0	2
ReadableInstant	0	2
ReadableInterval	0	2
ReadablePeriod	0	2
Duration	1	2
MutableDateTime	2	2
Period	1	3
Interval	1	3
DateTime.Property	1	3
Chronology	2	3
DateTime	8	18

Table C.3: Math Statistics per Section

Parent Section Title	**Section Title**	**Rel**	**Total**
3 Linear Algebra	3.4 Solving linear systems - part 2	0	1
11 Geometry	11.3 Binary Space Partitioning	0	1
4 Numerical Analysis	4.5 Integration	0	1
4 Numerical Analysis	4.7 Differentiation - part 5	0	1
11 Geometry	11.2 Euclidean spaces - part 6	1	1
3 Linear Algebra	3.4 Solving linear systems - part 3	1	1

4 Numerical Analysis	4.4 Interpolation - part 5	1	1
8 Probability Distributions	8.2 Distribution Framework	1	1
4 Numerical Analysis	4.3 Root-finding - part 7	1	1
4 Numerical Analysis	4.7 Differentiation - part 2	1	1
4 Numerical Analysis	4.7 Differentiation - part 4	1	1
8 Probability Distributions	8.3 User Defined Distributions	0	2
4 Numerical Analysis	4.4 Interpolation	0	2
11 Geometry	11.2 Euclidean spaces - part 3	1	2
3 Linear Algebra	3.3 Real vectors	1	2
16 Exceptions	16.4 Features	1	2
4 Numerical Analysis	4.7 Differentiation	1	2
4 Numerical Analysis	4.6 Polynomials	1	2
7 Complex Numbers	7.3 Complex Transcendental Functions	1	2
7 Complex Numbers	7.4 Complex Formatting and Parsing	2	2
4 Numerical Analysis	4.3 Root-finding - part 4	2	2
4 Numerical Analysis	4.3 Root-finding - part 5	2	2
9 Fractions	9.3 Fraction Formatting and Parsing	2	2
9 Fractions	9.2 Fraction Numbers	2	2
3 Linear Algebra	3.5 Eigenvalues/eigenvectors ...	2	2
11 Geometry	11.2 Euclidean spaces - part 2	2	2
4 Numerical Analysis	4.7 Differentiation - part 7	2	2
4 Numerical Analysis	4.3 Root-finding	0	3
3 Linear Algebra	3.4 Solving linear systems	1	3
7 Complex Numbers	7.2 Complex Numbers	1	3
4 Numerical Analysis	4.4 Interpolation - part 3	2	3
15 Filters	15.2 Kalman Filter	3	3
11 Geometry	11.2 Euclidean spaces	3	3

3 Linear Algebra	3.6 Non-real fields (complex,fractions ...)	0	4
16 Exceptions	16.3 Hierarchies	0	4
3 Linear Algebra	3.2 Real matrices	1	4
4 Numerical Analysis	4.7 Differentiation - part 6	1	4
4 Numerical Analysis	4.4 Interpolation - part 4	3	4
4 Numerical Analysis	4.3 Root-finding - part 6	5	5
14 Genetic Algorithms	14.2 GA Framework	1	6
14 Genetic Algorithms	14.3 Implementation	4	6

Table C.4: Math Statistics per API type

API type	**Rel**	**Total**
Fitness	0	1
SparseRealMatrix	0	1
MathArithmeticException	0	1
BigReal	0	1
UnivariateDifferentiableSolver	0	1
Space	0	1
NaN	0	1
ExceptionContextProvider	0	1
Vector3DFormat	0	1
MathIllegalArgumentException	0	1
Array2DRowRealMatrix	0	1
UnivariateIntegrator	0	1
PolynomialSolver	0	1
MathIllegalStateException	0	1
TrivariateFunction	0	1
BlockRealMatrix	0	1

CrossoverPolicy	0	1
BivariateFunction	0	1
UnivariateInterpolator	0	1
MutationPolicy	0	1
MathUnsupportedOperationException	0	1
IntegerDistribution	0	1
SelectionPolicy	0	1
RealVectorFormat	0	1
PolynomialFunction	0	1
FixedGenerationCount	0	1
SecantSolver	1	1
SingularValueDecomposition	1	1
ComplexFormat	1	1
ExceptionContext	1	1
TournamentSelection	1	1
FiniteDifferencesDifferentiator	1	1
TrivariateGridInterpolator	1	1
PolynomialsUtils	1	1
IntervalsSet	1	1
OnePointCrossover	1	1
IllinoisSolver	1	1
BisectionSolver	1	1
HermiteInterpolator	1	1
BicubicSplineInterpolator	1	1
KalmanFilter	1	1
MeasurementModel	1	1
FractionFormat	1	1

Vector3D	1	1
Interval	1	1
ConvergenceException	1	1
EigenDecomposition	1	1
SmoothingPolynomialBicubicSplineInterpolator	1	1
RandomKeyMutation	1	1
BivariateGridInterpolator	1	1
PolyhedronsSet	1	1
PegasusSolver	1	1
PolygonsSet	1	1
BrentSolver	1	1
RegulaFalsiSolver	1	1
ProcessModel	1	1
TricubicSplineInterpolator	1	1
ComplexUtils	0	2
UnivariateFunction	0	2
StoppingCondition	0	2
BigFraction	1	2
UnivariateDifferentiableFunction	1	2
UnivariateFunctionDifferentiator	1	2
UnivariateSolver	1	2
RealMatrix	1	2
AbstractIntegerDistribution	1	2
DecompositionSolver	2	2
BracketingNthOrderBrentSolver	2	2
GeneticAlgorithm	2	2
Rotation	2	2

RealVector	1	3
Fraction	2	3
Complex	3	4
DerivativeStructure	3	5

Table C.5: Collections(official) Statistics per Section

Parent Section Title	**Section Title**	**Rel**	**Total**
Wrapper Implementations	Wrapper Implementations	0	1
Set Implementations	Set Implementations	0	1
Lesson: Custom Collection Implementations	Lesson: Custom...	0	1
Lesson: Introduction to Collections	Benefits of the J...	0	1
Lesson: Introduction to Collections	What Is a Collections F...	0	1
Lesson: Algorithms	Composition	0	1
Map Implementations	Map Implementations	0	1
API Design	Return Values	0	1
Wrapper Implementations	Checked Interface Wrappers	0	2
Wrapper Implementations	Unmodifiable Wrappers	0	2
Summary of Implementations	Summary of Implementations	0	2
Deque Implementations	Deque Implementations	0	2
Answers to Questions and Exercises:	Exercises	0	2
Lesson: Algorithms	Searching - part 2	0	2
List Implementations	General-Purpose List Implementations	1	2
Map Implementations	Special-Purpose Map Implementations	1	2
Set Implementations	General-Purpose Set Implementations	1	2
API Design	Return Values - part 2	1	2
Lesson: Algorithms	Sorting	2	2
Deque Implementations	Concurrent Deque Implementations	2	2

Compatibility	Compatibility	0	3
Compatibility	Backward Compatibility - part 2	0	3
Lesson: Algorithms	Lesson: Algorithms	0	3
Convenience Implementations	List View of an Array	0	3
Lesson: Implementations	Lesson: Implementations - part 3	0	3
Lesson: Algorithms	Routine Data Manipulation	0	3
Wrapper Implementations	Synchronization Wrappers - part 2	0	3
Convenience Implementations	Immutable Multiple-Copy List	1	3
Set Implementations	General-Purpose Set Implementations	1	3
Lesson: Algorithms	Sorting - part 3	1	3
Lesson: Algorithms	Searching	1	3
Lesson: Algorithms	Finding Extreme Values	2	3
Lesson: Algorithms	Sorting - part 2	2	3
List Implementations	General-Purpose List Impl...	3	3
Deque Implementations	General-Purpose Deque Impl...	3	3
Set Implementations	Special-Purpose Set Impl...	3	3
API Design	Parameters	0	4
Lesson: Algorithms	Shuffling	1	4
Lesson: Custom Collection Implementations	How to Write...	2	4
Map Implementations	Concurrent Map Implementations	2	4
Map Implementations	Special-Purpose Map Implementations	3	4
Convenience Implementations	Immutable Singleton Set	4	4
Convenience Implementations	Empty Set List and Map Constants	0	5
Compatibility	Backward Compatibility	1	5
Lesson: Custom Collection Implementations	Reasons to Write...	1	5
Lesson: Implementations	Lesson: Implementations - part 4	2	5
Set Implementations	General-Purpose Set Impl...	3	5

Wrapper Implementations	Synchronization Wrappers	1	6
Compatibility	Upward Compatibility	0	7
Queue Implementations	General-Purpose Queue Impl...	2	7
Map Implementations	General-Purpose Map Impl...	2	8
List Implementations	Special-Purpose List Impl...	3	8
Answers to Questions and Exercises:	Questions	1	9
Queue Implementations	Concurrent Queue Implementations	1	9
Summary of Implementations	Summary of Implementations	0	10
Lesson: Custom Collection Implementations	How to Write...	2	10
Lesson: Implementations	Lesson: Implementations - part 2	0	17

Table C.6: Collections(official) Statistics per API type

API type	**Rel**	**Total**
AbstractSet	0	1
SynchronousQueue	0	1
ArrayBlockingQueue	0	1
Serializable	0	1
LinkedBlockingQueue	0	1
AbstractSequentialList	0	1
BigInteger	0	1
String	0	1
AbstractQueue	0	1
ClassCastException	0	1
DelayQueue	0	1
PriorityBlockingQueue	0	1
ConcurrentModificationException	0	1
Object	0	1

ListIterator	0	1
File	0	1
TransferQueue	0	1
Random	0	1
Integer	0	1
Card	0	1
CopyOnWriteArrayList	1	1
AbstractMap	1	1
LinkedBlockingDeque	1	1
EnumSet	1	1
WeakHashMap	1	1
EnumMap	1	1
ConcurrentHashMap	1	1
UnsupportedOperationException	0	2
TreeMap	0	2
CopyOnWriteArraySet	1	2
PriorityQueue	1	2
AbstractCollection	1	2
LinkedHashMap	1	2
IdentityHashMap	1	2
AbstractList	2	2
ConcurrentMap	2	2
BlockingQueue	2	2
SortedSet	0	3
Enumeration	0	3
ArrayDeque	1	3
Queue	1	4

SortedMap	1	4
TreeSet	1	4
LinkedHashSet	1	4
Hashtable	0	5
HashMap	0	5
Deque	2	5
Arrays	0	6
Vector	1	6
Comparator	1	6
HashSet	3	6
LinkedList	2	7
Set	3	12
ArrayList	2	13
List	4	14
Map	3	19
Collection	5	22
Collections	7	22

Table C.7: Collections(Jenkov) Statistics per Section

Parent Section Title	**Section Title**	**Rel**	**Total**
Java Collections Tutorial	Java Collections Tutorial	0	1
Java's Map Interface	Removing Elements	0	1
Java's Deque Interface	Removing Elements	0	1
Java's Deque Interface	More Details in the JavaDoc	0	1
Java's Map Interface	More Details in the JavaDoc	0	1
Java's Set Interface	More Details in the JavaDoc	0	1
Sorting Java Collections	Sorting Objects Using a Comparator	0	1

Java Collections Tutorial	Java Collections and Generics	0	1
Java's Queue Interface	More Details in the JavaDoc	0	1
Java's Collection Interface	Java's Collection Interface	0	1
Java's Collection Interface	Iterating a Collection	0	1
Java's List Interface	More Details in the JavaDoc	0	1
Java's NavigableMap Interface	More Detail in the JavaDoc	0	1
Java's Collection Interface	Collection Size	0	1
Java's NavigableMap Interface	headMap(),tailMap() and...	1	1
Java's NavigableSet Interface	pollFirst() and pollLast()	1	1
Java's List Interface	Removing Elements	1	1
Java's Set Interface	Removing Elements	1	1
Java's Queue Interface	Removing Elements	1	1
Java's Collection Interface	Adding and Removing Elements	1	1
Java's Stack Class	Searching the Stack	1	1
Java's NavigableMap Interface	ceilingKey(), floorKey(),...	1	1
Java's Collection Interface	Checking if a Collection Contains ...	1	1
Java's Map Interface	Adding and Accessing Elements	1	1
Java's NavigableMap Interface	pollFirstEntry() and pollLastEntry()	1	1
Java's NavigableSet Interface	ceiling(), floor(), higher()...	1	1
Java's NavigableSet Interface	headSet(), tailSet() and subSet()	1	1
Implementing hashCode() and equals()	equals()	0	2
Java's Set Interface	Generic Sets	0	2
Java's SortedMap Interface	More Details in the JavaDoc	0	2
Implementing hashCode() and equals()	hashCode()	0	2
Java's List Interface	List Implementations	0	2
Sorting Java Collections	Sorting Java Collections	0	2
Java's Map Interface	Java's Map Interface	0	2

Java's Queue Interface	Generic Queue	0	2
Java's SortedSet Interface	More Details in the JavaDoc	0	2
Java's Set Interface	Java's Set Interface	0	2
Java's Deque Interface	Generic Deque	0	2
Java Collections Tutorial	Java Collections and the equals()...	0	2
Java's List Interface	Java's List Interface	0	2
Java's NavigableMap Interface	celingEntry(), floorEntry()...	0	2
Java's List Interface	Generic Lists	0	2
Java's Deque Interface	Adding and Accessing Elements	1	2
Java's NavigableSet Interface	descendingIterator() and...	1	2
Java's Iterable Interface	Java's Iterable Interface	1	2
Java's Set Interface	Java Set Example	1	2
Java's Deque Interface	Java's Deque Interface	1	2
Sorting Java Collections	Sorting Objects by their Natural Order	1	2
Java's NavigableSet Interface	Java's NavigableSet Interface	0	3
Java's Queue Interface	Java's Queue Interface	0	3
Java Collections Overview	Java Collections Overview	0	3
Java's Map Interface	Generic Maps	0	3
Java's Set Interface	Adding and Accessing Elements	1	3
Java's Collection Interface	Adding and Removing Elements	1	3
Sorting Java Collections	Sorting Objects Using a Comparator	1	3
Java's Queue Interface	Adding and Accessing Elements	1	3
Java's Stack Class	Java's Stack Class	1	3
Sorting Java Collections	Sorting Objects by their Natural Order	1	3
Java's NavigableMap Interface	descendingKeySet() and descendingMap()	2	3
Java's Map Interface	Map Implementations	3	3
Java's NavigableMap Interface	Java's NavigableMap Interface	0	4

Generic Collections in Java	Generic Collections in Java	0	4
Java's List Interface	Adding and Accessing Elements	1	4
Java's Deque Interface	Deque Implementations	2	5
Java's SortedSet Interface	Java's SortedSet Interface	2	5
Java's Set Interface	Set Implementations	3	5
Java's SortedMap Interface	Java's SortedMap Interface	1	6
Java's Set Interface	Set Implementations - part 2	1	6
Java's Queue Interface	Queue Implementations	2	6

Table C.8: Collections(Jenkov) Statistics per API type

API type	Rel	Total
ArrayList	0	1
Entry	0	1
Iterable	1	1
LinkedHashSet	1	1
PriorityQueue	1	1
ArrayDeque	1	1
LinkedList	0	2
Stack	2	2
SortedMap	1	3
HashSet	1	3
HashMap	1	3
Iterator	0	4
TreeMap	1	4
TreeSet	2	4
Object	0	6
String	0	6

Comparator	1	6
SortedSet	1	6
Comparable	2	6
Deque	3	6
NavigableSet	5	6
NavigableMap	4	7
Collections	0	8
Map	2	9
Queue	3	9
Set	4	11
List	2	15
Collection	3	18

Table C.9: Smack Statistics per Section

Parent Section Title	**Section Title**	**Rel**	**Total**
Pubsub	Publishing to a node	0	1
Messaging using Chats	Messaging using Chats	0	1
Roster Item Exchange	Send a entire roster	1	1
Roster Item Exchange	Send a roster entry	1	1
File Transfer	Send a file to another user - part 2	1	1
XHTML Messages	Discover support for XHTML Messages	1	1
Multi User Chat	Discover MUC support	1	1
Multi User Chat	Manage affiliation modifications - part 2	1	1
Pubsub	Retrieving persisted pubsub messages	1	1
Multi User Chat	Discover room information	1	1
Data Forms	Answer a Form - part 2	1	1
Multi User Chat	Join a room	1	1

Multi User Chat	Discover joined rooms	1	1
Pubsub	Node creation and configuration	1	1
Multi User Chat	Create a new Room	1	1
Service Discovery	Manage XMPP entity features	1	1
Roster Item Exchange	Send a roster group	1	1
Pubsub	Receiving pubsub messages	1	1
Message Events	Requesting Event Notifications	1	1
File Transfer	Send a file to another user	1	1
XHTML Messages	Receive an XHTML Message	1	1
XHTML Messages	Send an XHTML Message	0	2
Provider Architecture:...	Provider Architecture:...	0	2
Message Events	Reacting to Event Notification Requests - part 2	0	2
Roster and Presence	Roster and Presence	1	2
Service Discovery	Discover information about an XMPP entity	1	2
Debugging with Smack	Debugging with Smack	1	2
Multi User Chat	Start a private chat	1	2
Service Discovery	Discover items associated with an XMPP entity	1	2
Data Forms	Answer a Form	1	2
XHTML Messages	Compose an XHTML Message - part 2	2	2
Service Discovery	Provide node information	2	2
Multi User Chat	Manage changes on room subject	2	2
Pubsub	Discover pubsub information	2	2
Data Forms	Create a Form to fill out	2	2
File Transfer	Recieving a file from another user - part 2	2	2
Service Discovery	Publish publicly available items	2	2
File Transfer	Monitoring the progress of a file transfer	1	3
Roster Item Exchange	Receive roster entries	1	3

Multi User Chat	Manage room invitations	1	3
Message Events	Reacting to Event Notifications	2	3
File Transfer	Recieving a file from another user	2	3
Processing Incoming Packets	Processing Incoming Packets	2	4
Message Events	Reacting to Event Notification Requests	2	4
Multi User Chat	Manage affiliation modifications - part 3	3	5
Multi User Chat	Manage role modifications - part 2	3	5

Table C.10: Smack Statistics per API type

API type	**Rel**	**Total**
PacketFilter	0	1
DiscoverInfo	0	1
SmackDebugger	0	1
InvitationListener	0	1
PacketExtension	0	1
MessageEventRequestListener	0	1
IQProvider	0	1
InvitationRejectionListener	0	1
SubjectUpdatedListener	1	1
RosterExchangeListener	1	1
Roster	1	1
MessageEventNotificationListener	1	1
FileTransferListener	1	1
FileTransfer	1	1
NodeInformationProvider	1	1
PacketCollector	1	1
PacketListener	1	1

XHTMLText	1	1
DefaultParticipantStatusListener	0	2
DefaultUserStatusListener	0	2
FileTransferRequest	1	2
DefaultMessageEventRequestListener	1	2
IncomingFileTransfer	1	2
DiscoverItems	1	2
OutgoingFileTransfer	1	2
FormField	1	2
LeafNode	1	2
PubSubManager	2	2
ParticipantStatusListener	2	2
FileTransferManager	2	2
UserStatusListener	2	2
Node	2	2
Chat	0	3
Form	3	3
XHTMLManager	3	3
RosterExchangeManager	3	4
MessageEventManager	3	4
ServiceDiscoveryManager	5	5
Connection	1	7
MultiUserChat	11	11