TECHNICAL UNIVERSITY OF CRETE
Department of Electronic and Computer Engineer (ECE)

# Managing Information Extraction in a Mashup Environment

*By:*

***Vasiliki P. Prokopi***

*Advisor: Professor Minos Garofalakis*

*Co-advisor: Professor Stavros Christodoulakis*

*Co-advisor: Professor Antonios Deligiannakis*

# Abstract

Unstructured text represents a large fraction of the world's data. It often contains snippets of structured information (e.g. people's names and zip codes). Information extraction (IE) techniques identify such structured information in unstructured or semi-structured text. As a task it can be seen as a way of filling database slots from sub-segments of text.

In parallel, Web mashups are Web applications developed using content and services available online. They offer the end user the ability to combine information or functionality from two or more existing data sources in order to create applications and Web pages customized to their unique needs.

In this work we combine the two aforementioned technologies in order to enable the analysis of extracted information with other structured user and enterprise data. For that purpose we use a state-of-the art statistical IE model-Conditional Random Fields (CRF)- in the setting of an open source data integration software application.

# Acknowledgments

First of all, I would like to dedicate this thesis to my wonderful parents. I am truly grateful to them; their continuous support and understanding was indispensible for the fulfillment of this work and generally for the completion of my undergraduate studies.

Secondly, I would like to thank my thesis supervisor, professor Minos Garofalakis, for his valuable assistance and collaboration during every step of this work.

Furthermore, I would like to thank my grate friends, inside and outside Technical University of Crete, for sharing with me some of the best years of my life, giving me so many things I will carry deep in my heart for all my life. Especially, thanks to their support and love, I managed to walk through this last year's difficulties.

I would also like to mention Apostolos Nydriotis; his previous work on the Apatar platform, and his will to help me and give me information whenever I needed was quite enlightening. Also, I would like to thank Daisy Zhe Wang, Sunny Khatri and Kun Li for the datasets they provided me generously.

At last, I would like to dedicate this especially work to my aunt Olga, who has left soon, and can't see me achieving this important goal. I know she would be full of proud and joy.

# Contents

List of figures .....

**1 Introduction** ..... 1

1.1 Thesis Contribution ..... 1

1.2 Thesis Outline ..... 2

**2 Background and Related Work** ..... 3

2.1 Information Extraction Basics ..... 3

2.1.1 Named Entity Recognition (NER) ..... 3

2.1.2 Methods of NER and IE ..... 4

2.1.3 The Statistical sequence model approach to NER ..... 4

2.1.4 Conditional Random Fields ..... 5

2.1.5 Inference algorithms ..... 7

2.2 CRF Model Selection ..... 8

2.2.1 Stanford Named Entity Recognizer (NER) ..... 9

2.3 Mashups ..... 11

2.3.1 What are Mashups? ..... 11

2.3.2 Enterprise Mashups ..... 13

2.4 Related Work ..... 14

**3 Mashup Platforms** ..... 15

3.1 Current State ..... 15

3.2 Platform Selection ..... 16

3.3 Apatar Architecture ..... 16

3.3.1 Core Engine ..... 17

3.3.2 Connectors ..... 18

3.3.3 GUI and Data Representation Layer ..... 18

**4 The ARAMIS platform** ..... 21

4.1 Training Widget (ieTrainer) ..... 21

4.1.1 Training and Evaluation Datasets ..... 10

4.2 Information Extraction Widget (IExtractor) ..... 26

4.2.1 IExtractor Architecture ..... 28

4.2.2 Real Time In-Memory Processing ..... 31

4.2.2.1 Stage 1: Classification ..... 31

4.2.2.2 Stage 2: Post Classification Processing ..... 32

4.2.2.3 Stage 2: Schema Creation ..... 38

**5 Demonstration** .....

**6 Conclusions and Results** ..... 40

# List of Figures


2.1: An example of CRF model on a String ..... 5

2.2: A Named Entity Recognition example for PERSON and ORGANIZATION entities identification ..... 5

2.3: The architecture of a typical mashup application ..... 11

2.4: A widget in Yahoo! Pipes that retrieves one or more RSS, Atom, RDF or iCal feeds from the URL(s) entered in the input box ..... 12

2.5: A simple mashup application built in the Yahoo! pipes Mashup platform ..... 12

2.6: Enterprise application development (Copyrights of the figure belong to (10)) ..... 13

3.1: Apatar's Architecture ..... 17

3.2: Apatar's main GUI ..... 19

4.1: The format of a training file ..... 22

4.2: Architecture of ieTrainer ..... 22

4.3: User interface for ieTrainer ..... 23

4.4: The *Property sheet page panel* for the IExtractor widget ..... 27

4.5: The architecture of *IExtractor* ..... 28

4.6: *IExtractor Graphical User Interface (GUI)* ..... 30

4.7: The *TokenTable* schema table showing the first three tokens extracted (*Lehman, Brothers, Federal*) with k=3 ..... 33

4.8: The process of constructing the Probability Bucket and the Histogram ..... 35

4.9: The histogram presenting the probability distribution of sentence estimation ..... 36

4.10: The prompt message user dialog. ..... 36

4.11: The input dialog for viewing the k-best labels the model has identified ..... 38

4.12: The database schemas ARAMIS supports. ..... 40

5.1: IExtractor output based on scenario 1.. ..... 43

5.2: Histogram for the extraction results of scenario 1.. ..... 44

5.3: Prompt window of IExtractor... ..... 44

5.4: IExtractor GUI demonstrating the “weak” entities... ..... 45
5.5: The window presenting the k choices for an underlined token... ..... 45
5.6:The mashup application for scenario 2... ..... 46
5.7: Result map..... 47

# Chapter 1

# Introduction

The majority of data we encounter, coming in their vast majority from the World Wide Web, contain a significant amount of information expressed using natural language. While unstructured text is often difficult for machines to understand, the field of Information Extraction (IE) offers a way to map textual content into a structured knowledge base. Information Extraction Systems implement tasks such as finding and understanding limited relevant parts of text, gathering information from many pieces of text and producing a structured representation of relevant information. Their goal is to identify and organize information in order to be useful to people and to put it in a semantically precise form that allows further inferences to be made by computer algorithms.

On the other hand, since it is important to make existing data more useful for personal and professional use, the mashup paradigm has emerged, triggered by the vast amount of WEB 2.0 applications created by developers and researchers. The mashup whole idea lies in the combination of data and functionality from two or more existing Web sources using an interactive graphical user interface. The Web sources that are used to build mashup applications mainly include Web applications and Web services.

Given the above, our idea was to combine the aforementioned technologies (namely, IE and mashups) into one, enabling users to select data of their own context of interest and use them effectively with other data gathered from many sources.

## 1.1 Thesis Contribution

In this thesis, our goal is to implement Information Extraction operations and more specifically Named Entity Recognition functionality, for the purpose of using it in an open source data integration platform. This flows the extracted information to be combined with many other sources of data, local or enterprise. In details, we planned to implement the following:

- An easy way for the user to manually train the system according to the Named Entities she wishes to recognize while hiding the complex theoretical knowledge needed for the feature extraction procedures (trainer widget.)
- A graphical, user friendly way to test trained models. The testing could be applied on raw text or on a website's content provided its *url* (extractor widget).

## 1.2 Thesis Outline

The remainder of this thesis is structured as follows: In Chapter 2 we present the background and related work to our project, presenting the two basic concepts we examine: information extraction and the mashup programming paradigm. Specifically, we introduce information extraction, briefly examining some key methods, and present the named entity recognition model we used and its parameters. Concerning the mashup technology, we introduce the mashup paradigm and discuss systems similar to our platform that embed information extraction technology inside a mashup platform. Then, Chapter 3 briefly introduces the most popular mashup platforms that are used today and analyzes Apatar, the platform that our system ARAMIS, is based on. Consequently, Chapter 4 discusses the design and implementation of our ARAMIS platform and Chapter 5 demonstrates the development of a mashup application using ARAMIS. Finally, in Chapter 6 the conclusions and future work of our project is presented.

# Chapter 2

# Background and Related Work

## 2.1 Information Extraction

Information extraction is the task of automatically extracting structured information such as entities, relationships between entities, and attributes describing entities from unstructured and/or semi-structured machine-readable documents. This enables much richer forms of queries on the abundant unstructured sources than possible with keyword searches alone. Typically, information extraction systems tract and understand limited relevant parts of texts, gather this information from different pieces of text, and produce a structured representation of relevant information such as relations(in the database sense) and knowledge bases. Information extraction includes the following techniques: *segmentation*, *classification*, *clustering and association.* This thesis focuses on the first two techniques, segmentation and classification and more specifically on the sub-task of Named Entity Recognition (NER).

### 2.1.1 Named Entity Recognition

As its name implies, Named Entity Recognition (NER) finds and classifies names in text. The term came at the center of attention of the Natural Language Processing community as a subtask of Information Extraction (IE) when it was noticed that it is essential to recognize information units like names, including person, organization and location names, and numeric expressions including time, date, money and percent expressions.

Specifically, the task of NER is, given a sentence, first to segment it into words that are part of entities, and then to classify each entity.

**Uses of NER**

NER is useful is useful in a diverse set of applications: The Named Entities recognized could be indexed, linked off, etc. Furthermore, NER is used in question answering applications where answers are often named entities. In addition, sentiments can be attributed to companies and products. Finally, it can be used as preliminary level of relation extraction since a vast majority of Information Extraction relations are associations between named entities.

### 2.1.2 Methods of IE and NER

The methods performing Information Extraction and Named Entity Recognition, analyzed in [1], are categorized along two categories: *hand-coded* or *learning based* and *rule-based* or *statistical.* A hand-coded system requires human experts to define rules or regular expressions or program snippets for performing the extraction whereas rule-based and statistical methods make decisions based on a weighted sum of predicate firings .

In this thesis we concentrate on statistical methods which are based on designing a decomposition of the unstructured text and then labeling various parts of it, either jointly or independently. They are ideal for open-ended domains, which is essential for our work in order to give the users of our system the ability to train the model on a domain of their preference.

### 2.1.3 The Statistical sequence model approach to NER

Statistical methods for performing Information Extraction and thus Named Entity Recognition based on Machine Learning techniques include two procedures, *Training* and *Testing.*

*Training* consists of the following steps:

**1)** Collect a set of representative training documents.
**2)** Label each token for its entity class or other (O)
**3)** Design feature extractors appropriate to the text and classes.
**4)** Train a sequence classifier to predict the labels from the data.

Finally, the *Testing* procedure is as follows:

**1)** Receive a set of testing documents
**2)** Run sequence model inference to label each token.
**3)** Appropriately output the recognized entities.

**Sequence Models**

Statistical methods use sequence models as the most prevalent methods of extracting data on plain text. The unstructured text is treated as a sequence of tokens and the extraction problem is to assign an entity label to each token. The following paragraph illustrates two examples of text segmentation taken from [2] and [3]

**Example 1:** Figure 2.1 shows a CRF model instantiated over an address string x "181 Shattuck North Berkeley CA USA". The possible labels are $Y = \{\text{apt.num, street num, street name, city, state, country}\}$. A segmentation $\mathbf{y} = \{y1,...,y_T\}$ is one possible way to tag each token in x into one of the field labels in Y.

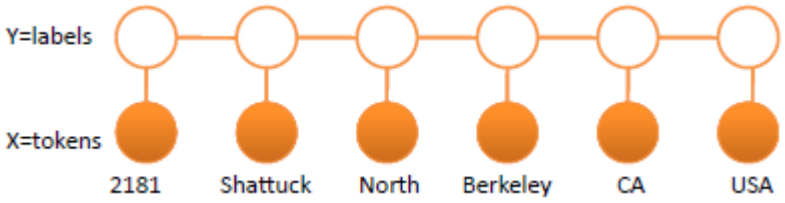


Figure 2.1: An example of CRF model on a String

| PERS | O | O | O | ORG | ORG |
|---|---|---|---|---|---|
| Murdoch | discusses | future | of | News | Corp. |

Figure 2.2: A Named Entity Recognition example for PERSON and ORGANIZATION entities identification

**Example 2:** Another sequence problem is presented in Figure 2.2. The output of extraction is a tagged sequence in which every $x_i$ is classified into one of a set $y$ of labels. The set of labels $y$ comprises of the set of entity types (like PERS and ORG) and a special label "O" for "Other" addressed to tokens that we do not want the model to recognize and apparently do not belong to any of the entity types.

There have been proposed many different models for assigning labels to a token sequence in a sentence. A very common approach is the task of *Classification* where the *classifier* in order to assign a label $y_i$ to each token $x_i$ uses features derived only from the token $x_i$ and its neighbors in $\mathbf{x}$. This category of models assumes that the labels we wish to predict are independent. However, in typical extractions systems the labels of adjacent tokens are rarely independent of each other. In addition, classifiers predict only a single label for each token every time. The aforementioned weaknesses of common classifiers have led to other models that can predict many variables which are interdependent. Popular approaches are *Hidden Markov Models (HMMs)*, *Maximum Entropy Markov Models (MEMM)* and *Conditional Random Fields (CRFs)*. CRFs, are the state-of-the-art method in Information Extraction (IE) tasks and we will elaborate on them the next.

### 2.1.4 Conditional Random Fields

Conditional Random Field (CRF) is a leading probabilistic model for solving IE tasks. The following definition taken from [4] and [5] defines the conditional probabilistic distribution of **y** given a specific assignment **x** by the CRF.

***Definition 1**:*

Let Y, X be random vectors, $\Lambda = \{\lambda_\kappa\} \in \Re^K$ be a parameter vector, and $\{f_k(y, y')\}_{k=1}^K$ be a set of real-valued feature functions. Then a *linear chain conditional random field* is a distribution $p(y \mid x)$ that takes the form

$$p(y \mid x) = \frac{1}{Z(x)} \exp\left\{\sum_{k=1}^{K} \lambda_k f_k\left(y_t, y_{t-1,}\mathbf{x}_t\right)\right\},$$

where Z(x) is an instance-specific normalization function

$$Z(x) = \sum_y \exp\left\{\sum_{k=1}^{K} \lambda_k f_k\left(y_t, y_{t-1,}\mathbf{x}_t\right)\right\}$$

$p(x)$

$x_t$

In order to indicate that each function can depend on observations from any time step, the feature function $f_k$ takes as parameter the observation vector $x_t$ which is defined as one containing all the components of the global observations $\mathbf{x}$ that are needed for computing features at time *t.* For example, if the CRF uses the next word $x_{t+1}$ as a feature, then the feature vector $x_t$ is assumed to include the identity of word $x_{t+1}$

**Advantages over other probabilistic models**

As mentioned in [4], the class of Conditional Random Fields is much more expressive, because it allows much more set of features to be used. Furthermore, their conditional nature escapes the need to model $p(x)$ . In addition, the features do not need to specify a state or observation allowing us to estimate the model using less training data. Another important property of CRFs is the convexity of the loss function (CRFs share all of the convexity properties of general maximum entropy models). The primary advantage of CRFs over hidden Markov models is their conditional nature, resulting in the relaxation of the independence assumptions required by HMMs in order to ensure tractable inference.

**Feature Functions**

Since feature functions are the key components of CRF, we now briefly examine them in further detail. For linear-chain CRFs (defined above), the general form of a feature function is $f_k\left(y_t, y_{t-1,}\mathbf{x}_t\right)$ which looks at a pair of adjacent states $y_t, y_{t-1,}$ , the current observation-token $x_t$ and produces a real valued number.

Now we present two possible feature functions, taken from [2], for **Example** 1 of 2.1.3 on address string segmentation:

$$f_1(y_t, y_{t-1}, x_t) = [x_t \text{ appears in a city list}]\,[y_t = \text{city}]$$
$$[x_t \text{is in a integer}]\,[y_t = \text{apt. num}][y_{t-1} = \text{street name}]$$
$$f_2(y_t, y_{t-1}, x_t) = i$$
$$i_{th}$$

The previous two feature functions produce binary values: 1 if the sequence under examination has the specified state identity and 0 otherwise. However, it should be mentioned that features are not limited to binary functions. Any real-valued function is allowed. Designing features for a CRF model or selecting ones is a very important procedure and we are going to reconsider them in chapter 4, where we will discuss the nature of features we are going to use for the Stanford NER classifier which performs IE inside Apatar, our mashup platform.

## 2.1.5 Inference

Inference, frequently referred to as sequential decoding, is defined as the process of finding the best tag sequences for given inputs. Traditionally the Viterbi algorithm is used (originally proposed in [6]). First, we present the 1-Best and k-Best Viterbi inference algorithms for exact computation. Then, we mention a method for approximate sequence decoding, Gibbs sampling which is a Monte Carlo method based on sampling.

### 1-**Best Viterbi**

The Viterbi algorithm is a classic dynamic programming based decoding algorithm. It has the computational complexity of $O(TL^2)$, where T is the input sequence size and $L$ is the size of the label alphabet. Here we give the equations of the dynamic programming algorithm that compute the best score of the sequence from 1 to with $i$ the $ith$ position labeled y.

$$V(t,y) = \begin{cases} \max_{y'} (V(t-1,y')) + \sum_{k=1}^{K} \lambda_k f_k(y, y', \mathbf{x}_t), & i \geq 0 \\ 0, & , \; i = -1 \end{cases} \tag{2.1}$$

The best labeling then corresponds to the path traced by $\max_{y'} (V(T,y'))$ where $T$ is the length of the observed sequence **x.**

### K-**Best Viterbi**

In order to produce *k*-best sequences, it is not enough to store 1-best label per node, as nth *k*-best sequential decoding gives up this 1-best label memorization in the dynamic programming paradigm. It stores up to *k*-best tables which are necessary to

form *k*-best sequences. The *k*-best Viterbi algorithm thus has the computational complexity of $KTL^2$ for both best and worst cases. Once we store the *k*-best labels, the *k*-best Viterbi algorithm uses the equation 2.1 and implements the decoding just like the 1-best Viterbi.

**Gibbs Sampling**

Gibbs sampling [7] defines a Markov chain in the space of possible variable assignments (in this case, hidden state sequences) such that the stationary distribution of the Markov chain is the joint distribution over the variables. The idea is to be able to sample from a posterior distribution (e.g. Defined by a CRF) Given a hidden state sequence Markov Model with N unknown variables choose an initial state for each variable at random, then samples as follows:

**i.** Select one variable at random, say **Xi**
**ii.** Compute the posterior over the states of **Xi**
**iii.** Select a state of **Xi** from this distribution
**iv.** Replace the value of **Xi** with the selected state
**v.** Repeat

## 2.2 CRF Model Selection

Taking into consideration all the benefits of Conditional Random Fields we talked about in section 2.1.4, we decided to use them for our NER needs inside the Apatar Platform. After selecting the model type, we focused on java implementations, because Apatar is a desktop Java application.
The two model implementations we found more suitable for our purposes were the CRF project by Sunita Sarawagi of IIT Bombay [8] and the Stanford Named Entity Recognizer (NER) by The Stanford Natural Language Processing Group [9].

After examining and experimenting with both of the implementations, we selected the Stanford Named Entity Recognizer. Our choice is based on the following reasons:

- Stanford NER provides a wide variety of well-engineered feature extractors for Named Entity Recognition providing the ability for the user to activate and deactivate any of them easily.
- This software package includes also three trained models: a 4 class model trained for ConLL[1], a 7 class model trained for MUC[2] , and a 3 class model trained on both data sets for the intersection of those class sets. These trained models contributed significantly on gaining a practical understanding of the purposes of Named Entity Recognition.
- It is a very well organized, structured and documented implementation that made comprehension and debugging an easy task.

[1] : http://www.cnts.ua.ac.be/conll2003/
[2] : http://cs.nyu.edu/faculty/grishman/muc6.html

- It makes it easy for the user to train new models.

### 2.2.1 Stanford Named Entity Recognizer (NER)[3]

Stanford NER (also known as CRFClassifier) is a Java implementation of a Named Entity Recognizer. The software provides a general (arbitrary order) implementation of linear chain Conditional Random Field (CRF) sequence models, coupled with well-engineered feature extractors for Named Entity Recognition. It is a product of the Stanford Natural Language Processing Group.

**Types of Features of Stanford NER**

- *Word Features:* Current word, previous word, next word, all words within a window.
- *Orthographic Features:* Identify patterns internal word or phrase features, ( Jenny $\rightarrow$ Xxxx, IL$-$2 $\rightarrow$ XX,
- *Prefixes and Suffixes:* Jenny $\rightarrow <$J, $<$Je, $<$Jen, …, nny$>$, ny$>$, y$>$
- *Label sequences*
- *Feature conjunctions*

**Trained Models/Classifiers Provided**

We already mentioned in section 2.2 that four different trained Models/classifiers are offered for recognition of named entities. The table below depicts the entities the 4 classifiers identify. The models were trained on a mixture of CoNLL, MUC-6, MUC-7 and ACE named entity corpora, and as a result they are fairly robust across domains. This is why we provide the users of our platform the ability to test their documents or websites of preference on these models besides the choice of training their own model.

| Classifier Name | Person | Location | Organization | Misc | Time | Money | Percent | Date |
|---|---|---|---|---|---|---|---|---|
| 3 class | + | + | + | - | - | - | - | - |
| 4 class | + | + | + | + | - | - | - | - |
| 7 class | + | + | + | + | + | + | + | + |

[3] : http://nlp.stanford.edu/software/CRF-NER.shtml

An important characteristic of the above models is that they use Distributional Similarity Features which offer some important properties such as:

- Induce a distribution over contexts (each word will appear in contexts.)
- Cluster words based on how similar their distributions are, use cluster IDs as features
- Provide a great way to combat sparsity.

This family of features provides some performance gain at the cost of increasing their size and runtime of the model, and the users of our platform should be advised to use it in order to train their own models via the properties file, in a way that we will demonstrate in Section 4.1.

# 2.3 Mashups

## 2.3.1 What are mashups?

A new breed of Web-based data integration applications is sprouting up all across the Internet. Colloquially termed mashups, they are defined as applications that combine data from more than one source into a single integrated tool. These applications are developed specifically for satisfying the users' need to combine multiple services and data sources to best serve this need. For example, many popular mashups make use of the Google Maps service to provide a location display of data taken from another source. The architecture of a typical mashup application is demonstrated in Figure 2.3

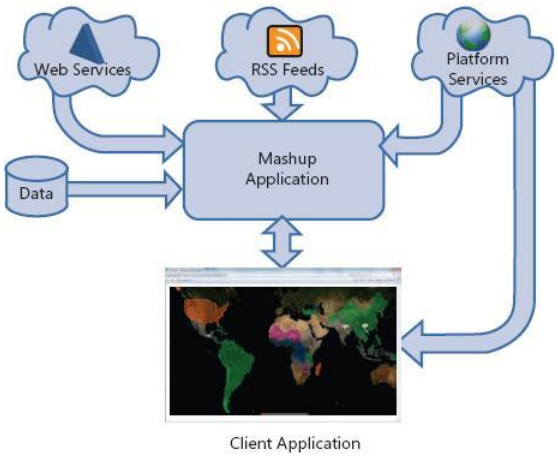


Figure 2.3 : The architecture of a typical mashup application

The main characteristics of a mashup are combination, visualization, and aggregation. It is important to make existing data more useful, for personal and professional use. To be able to permanently access the data of other services, mashups are generally client applications or hosted online. It is clearly observed that such kind of

applications are easy to implement and reusable. The first prerequisite comes under the notion of being used by non-expert users, and the second is necessary for saving time and effort needed to develop.

The core component of a mashup application is **widget**. A Widget (often called gadget, block, and flake) is a small program or piece of dynamic content that can be easily placed into a web site. Widgets can be written in any language (JAVA™, .NET, PHP, etc.) and can be as simple as an HTML fragment. They provide intuitive graphical user interfaces (GUIs) and are responsible for limited computational tasks. The most important property of "mashable" widgets is the one of passing events, so that they can be wired together, enabling data to flow into one, be processed and then flow out of it into another. This widget composition can lead to an application created as a network of widgets, wired together to produce the desired output.

Example of a Widget built on Yahoo! Pipes and a mashup application are shown on Figure 2.4 and Figure 2.5 respectively. Once the mashup is designed, the creator has the option to reuse it and/or and share it so that other users can use and edit it.

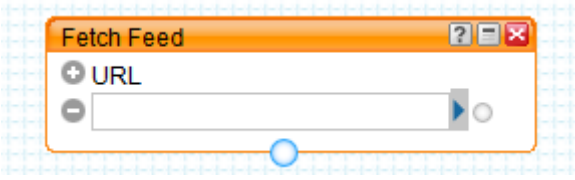


Figure 2.4: A widget in Yahoo! Pipes that retrieves one or more RSS, Atom, RDF or iCal feeds from the URL(s) entered in the input box.

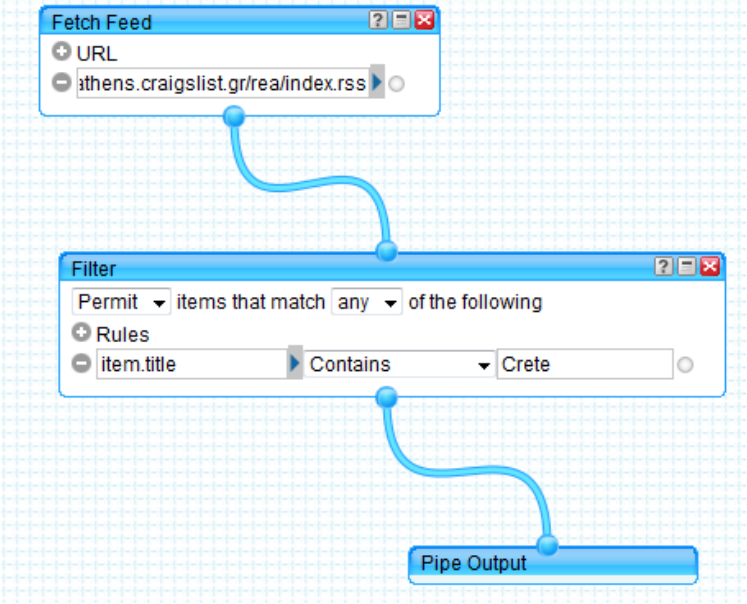


Figure 2.5: A simple mashup application built in the Yahoo! pipes Mashup platform

On the purpose of understanding the use and effectiveness of mashups, we demonstrate a real estate purchase scenario. A hotel business channel wants to buy land in Crete in order to build touristic resort in the island. The supervisor of the project wants to have a first view of the lands available, their exact locations and how much do they cost. A mashup the following way: A widget could fetch all the available lands and real estates in Greece (e.g. from Craigslist site) another service would filter the results to eliminate the lands out of Crete and finally a map service can be used for a visual representation of the results. The mashup application drastically simplifies the whole procedure since all the user has to do is to "describe" the data she needs and the mashup will deliver it. In addition, this simple mashup could be extended by the same or another user in constructing another more complex application.

### 2.3.2 Enterprise Mashups

The popularity of mashups has attracted the attention of enterprises that are beginning to take mashups from a pleasant Web hobby to enterprise-class systems because they see this as a way to augment their model for delivering and managing applications. Businesses can remix information from inside and outside the enterprise to solve situational problems quickly.

The lightweight nature of enterprise mashups enables employees to create and customize content on-the-fly [10], for the purpose of solving situational problems and taking advantage of business opportunities. The data enterprise mashups use often spreadsheets, e-mails and presentations.

Furthermore, these situational applications, having short-term lifespans, are usually address a smaller community of users and focus on solving specific local business requirements. On the other hand, typical enterprise applications are developed by IT experts for a large number of generic users and a more general purpose. Thus, enterprises represent the long-tail of enterprise application development, illustrated in mashup systems 2.6

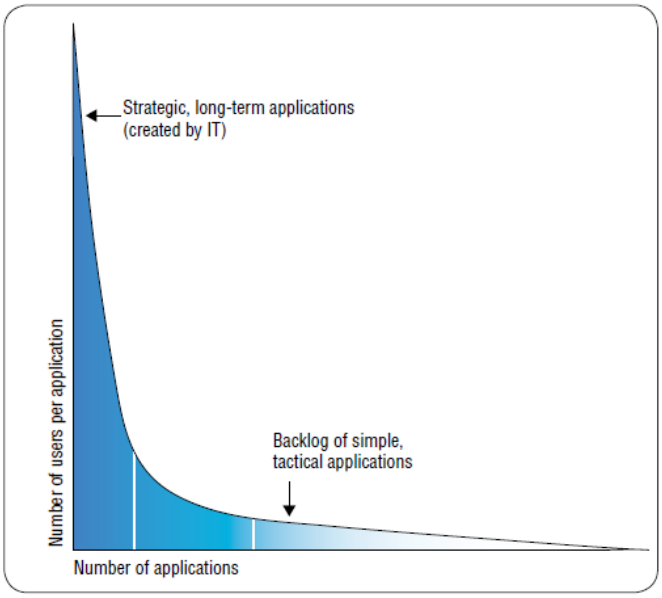


Figure 2.6 Enterprise application development
(Copyrights of the figure belong to **[10]**)

# 2.4 Related Work

## Previous approaches of IE usage in mashup Platforms

Earlier, we examined the state-of-the art in both Information Extraction and mashup development. In this section, we examine the conjunction of the two worlds: mashup platforms that perform information extraction tasks.

There has been some recent work concerning the manipulation of information extraction inside a data integration environment. However, the vast majority of mashup and data integration tools, web-based or desktop simply use information retrieval for the purposes of partial extraction of information.

*MashRank,* [11], provides a mashup authoring tool that builds on supervised extraction methods, where users annotate and refine examples, during the data gathering. Rules are the result produced by the example learning procedure and those learned extraction rules are applied to a given page before they result in schema records. Another approach based on rule induction is *SystemT* [12] (integrated in *IBM'S* MashupHub) which supports the iterative process of constructing and refining rules for information extraction via an annotation language. The result of the annotator (the rules produced on the specified documents provided by the user) is published.

Finally, we have examined the *Location Extractor* of Yahoo! Pipes, which we briefly describe in Section 3.1. This module analyzes text in each feed items title and description and attempts to identify addresses, location names or popular map service URLs. If the extractor finds location entities in the feed, it will annotate each item with a y: location sub-element containing that item's latitude and longitude in order to be presented by a map module. *Location Extractor* provides users with an already trained model, but only for addresses and locations. It does not give the user the ability to train a model on a domain of her preference, as we provide in our platform.

None of these earlier systems has the flexibility of our platform.
First, the system we propose allows users to train their own model, according to their personal preferences. Second, contrary to previous approaches that use rule-based techniques, training is based on a statistical approach with the use of Conditional Random Fields. This exploits the special properties of Conditional Random Fields that are more robust to noise in the unstructured data while long-range dependencies are well represented.

# Chapter 3

# Mashup Platforms

A mashup platform offers an intuitive, GUI based framework where an average user with no programming experience can build mashups easily. Section 3.1 gives an overview of the current state in mashup platforms.

## 3.1 Current State

Currently a number of platform tools exist, both consumer and enterprise oriented. Here, we briefly present some of the most popular ones, which are analyzed in [13]. Our objective is not to analyze all the tools but to give a view on the current state.

- *Damia*: a mashup tool provided by *IBM*. It offers users the ability to assemble data feeds from the Internet and enterprise sources. This tool focuses on data feed aggregation and transformation inside the enterprise environments. It also offers various presentation tools and technologies like *QUED-Wiki* and feed readers that consume Atom and RSS, to illustrate the data feeds of the platform.

- *Yahoo*! *pipes* is a web-based tool by Yahoo. The users can build mashup applications by aggregating data from web feeds, web pages, and other services. The mashups inside Yahoo! Pipes comprise one or more modules, each performing a separate task, like loading feed from a website, filtering, aggregating or sorting data. Feeds.

- *Popfly* is a web-based mashup application provided by Microsoft. Users use *Popfly* to create mashups combining data and media sources. Just like the modules of Yahoo! pipes, here we have *blocks* that construct the mashup. Each block is associated to a service like “Flickr” and exposes one or more functionalities. *Popfly* focuses on data presentation rather than data manipulation.

- *Google Mashup Editor (GME)* is a Mashup development, deployment and distribution environment by Google. It uses technologies like HTML, JavaScript, CSS along with GME, XML tags and JavaScript API that further allow a user to customize the presentation of the mashup output.

- *Exhibit* is a framework for creating webpages that contain dynamic and rich visualization of structured data. It enables its users to aggregate data obtained in various formats, like RDF/XML and Bibtex. *Exhibit uses* HTML pages as

standard output but also provides functionality for exporting its output to different formats, such as RDF/XML or Exhibit JSON.

- *Mashmaker* is an interactive web-based tool by Intel Corporation. It allows editing, querying, manipulating and visualizing semi-structured data. It differs from other tools in the sense that it works directly on Web pages and allows users to create mashups when browsing by combining content from different Web pages. The final goal of MashMaker is to suggest mashups or widgets for the visited Web pages that the user may want to use.

- *Apatar* is a mashup data integration tool that helps users integrate desktop data with the web. It is addressed both to individuals and organizations that need to move data, with tools for application integration, data migration and warehousing and synchronization. *Apatar* provides to its users connectivity to Microsoft Access and SQL Server, MySQL, Oracle, PostgreSQL, Sybase and XML. Its goal is to aggregate and manipulate data that can be reused from other applications, so additional tools that consume Apatar output formats can be used as the presentation layer.

In our work, we build on Apatar platform, an open source cross-platform data integration tool designed to enable batch data integration and provide simple user interfaces so that anyone, not just technical experts, can set up data integrations. The platform's architecture is described analytically in next section.

## 3.3 Apatar Architecture

Apatar[4] is an open source ETL (Extract-Transform-Load) and data integration software application. The platform's architecture is presented in Figure 3.1. The platform consists of three main components: the core component, the connectors component and the user interface component.

[4] : http://www.apatar.com

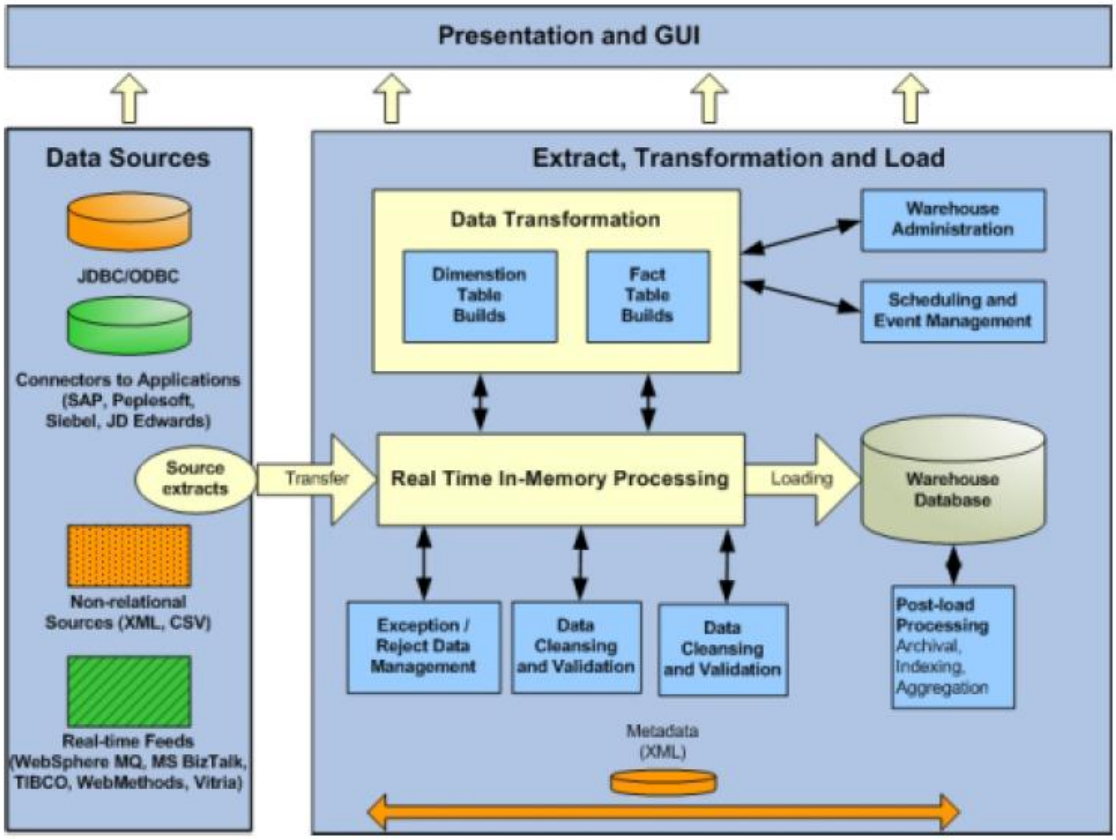


Figure 3.1: Apatar's Architecture

### 3.3.1 Core Engine

The core component consists of the application's ETL engine, the main data processing unit. In order to perform an operation, the input to the Core is provided through the connectors component and comes from one or more data sources. Subsequently, the data is transformed to tuples in Apatar's internal database. In this form, data is processed by the application's engine, and then loaded again to one or more connectors and probably to the presentation layer. The operations that are supported from the platform are both high level operations such as joins, selection, aggregations, filtering and so on, as well as lower level operations, such as transformations between different data types.

The core component is also responsible for defining fundamental structures that hold the relevant data manipulation information. Furthermore, it provides a mechanism to secure the system's consistency and extensibility. For instance, core defines all the structures that comprise the platform's internal database, its tables and its records, and also abstractly provides the basic features and structures that a connector must have in order to be functional.

### 3.3.2 Connectors

The connectors component, as its name implies, plays the role of connecting the core engine with data sources. Every connector provides a connection point for a specific data source through which data can be read, written or both. The platform supports over 30 connectors from various data source categories like database connectors (MySQL, Oracle, PostgreSQL, etc.), application connectors (Salesforce, etc.), file (.XML,.TXT) and others like e-mails, custom tables, and WEB 2.0 APIs

### 3.3.3 GUI and Data Representation Layer

The last component of Apatar comprises of a graphical user interface and a data presentation layer that offers the user ease of use and navigation by having supreme control over the data. Through the GUI, users take advantage of Apatar's features; they can create, modify, publish or run mashup applications, while the data presentation layer allows data supervision during the users' manipulations over the data.

The main GUI is presented in Figure 3.2. As can be clearly seen, the platform's main window is divided into two areas. The connectors and functions' area, where all the connectors and operations offered to the user are displayed as widgets, and the work area where the data integration job, called *Datamap* takes place. In order to create a *Datamap* user only has to drag and drop the connectors of their choice in the work area, configure them and connect them together.

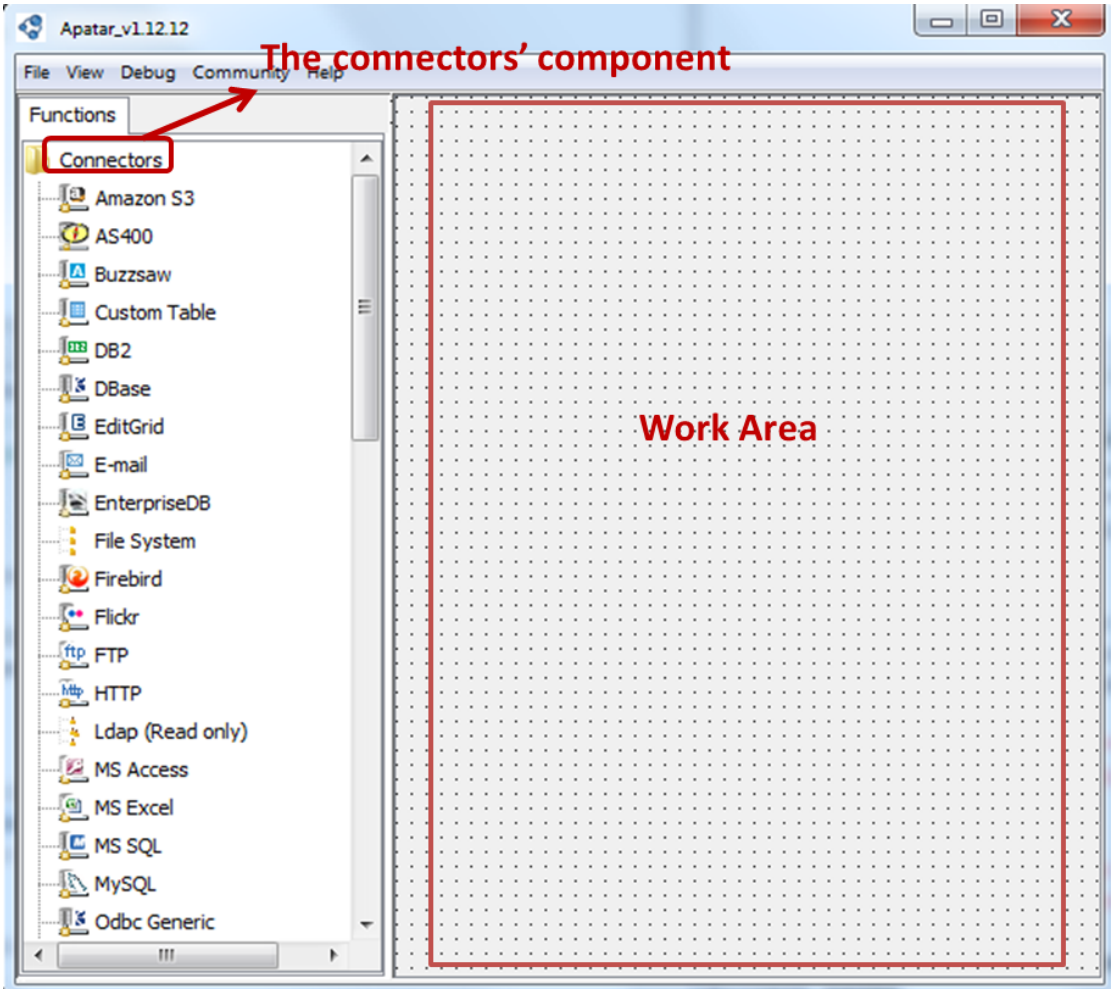


Figure 3.2: Apatar's main GUI

### 3.3.4 Extensibility

One of Apatar's great features is its extensibility. This is achieved through the use of Java Plug-in Framework (JPF). JPF provides a runtime engine that dynamically discovers and loads "plug-ins". A plug-in is a structured component that describes itself to JPF using a "manifest". Plug-ins are added to the registry at application start-up or while the application is running but they are not loaded until they are called.

The major goal of JPF is that the application (and its end user) should not pay any memory performance penalty for plug-ins that are installed, but not used. So, plug-ins are added to the registry at application start-up or while the application is running but they are not loaded until they are called.

Based on the above mechanism, every new widget (connector, operation, function or GUI component) in Apatar is implemented as a plug-in. This is achieved with an XML file that describes the widget, named after the name of the plug-in. This necessary document describes the plug-in to JPF in order to be registered in the framework and loaded upon call. Thus, whenever the application starts, a predefined

plug-in folder containing the entire manifest files is scanned, the available plug-ins are registered to JPF and are available for use.

# Chapter 4

# The ARAMIS Platform

This chapter introduces the main work of this thesis, the ARAMIS (Automatic Recognition and Mashup Integration System) platform, which extends the Apatar platform (analyzed in Chapter 3) in order to implement named entity recognition (NER) tasks inside the Apatar mashup environment. To achieve this goal, the work was split in 2 basic components-widgets; the training component (ieTrainer) where the CRF model is trained by the user and the extractor component (iExtractor) that performs the Named Entity Recognition task, the extraction of named entities as described in chapter 2.

## 4.1 Training widget (ieTrainer)

Generally, ARAMIS is designed as a learning-based system, requiring manually labeled unstructured examples to train the machine learning model for extraction. The purpose of this widget is to train a Conditional Random Field Model (the selected Named Entity Recognizer statistical model) and offer its users the ability to train the model for the purpose of recognizing entities specific to their application. For instance, one could train the model to recognize location names, organization names, car names, sport teams, university names and courses, etc.

Concerning the ieTrainer's design, a training file is given as input to the component. It can be either a text file (.txt), a file with comma separated (.csv) values or a file with tab separated values (.tsv). The training file must be in a one token per line format in order to be read by the `ColumnDocumentReaderAndWriter` class of the CRFClassifier package. The training data source should be also annotated with the correct answer (appropriate entity), or with the other symbol (O). This is done manually by the user or by an annotation tool. Furthermore, the users can explicitly specify more features for the word, by adding these in the training file in a new column and then put the appropriate structure of their file in the map line (*map* property) in the features file discussed in section 4.2. Figure 4.1 illustrates an example of the input format required for the ARAMIS training component.

| | |
|---|---|
| Germany | LOCATION |
| 's | O |
| representative | O |
| to | O |
| The | O |
| European | ORGANIZATION |
| Union | ORGANIZATION |

Figure 4.1: The format of a training file

In addition to the training file, ieTrainer takes for input a features file. The features generally, are the most important part of the Named Entity Recognizer task because they significantly determine how well the model is trained in order to recognize the majority of the entities it has learned during the extraction procedure. The format of a features file is the form *property* = *value* for every row. The addition of new features and even the selection of the features provided with NER recognizer is a procedure that requires experience with natural language processing issues. Thus we have chosen to provide the non-expert user the ability to train a model by only worrying about having a proper training file. This is done by creating the features file (.prop file) on the fly inside the folder where the user has chosen to save the trained model. The default features created include some basic features (illustrated in section 4.2.). This design achieves its goal of hiding the complex linguistic details of the features. Figure 4.2 illustrates the architecture of the ieTrainer connector.

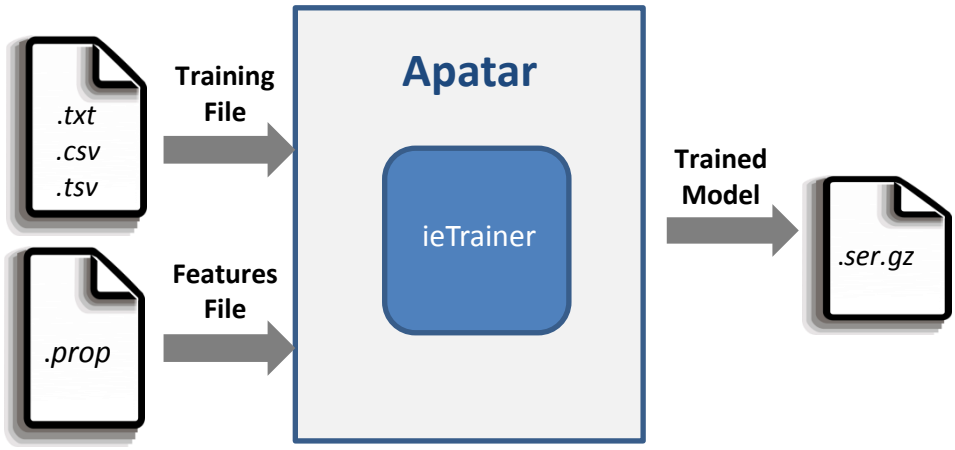


Figure 4.2: Architecture of ieTrainer

In Figure 4.3 the simple ieTrainer user interface is depicted. Via the GUI, the user performs the following actions (in order):

1) Adds a file for the training of the model.
2) Chooses the folder where the model is going to be saved,
3) Inserts a name for the trained model.
4) Uploads a properties file (this action is optional).

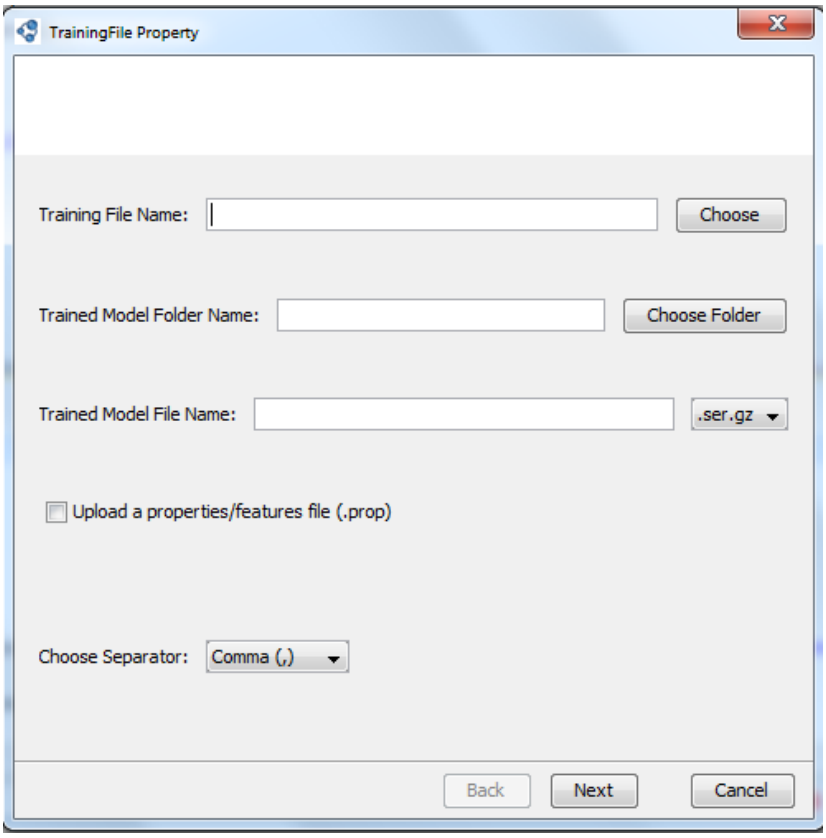


Figure 4.3: User interface for ieTrainer

As soon the user inserts all the appropriate elements in the GUI form, ieTrainer performs the following actions:

- The training file is parsed. The $property = value$ contents of a features file are being processed.
- An instance of the model is created taking as parameter a Properties object that contains the features of the model to train.
- The training of the model is performed. (the $\lambda_\kappa$ parameters of the model are estimated).
- The trained model is serialized to a file on the given path the user has chosen through the *Trained Model File Name* and the *Trained Model Folder* name.

- The serialized trained file (.ser) is zipped automatically making it faster and smaller.

One thing that must be mentioned is that the design of ieTrainer allows the user to process the input of the training file (the labeled words) and to perform any manipulations enabled in the Apatar platform enables through the its widgets and operations.

## Feature Selection

As we already mentioned, the selected features are a very important aspect of the Conditional Random Fields model. In this section we are going to examine in more details the selection of features for the NER Stanford classifier we used for the purposes of the ARAMIS platform.

Through a text file some properties can be defined. The properties define a great number of features which are all included in *SeqClassifierFlags* class. Every property enables one or more family features to perform on the data. The properties file is either created on the fly or provided by the user. The default properties that are created automatically for the non-expert user are the following:

- $map = word = 0, answer = 1$ :
  This property defines the structure of the training file; for example, this tells the classifier that the word is in column 0 and the correct answer is in column 1.

- *useClassFeature* :
  It includes a feature for every class (label). It is triggered whenever a token is identified as a class we want to recognize.

- *useWord* :
  It gives a feature for every word/token.

- *useNGrams* :
  It creates features for letter n-grams[5], substrings of each word.

- *noMidNGrams* :
  Do not include character n-gram features for n-grams that contain either the beginning or end of the word.
- *usePrev* :
  It is triggered for previous words.

- *useNext* :
  It gives features for next word

[5] : An n-gram is a contiguous sequence of *n* items from a given sequence of text or speech. An *n*-gram could be any combination of letters.

- *useDisjunctive*:
  Used to include as features disjunctions of words anywhere in the left or right *disjunctionWidth* words (preserving direction but not position)

- *useSequences*:
  This feature enables previous and next words or classes (labels).

- *usePrevSequences*:
  Used just like *useSequences* to trigger **only** previous words and tokens that belong to previous classes.

The following properties are word shape features. These types of features map words to simplified representation that encodes attributes such as length, capitalization, numerals, Greek letters, internal punctuation, etc.

- *useTypeSeqs*:
  This provides basic zeroeth order word shape features.

- *useTypeSeqs*2:
  Adds additional first and second order word shape features.

- *useTypeySequences*:
  It includes some first order word shape patterns.

- *wordShape* = *chris*2*useLC*:
  This is the class used for identifying lexical patterns for each word. Other classes to choose exist in *WordShapeClassifier* class.

- *maxleft* = 1:
  This specifies the order of the CRF: order 1 means that features apply at most to a class pair of previous class and current or current class and next class.

Besides the default properties, in our experiments we have used the following:

- *useTitle*:
  Matches a word against a list of name titles (Mr., Mrs.)

- *useLastRealWord*:
  Checks whether the previous word is of length 3 or less, and in this case it adds an extra feature that combines the word two back and the current word's shape.

- *useNextRealWord*:
  Similar to the previous one, it checks whether the next word is of length 3 or less, and adds an extra feature that combines the word after next and the current word's shape.

- *disjunctionWidth* :
  Defines the number of words on each side of the current word that are included in the disjunction features.
- *saveFeatureIndexToDisk* :
  Used to save the feature index to disk and read in later.

- *useLongSequences* :
  Allows plain higher-order state sequences out to minimum of length or *maxLeft*

- *useOccurencePatterns* :
  This is a much engineered feature designed to capture multiple references to names.

### 4.1.1 Training and Evaluation Datasets

The datasets that we trained our model were:

- *DBLP* Computer Science Bibliography (around 36000 records) with 5 classes
  i. Title
  ii. Author
  iii. Topic
  iv. Editor
  v. Date

- Address Strings (around 5000 records) extracted from the yellow pages trained to recognize 5 classes:
  i. Street Number
  ii. Street Name
  iii. City
  iv. State
  v. Zip Code

The documents we used for training were datasets of the above two categories and some webpages where we performed extraction using the models described in 2.2.1. One website we used for extracting entities is Craigslist[6] a classified advertisements website with sections devoted to jobs, housing, personals, for sale,

[6] : http://www.craigslist.org/about/sites/

## 4.2 Information Extraction Widget (IExtractor)

The basic widget of the ARAMIS platform is the IExtractor widget; this is responsible for performing the Named Entity Recognition task, based on user input (text or a website) and a trained model. This service enables a mashup platform to focus on some semantics of the data in order to save in its internal database a portion of the data that are mostly important to the user instead of keeping long segments of text. As long as the data a user daily encounters and saves augment significantly, it is easily understood that an information extraction task is a very useful tool for a Mashup platform like Apatar.

For the purpose of implementing information extraction, user can is provide in the means of a text file or a website. In more detail, users can perform the recognition task either locally (on a text file located inside their hard drives), or in a website of their choice. Concerning the first choice offered, as explained in Section 4.1 , the test file can be either a text file (.txt), a file with comma separated values (.csv) or a file with tab separated values (.tsv). The training file must be in a one class of CRFClassifier package. On the other hand, a URL could be provided by the user to extract the desirable entities. We should note that the target website for extraction should be a simple html page because the `JEditorPane` class that is used to display the unstructured content for extraction has limited *html* and *Css* support while it does not support *JavaScript* or *applets*. However, some basic *html* websites can be displayed and those are the ones we aim to focus on for our experiments. Finally, the user can also type some text inside the `JEditorPane` panel through the keyboard device in order to get the extracted entities.

User's input to IExtractor, is given through the widget's user interface that is borrowed from Apatar's Presentation and User Interface component and is called *Property sheet page panel.* This is shown in Figure 4.4. The parameters the user must insert are the following:

- *testFile* : The text file to perform the extraction task.
- *trainedModel* : The trained serialized NER model. This could be a file that the user already has trained using the ieTrainer (section 4.1) or one from the included serialized models which is provided with the NER Stanford Named Entity Recognizer (NER). These models are located inside IExtractor's folder inside the *classifier* folder.
- *url* : In case user prefers to extract structured entities from a website, this is the field where the *url* is provided. In addition, the checkbox *readFromFile* must be unchecked in order to inform the platform that the input comes from a website and **not** a user file. In other words, *testFile* and *url* fields are used in a mutually-exclusive manner.
- *k* : This parameter concerns the number of possible sequences we want the classifier to extract. Particularly, this refers to the k-best Viterbi inference algorithm used for extraction.

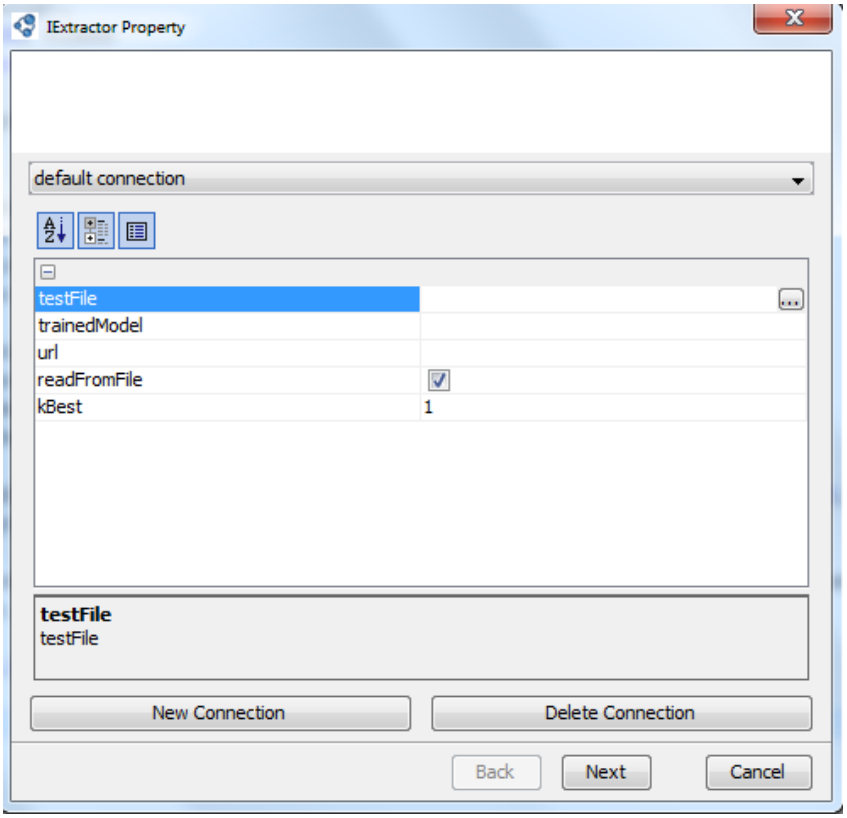


Figure 4.4: The *Property sheet page panel* for the IExtractor widget.

### 4.2.1 IExtractor Architecture

We focus on architecture of IExtractor in this section. It is illustrated in Figure 4.5. There are two main components in the platform: the *Presentation Layer* and the *Real-Time In-Memory Processing* Unit.

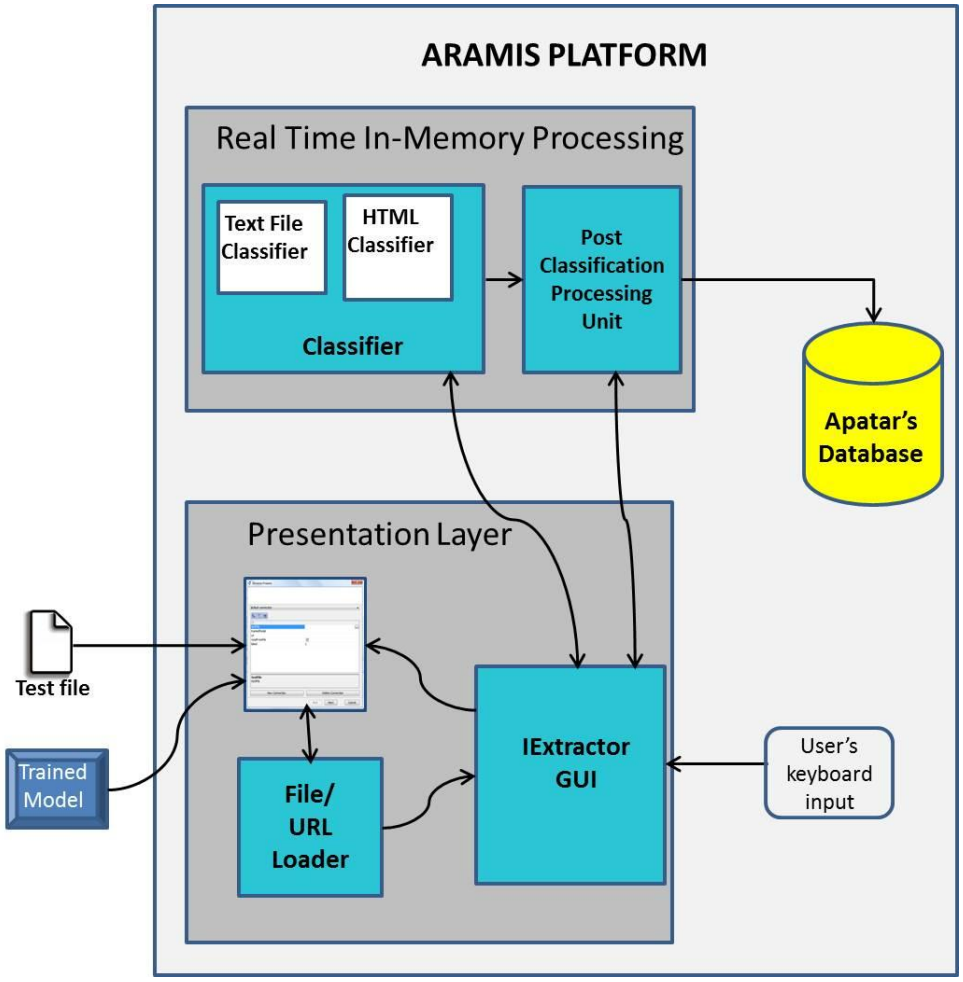


Figure 4.5: The architecture of *IExtractor*.

**Presentation Layer**

The presentation layer comprises all the components that handle the visual representation of the iExtractor. First of all it contains the *Property sheet page Panel* (shown in Figure 4.2 ), which is borrowed from Apatar's Presentation and GUI layer. Furthermore, it contains *IExtractor GUI* the basic representation component of the IExtractor widget and *the File/URL loader* that identifies the source of text that is going to be passed to the editor of the GUI, and loads the file or the contents of the website. The first component of the *Presentation Layer* is analyzed in 4.2, therefore we are going to examine the other two.

From the three components of *Presentation layer*, the *IExtractor GUI* is the most important, as it visualizes the user input, presents the output of the classifier, and

visualizes the results of the post-classification processing unit over the data. Figure 4.6 depicts *IExtractor GUI*'s layout. The main area of the panel is occupied by the JEditorPane which is the white editable area where the contents of user input are presented before and after extraction. At the top of the editor area there is a tool bar that allows some basic actions over the data. The *File* menu offers the ability to save the contents of the editor. The user can save untagged contents of the editor area or the tagged contents that result after the extraction. The *Edit* option consists of all the common edit options such as *Cut, Copy, Paste, and Clear.* At the bottom of the text area the *NER* button triggers the Named Entity Recognition procedure.

The panel on the right of the editor demonstrates the colors for each entity tag. In other words, every entity label such as *Person, Location, etc.* is mapped during runtime (dynamically) with a tag color in order to get every entity of the category highlighted, thus distinguished from the raw text and the other tagged entities.
The colors are assigned randomly, independent from the name of the labels or their number.

Finally, the interaction with users offered by the *IExractor GUI* is a product of the *Event Listeners* registered to the GUI's components. Except from the *Listeners* for the buttons and menu items detecting mouse action events, one less obvious case is the *Caret Listener.* This is triggered by caret events that occur when the *caret* – the cursor in the editor indicating the insertion point – moves or when a selection in a text component changes. The use of *Caret Listener* enables capturing of users' actions in the editor and the editing of its contents. Section 4.2.2.2 justifies the choice for *CaretListener* because the caret events that occur are very important in gathering information for some procedures of the *Post Classification Processing Unit.*

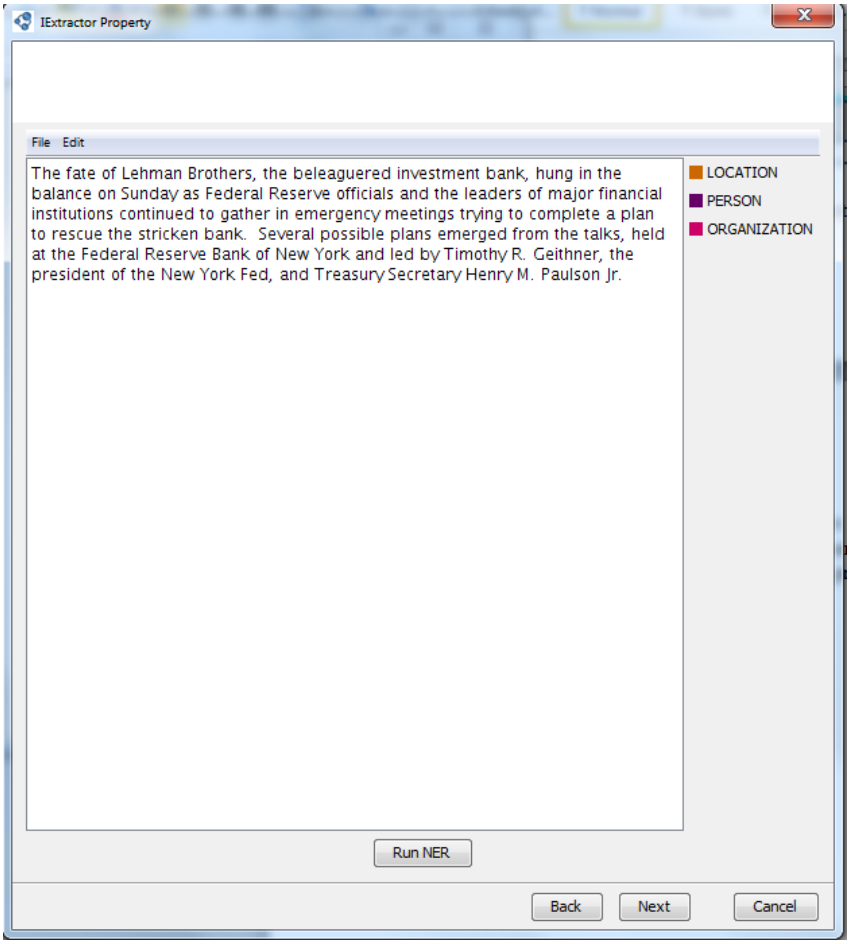

Figure 4.6: *IExtractor Graphical User Interface (GUI)*

The *File/URL Loader* component does a very simple job. It takes the input from the *property sheet page*, checks whether its source is a user file or a website and loads the content to the JEditorPane panel that represents the editable text area of the GUI. In the website case, the component does the following trick: it causes the thread that is handling the *IExtractor GUI* to suspend execution (sleep) for a 2 second period. This was done because the editor sometimes could not load the website to the editor, so by pausing the thread's execution, the elements of the website would have enough processor time to be fetched.

### 4.2.2 Real Time In-Memory Processing

After examining the structure, the visualization and the parameters IExtractor takes, we proceed to analyze the essential task of *IExtractor* widget, extraction. This is performed by the *Real Time In-Memory Processing* unit, and specifically by the *Classifier* and the *Post Classification Processing* component. The first one performs the statistical classification procedure and the second processes the output of the classifier and performs important tasks in order to deliver it at first to the *Presentation*

*Layer* and then to the platform's warehouse database. Eventually, the selection and creation of the schemas to be saved in database take place. The remainder of this chapter explores these stages' procedures in further detail.

The tasks performed from this unit, are split in three stages: The first stage implements the core procedure of Named Entity Recognition (generally called classification) in the contents of the editor either the source is a local file or a website. The second stage performs various post classification processes in order to deliver the text content properly to the platform's database and to visualize it in a nice manner to a degree of user interaction. The last stage concerns the creation of the schemas in the database based on user's choices.

#### 4.2.2.1 Stage 1: Classification

This stage is performed after user input is loaded in the text editor and the *NER* button is pressed.

**Text file/Html Contents Classification**

The first step of the classification concerns reading the text contents of the file or the html contents of a website. The Stanford NER classifier uses the `DocumentReaderAndWriter` general interface for reading data and writing output into and out of `SequenceClassifier` models. For text and xml (html format is treated similarly to xml) format the model uses the `PlainTexDocumentReaderAndWriter` to read plain text documents and write those documents once classified. The classifier will tokenize the text and treat each sentence as a separate document. Note that every whole sentence is assigned a probability as a result of the inference procedure. Afterwards, we take this whole sentence and split it in tokens in order to process them and save them inside the ARAMIS database schemas. Once a `PlainTexDocumentReaderAndWriter` object is instantiated the `init` method is called taking as parameter a `SeqClassifierFlags` object that represents the properties discussed in section 4.1. As we already know, the properties are defined by default or by the user specified properties file. So `init` loads these properties to the model in order to define its behavior. Furthermore, the contents of the text editor are processed as a *String* object by a method called `makeObjectBankFromString`. The latter reads the String in an `ObjectBank`, a collection of `Objects` taken from input sources and then tokenized and parsed into the desired kind of `Object`. This process helps to wrap the tokenized contents into lists of `CoreLabels`; each list represents all the appropriate info for each token (the token itself, the start and end character offset inside the sentence, its shape annotation and its sequence number in the sentence.)

After the text has been tokenized and read into proper objects supported by the model, the classification process takes place where `classifyKBest` method classifies and returns the k-best sequences based on the *k* parameter the user has chosen. This method runs for every list of `CoreLabel`. Concerning the inference algorithm that finds the k-best sequences, the Viterbi algorithm is run on the sequence model in order to find the best sequence. Afterwards, every classified list is sorted from highest count to lowest. The output of the classifier is in `inlineXMLformat` (e.g.,

<PERSON>Bill Smith</PERSON> went to <LOCATION>Paris</LOCATION>). This format is used to benefit all the post classification procedures we are going to examine in the following section. Both for file and *html* content, the classified contents for **only** the best sequence-the sequence with the greatest probability score-are passed to the editor and presented through *IExractor GUI* .

#### 4.2.2.2 Stage 2: Post Classification Processing

This stage implements all the processes that follow the classification procedure. These processes include:

**a)** Construction of the structure to hold the all the appropriate information about the classified tokens
**b)** Processing of the *inlineXml* formatted classified tokens, highlighting them similarly to their assigned entities/labels, and visualization of the result through the editor.
**c)** Construction of a structure to hold the tokens and their information per probability **only** for the best sequence and histogram creation to illustrate the distribution of the sentences classification, based on this structure.
**d)** Highlight "weakest" sentences.
**e)** Presentation of the k-options for a specific token and update the content of the structure that will fill the database's tuples (this process is optional and depends on the user' purpose.)

**a) Priority queue construction for the classified tokens**

Since the text or html contents are classified per sentence – every sentence is assigned a probability and its tokens are recognized and labeled with the entity the model chose to assign. The labeling is implemented with an `inlineXmlFormat` output format we mentioned in 4.2.2.1. Since our final goal is to construct a schema table with one token per tuple, the first step is to split every classified sentence and deduce the desired information. The information we want to save for every token is stored in a `TokenTableRecord` object. This structure consists of the following fields:

- *pos* : The word position of every special token inside a sentence
- *sentenceId* : The sequential number of the sentence that the token belongs to.
- *token* : The word token that has been labeled from the classification procedure.
- *label* : The labeled entity assigned to every token during the classification process
- *prob* : The probability score counted for every sentence during inference (top k best Viterbi).
- *k* : The k parameter defining the entities with the top-k highest probabilities.
- *characterOffsetBegin* : The sequence number of the first character of each token inside the classified sentence.
- *characterOffsetEnd* : The sequence number of the last character of each token inside the classified sentence.

As soon as we construct the object that holds the entire classified token information, we have to decide on the structure to place. We choose a `PriorityQueue`, because we want to keep the classified entities sorted by the *sentenceId* , the position in the sentence ( *pos* ) and *k*. In other words, starting from the first token of the first sentence, we put in the queue all the *k* best `TokenTableRecord` objects for it and only then we go on to the next sentence. The output of this structure is maintained in the *TokenTable* (see Figure 4.7 for a snapshot), which is the schema table with all the tokens that have been extracted by the Named Entity Recognition procedure. Note that regardless the *k* parameter the user has entered, if a probability of the sentence segmentation is under a very low threshold defined internally by the model, this sentence is eliminated and discarded from the structure.

| No. | pos | sentenceId | token | label | prob | k |
|---|---|---|---|---|---|---|
| 1 | 3 | 0 | Lehman | ORGANIZAT... | 0.96220861... | 1 |
| 2 | 3 | 0 | Lehman | LOCATION | 0.01161753... | 2 |
| 3 | 3 | 0 | Lehman | ORGANIZAT... | 0.00960977... | 3 |
| 4 | 4 | 0 | Brothers | ORGANIZAT... | 0.96220861... | 1 |
| 5 | 4 | 0 | Brothers | LOCATION | 0.01161753... | 2 |
| 6 | 4 | 0 | Brothers | ORGANIZAT... | 0.00960977... | 3 |
| 7 | 18 | 0 | Federal | ORGANIZAT... | 0.96220861... | 1 |
| 8 | 18 | 0 | Federal | ORGANIZAT... | 0.01161753... | 2 |
| 9 | 19 | 0 | Reserve | ORGANIZAT... | 0.96220861... | 1 |

Figure 4.7: The *TokenTable* schema table showing the first three tokens extracted (*Lehman, Brothers, Federal*) with k=3

### b) Visualization of the classification results

This process takes place after classification has finished and all the appropriate structures have been created. Because our platform is user oriented, the recognized named entities should be visualized through the *IExtractor GUI*. For this purpose, the **best only** sequence is passed back to the *Presentation Layer* which examines the tagged string sequence and with the help of the *inlineXml* format, it distinguishes the tagged entities. As explained in 4.2.2.1 the tags are expressed with angle brackets (< >). An example of a classified tagged sentence is:

"*Several possible plans emerged from the talks, held at the <ORGANIZATION>Federal Reserve Bank of New York</ORGANIZATION> and led by <PERSON>Timothy R. Geithner</PERSON>, the president of the <ORGANIZATION>New York Fed</ORGANIZATION>, and Treasury Secretary <PERSON>Henry M. Paulson Jr.</PERSON>".*

In order to identify the entities and the tagged classes, the *Presentation GUI* initially constructs a regular expression of the form $<(X_1 | X_{2|} ... | X_N)>$ where $X_i$ is the i-th entity and N the number of all entity categories of the model. (e.g. PERSON, ORGANIZATION, LOCATION, STREET NAME, etc.). The "|" symbol represents

the logical disjunction (OR) operator. The purpose of this regular expression is to catch all the tagged entities in the *insideXMLformat* of the string. Subsequently, the regular expression is compiled through the `Pattern` class which represents the regular expression's compiled representation. The resulting pattern is used to create a `Matcher` object that can match arbitrary character sequences against the regular expression. Once we create a logical expression for the end of the tag notations "< / >", we run the `Matcher` object on both compiled expressions and we mark the start and end positions inside the string sequence where each pattern is identified. The substring these two positions define produces a recognized entity. The only thing that has remained is to define the `AttributeSet` object that will add style properties to the string entity. After the addition of Color properties to every entity class, the enriched string entity is passed to the editor, which finally presents the initial text or html contents.

### c) Construction of the Probability Bucket and Histogram creation

At the time we construct the `PriorityQueue` structure containing all the `TokenTableRecord` objects, we create a 10 length array; a local list that creates all the `TokenTableRecord` objects for each sentence and parallel to this list an array that holds **only** the probabilities for every sentence. The goal is to have a structure that will hold, for the probability range 0 to 1 with a 0.1 step, all the tokens of the sentences that were assigned a probability from the classifier that belongs to that range. For example, the tokens of sentences assigned with a probability of 0.45 will be saved to the `BucketCounter's` position 5, because this position is addressed to any probability in the 0.4 - 0.5 range.

Therefore, the construction of the `BucketCounter` array we described goes as following: First of all, since every position in the array must contain probabilities of a specific range, before insertion of any tokens in the array take place, we use a simple method called `findBucketPos` to return an index that represents the position of the token given the probability assigned to the sentence. As soon as we get the position index in the array, there are two options. Either the bucket in this position is empty so we first need to instantiate a list that will keep the lists of sentence tokens. Otherwise, the array contains at this position at least one list of tokens, so the outer list has already been instantiated. Finally we put the list of `TokenTableRecord` objects inside the outer list, and we keep on building the `BucketCounter` array by processing the remaining sentences. The complete process is illustrated in Figure 4.7

The remaining step to complete this procedure is the creation of the histogram. Its purpose is to demonstrate the distribution of the sentences over the probabilities. Particularly, by finding how many sentences belong to each probability portion, we get a visual way to evaluate the precision and accuracy of our model. For example, the more many sentences belong to higher probabilities the best our classification model is.

Concerning the technical details of the histogram creation, we have used the `JFreeChart` Java open-source library that allows the creation of a wide-variety of charts. Our `HistogramClass` creates an `IntervalXYDataset` object that creates the data that the histogram will demonstrate. This takes for input the probabilities array, the number of bins (how many probability slots we wants to have) and the minimum and maximum numbers in *x* axis (minimum and maximum probabilities). After the creation of the dataset we instantiate a `JFreeChart ChartPanel` object to visualize the histogram. Figure 4.8 depicts the result of this process.

The design of the histogram class is such to perform some actions when certain actions take place. Specifically, our `HistogramClass` listens to `WindowEvents` and performs the two following tasks. As soon as the user has pressed the "Close" button the `windowClosing`( ) method is called that shows a popup prompt message window as shown in Figure 4.9. The user is asked to enter a probability as an upper bound in order to highlight the tokens that have been assigned probabilities equal or less than this threshold. The probability entered is the base for the highlight process. This process, triggered from the `windowClosed`( ) method is the center of our focus in next section.

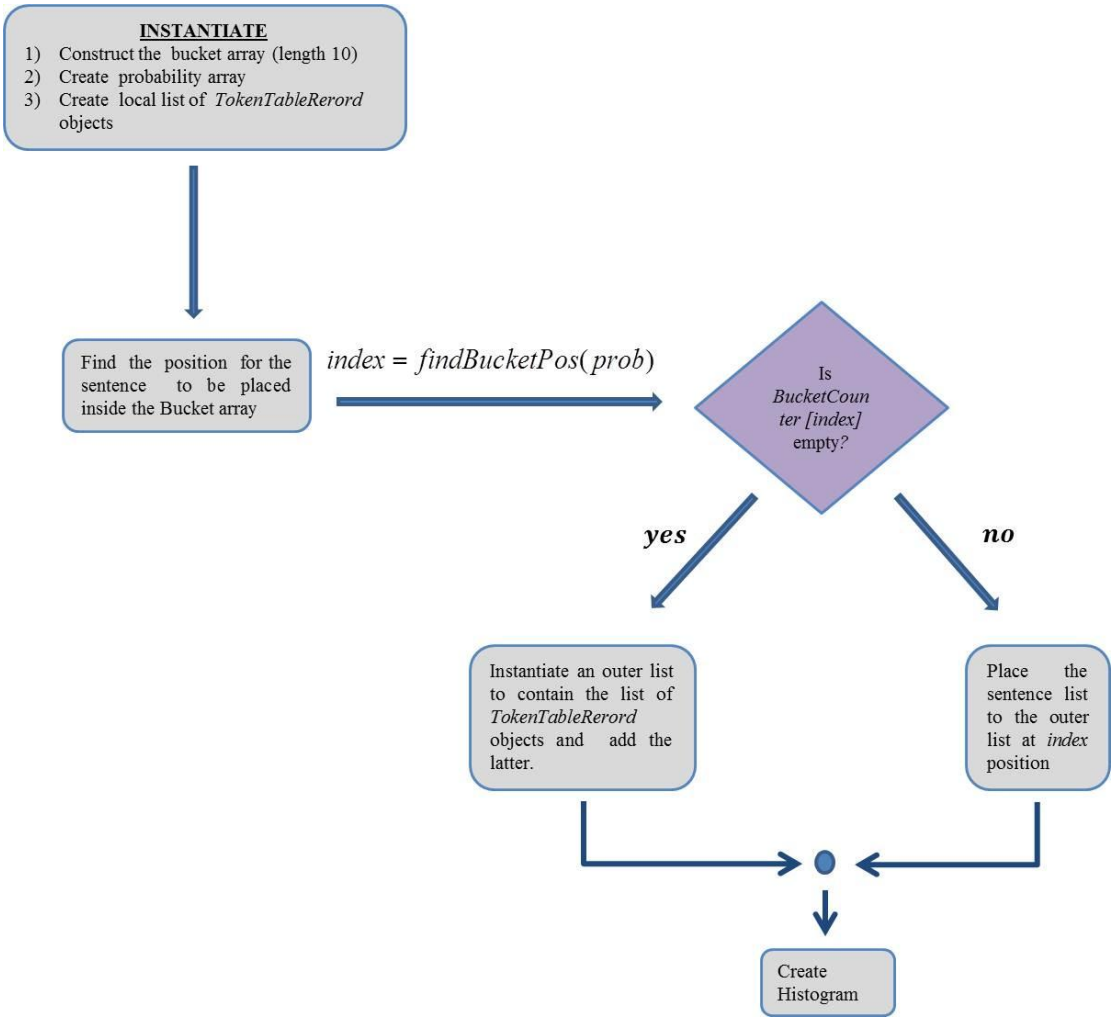


Figure 4.8: The process of constructing the Probability Bucket and the Histogram

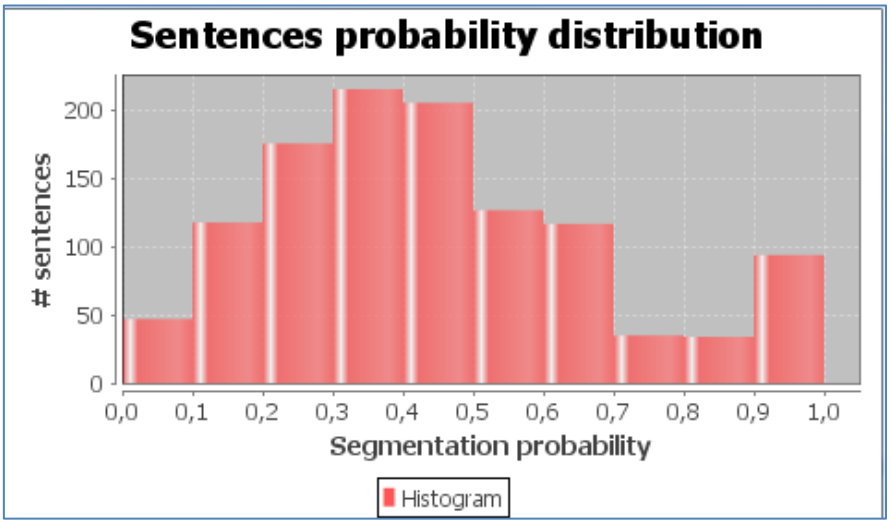


Figure 4.9: The histogram presenting the probability distribution of sentence estimation.

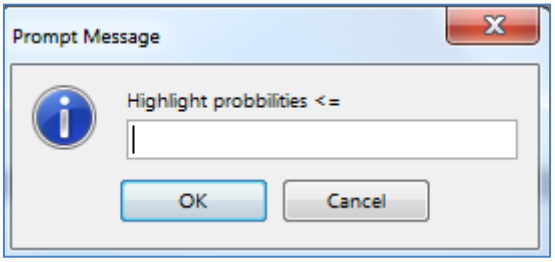


Figure 4.10: The prompt message user dialog.

**d) Highlight "weakest" sentences given the user's option.**

Previously we discussed about the need to demonstrate the sentences the system has assigned with small probability scores in order to estimate how well the classifier of our model has performed. In conjunction with the quantified result of the histogram, it is desirable to view via the editor the "weak" sentences. The user's input via the prompt message, defines the upper bound from under which we consider a sentence to be "weak". Ideally, the user has observed the histogram and identified approximately the bucket where there are many sentences gathered. Of course, this interaction with the user is optional.

In order to highlight the "weak" tokens, firstly we construct a `HashMap` structure named `highlighenedTokens` that will keep tokens of the sentences we want to highlight based on the user's input; a `HashMap` is chosen to map an encoded String identity of each token to its `TokenTableRecord` object. Specifically, $key = characterOffsetBegin + "\_" + characterOffsetEnd$ . The `HashMap` structure, even though it is created during the highlight procedure, it is not used at the time, but it

serves as the basic structure for the next procedure that presents all the alternative labels assigned to every token by the classifier. Furthermore, user's given probability is passed to the familiar `findBucketPos` function that returns the position until which we will scan the `BucketCounter` array to track the tokens. Therefore, we process the array and for each position until the `BucketCounter's` index, we take the list of lists of `TokenTableRecord` objects (list of sentences) and for each list of `TokenTableRecord` records we take every token's start and end character offsets. Having these, it is simple to highlight the tokens by setting a different `AttributeSet` on data with the Color parameter we prefer. *IExtractor GUI* highlights tokens using yellow color.

### e) Illustration of the k-best sentence segmentations and database update

This is the last process of the *Post Classification Processing Unit* and its result is forwarded and demonstrated to *Presentation Layer*. It follows as an aftermath of the highlighting procedure, but it is triggered **only** if the user demands it. As soon as the "weak" entities are illustrated through the editor, it is desirable to view all the top-k labels assigned to each one of them and inspect them. In addition to marking the "weakest" entities, entities the model has assigned a different label/class from the top-1 segmentation are being underlined. This action encourages the user to seek other options for this token. Therefore, if the user decides to view these other options by clicking on a highlighted entity a window popup appears presenting the k-best entities list for the word the user clicked on. This process is enabled by the services of the Caret listener we have previously discussed.

Concerning the implementation details, similar to the construction of the `highlighenedTokens HashMap`, another `HashMap` is constructed mapping every highlighted token to a list that contains the k-alternative `TokenTableRecord` objects we discussed. Then, this k-best tokens list is being scanned and for every token it is checked whether there any other classes assigned to it. As soon as these actions are executed, caret events present the k-best classified entities when occur. The `caretUpdate` function when triggered calls `createPopupWithKbestAlternatives`, which identifies whether the caret event was caused by a mouse action over a highlighted entity or not. In case a "weak" entity has been clicked, the *JOpionPane* input Dialog of Figure 4.10 is demonstrated containing the k-best entities. The user views the possible label assignments for the word she has clicked, and has the option of choosing a different one that suits her belief about the word class. Once the user has made an option, the `findTokenAndUpdateTokenTable` is called that searches through the general structure that contains all the `TokenTableRecord` objects to find the token and replace its label class with the user option. The token is identified uniquely from its *sentenceId* in conjunction with its *pos*.

This ability offered by the ARAMIS platform is an important one, because it makes the system quite flexible by enabling both automatic recognition and human annotation and correction of the automated results the model produces. Therefore, the system becomes more reliable.

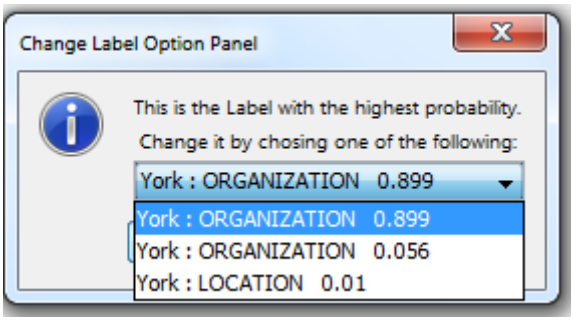


Figure 4.11: The input dialog for viewing the k-best labels the model has identified

### 4.2.2.3 Stage 3: Schema Creation

The final stage of the Real Time In-Memory Processing concerns the creation of the schema tables inside Apatar's Database. Because the nature of the extracted tokens offers many options for the attributes (columns) of the database tables, we have decided to create two kinds of database schema: **a)** *TokenTable*, a predefined schema table and **b)** a dynamically defined schema table or tables (during runtime), that contains separately all the entries of an extracted entity.

As it is observed from Figure 4.11, *TokenTable* contains for every recognized entity all the information extracted that exist inside a *TokenTableRecord* object and are explained in 4.2.2.2.

The other schema table the ARAMIS platform supports is a table that contains all the appropriate information for a recognized entity, that is the *token_id* the *Entity_Name* and *prob*. For example, in a task that we recognize a PERSON and a LOCATION entity, we have two entity tables, PERSON_TABLE and LOCATION_TABLE. This type of tables are defined dynamically depending on the entities that the model has been trained to recognize every time.

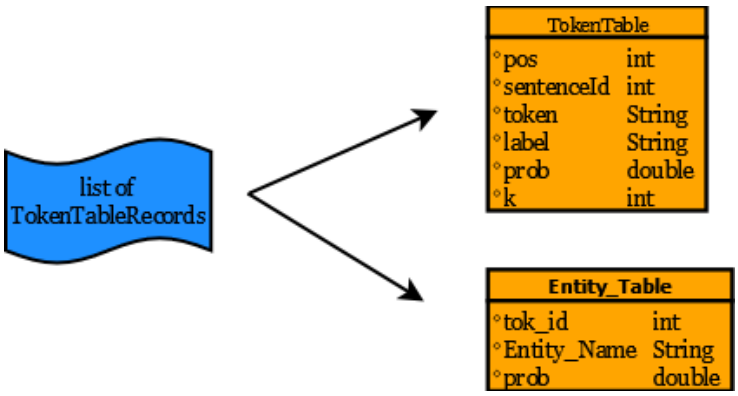


Figure 4.12: The database schemas ARAMIS supports.

After the user makes a table selection and schemas have been created, the data integration and transfer of the ARAMIS platform is ready to unfold.

# Chapter 5

# Demonstration

To demonstrate our work, we will focus on presenting the IExtractor functionality. The ieTrainer widget does not present its output in a graphical way since it just produces the trained model file. The demonstration is based on **two scenarios.** The first is to recognize *Author, Editor, Date, Topic* and *Title* from a file containing raw *DBLP* Computer Science Bibliography. To accomplish that, the user creates a mashup application, loads the IExtractor widget into ARAMIS platform and makes the appropriate configuration by loading the file she prefers, the trained model, she has previously trained and pressing the "Run NER" button. The output of the IExtractor GUI is presented in Figure 5.1

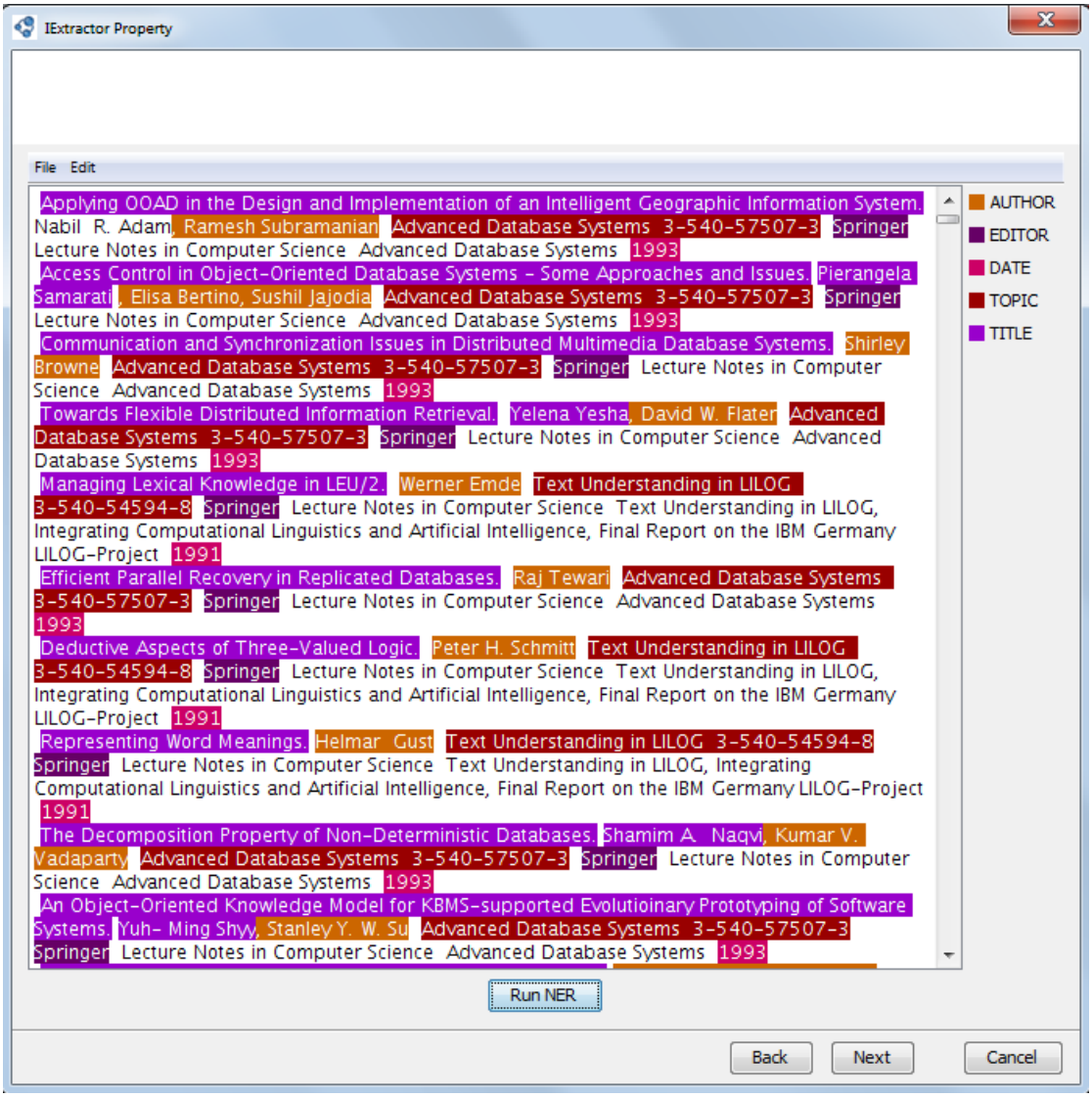


Figure 5.1: IExtractor output based on scenario 1.

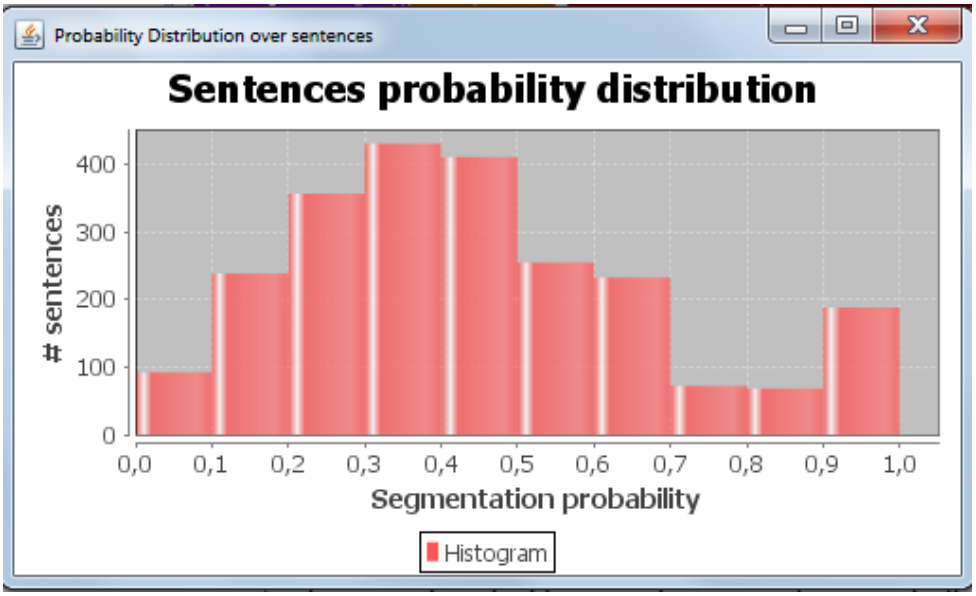


Figure 5.2: Histogram for the extraction results of scenario 1.

At the same time the histogram is presented automatically to the user (Figure 5.2). The user observes the sentence distribution based on the probabilities assigned to them by the model. When the histogram window is closing, a prompt appears encouraging the user to enter a probability number (Figure 5.3)

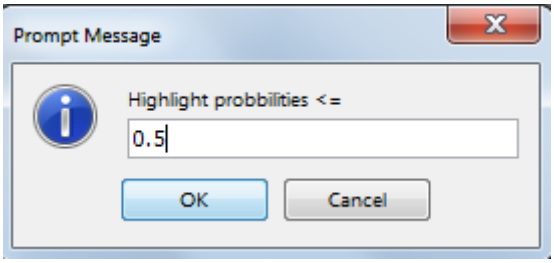


Figure 5.3: Prompt window of IExtractor.

The user enters 0.5 as the probability value and IExtractor GUI presents all the highlighted entities (“weak entities”) (Figure 5.4)

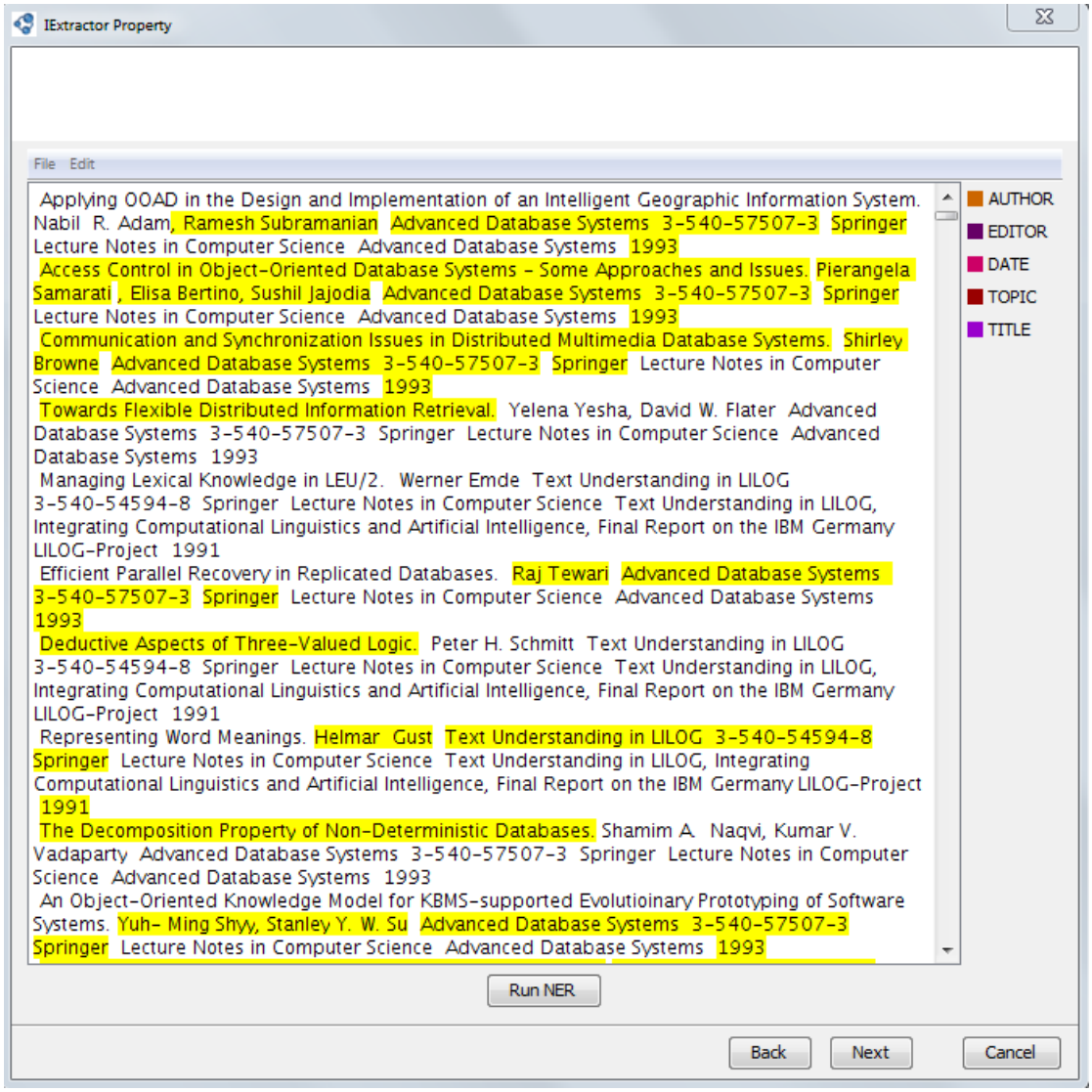


Figure 5.4: IExtractor GUI demonstrating the "weak" entities.

Finally the user "clicks" on an underlined word and views all the entities the model has recognized having a different label. (Figure 5.5)

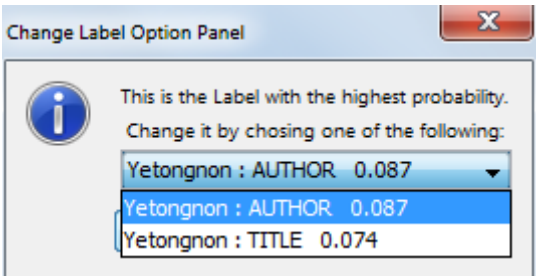


Figure 5.5: The window presenting the k choices for an underlined token.

The second scenario is the following one: A user wants to extract **only** location information from apartments in Athens from Craigslist website and illustrate it in Google Maps. To achieve this goal, she creates a mashup application (Figure 5.6) and makes the proper configurations in IExractor GUI inserting the website url. She adds a *Transform* widget in order to keep only the location information from the LOCATION table. Finally she uses a *Table* widget for putting results in a supported format from the Google maps widget. The output demonstrated in Google Maps is presented in Figure 5.7

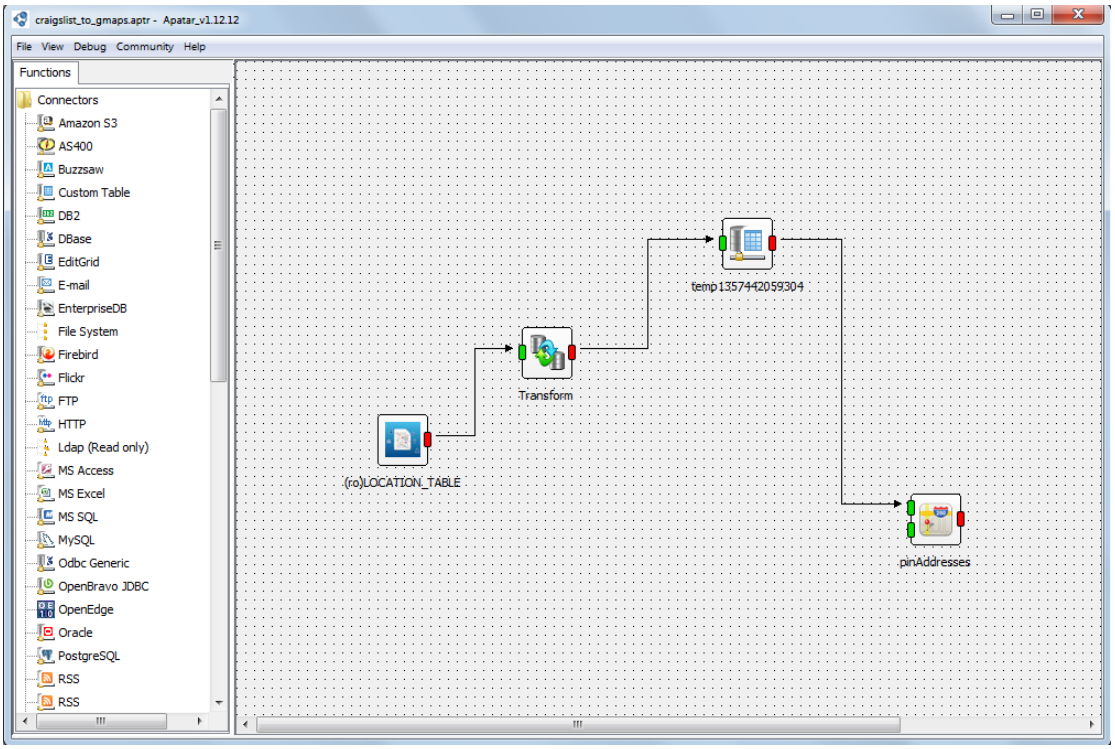


Figure 5.6: The mashup application for scenario 2

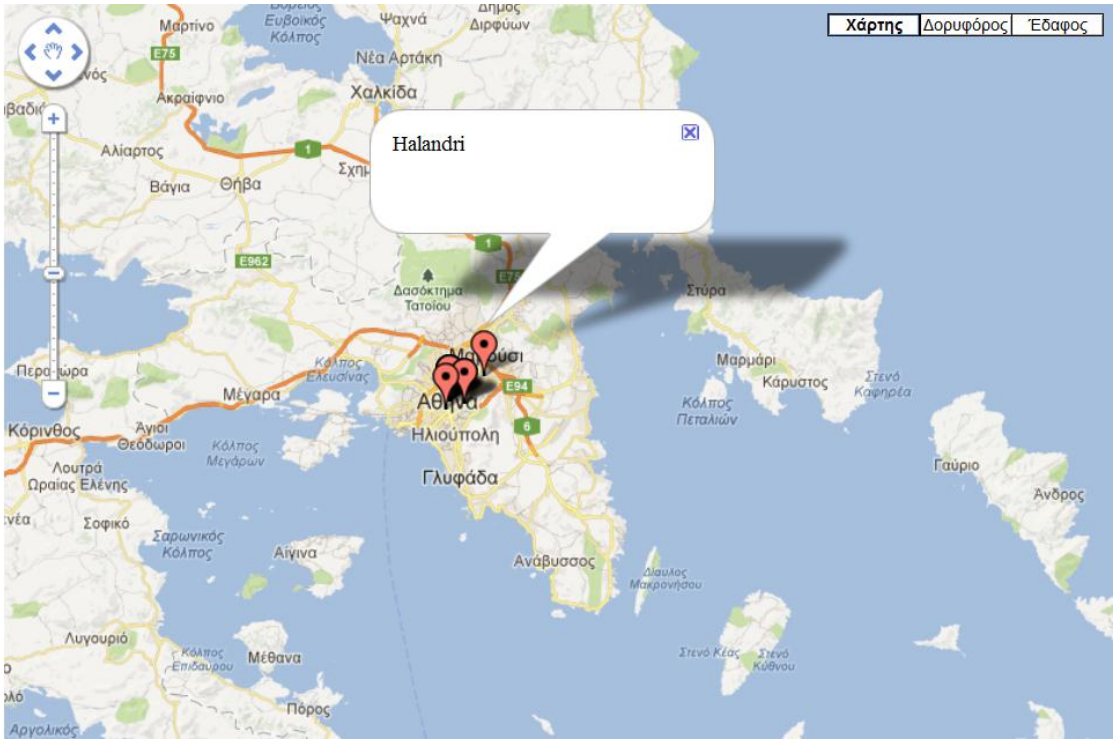


Figure 5.7: Result map

# Chapter 6

# Conclusions and Future Work

Based on the Apatar open-source mashup platform, we designed and implemented ARAMIS, an information extraction system that implements supervised named entity recognition based on linear-chain Conditional Random Field (CRF) sequence models. Such functionality offered by our platform enables the users to train models based on datasets of their interest and extract information from local files or websites based on these user trained models or some already trained classifier models provided. In addition, ARAMIS provides an explicit representation layer and a post-classification processing unit that benefits users for visualization and data validation purposes. Considering the vast amount of data being manipulated inside the APATAR platform, we have come up with this idea in order to automatically populate Apatar's databases inserting uncertainty over the extracted data due to the probabilistic nature of information extraction used for the extraction. In this way we enable database operations over probabilistic data. Another important feature we consider is user's ability to self-annotate the extracted entities in case the model provides alternative solutions for mislabeled tokens.

Besides the work presented in this thesis, there are many additions and optimizations that could be done. First of all, concerning the training of the model, an automated feature generation system could be designed in order to produce automatically the appropriate features based on the user's annotations in the training file. Even though we provide in ARAMIS some automatically produced features, these are some very basic options to perform the extraction without taking into account the user's preferences concerning what patterns to take into consideration during the learning procedure.

In addition, some procedures could be added to *Post Classification Processing Unit.* For example, when *k* parameter is greater than 1 the best segmentation's probability could be boosted with the probability of the other k-1sentence segmentations, in case their label sequence remains the same with the best.

Finally, the *IExtractor GUI* could be enriched in order to gain more flexibility and interactivity, such as maintaining the label information after the highlighting procedure and replacing the `JEditorPane` with another editor that allows better representation of websites.

# Bibliography

(

[1] Sunita Sarawagi, "Information Extraction," *Foundations and Trends in Databases*, vol. 1, no. 3, pp. 261-377, 2008.

[2] Daisy Zhe Wang, Eirineos Michelakis, Michael J. Franklin, Minos Garofalakis, and Joseph M Hellerstein, "Probabilistic declarative information extraction," in *ICDE 2010*, Long Beach, California, USA, 2010, pp. 173-176.

[3] Christopher Manning. (Winter 2012) Stanford University. [Online]. http://www.stanford.edu/class/cs124/lec/Information_Extraction_and_Named_Entity_Recognition.pdf

[4] John D Lafferty, Andrew McCallum, and Frenando C.N. Pereira, "Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data," in *ICML '01 Proceedings of the Eighteenth International Conference on Machine*, 2001.

[5] Charles Sutton and Andrew McCallum, *Introduction to Conditional Random Fields for Relational Learning*.: MIT PRESS, 2006.

[6] G.D Forney, "The Viterbi Algorithm," vol. 61, no. 3, pp. 268-278, March 1973.

[7] Jenny Rose Finkel, Trond Grenager, and Christopher Manning, "Incorporating non-local information into information extraction systems by Gibbs sampling," in *ACL '05 Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics* , University of Michigan, USA, 2005.

[8] Sanita Sarawagi. CRF Project Page. [Online]. http://crf.sourceforge.net/

[9] The Stanford NLP Group. The Stanford Natural Language Processing Group. [Online]. http://nlp.stanford.edu/software/CRF-NER.shtml

[10] Nicole Carrier, Tom Deutch, Chris Gruber, Mark Heid, and Lisa Lucadamo, "The business case for enterprise mashups.," 2008.

[11] Mohamed A. Soliman, Ihab F. Ilyas, and Mina Saaleeb, "Building ranked mashups of unstructured sources with uncertain information," vol. 3, no. 1-2, pp. 826-837, September 2010.

[12] David E. Simmen, Frederick Reiss, Yunyaou Li, and Suresh Thalamati, "Enabling enterprise mashups over unstructured text feeds with InfoSphere MashupHub and SystemT," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data* , New York, 2009, pp. 1123-1126.

[13] Giusy Di Lorenzo , Hakim Hacid, Hye-young Paik, and Boualem Bentallah, "Data integration in mashups," in *ACM SIGMOD Record*, vol. 38, New York , NY, USA, March 2009, pp. 59-66.

[14] Apostolos K. Nydriotis, "Dynamic Web Service Mashups," Chania, December 2010.