Introduction to Grid Computing

Lab Assignment Report

Deployment of a Language Detector Grid Service

Roberto Valenti, 0493198
Felix Hageloh, 0425257

Introduction

The aim of this assignment was to build a grid service and to aggregate the service with another to provide additional, higher-level services. There are two issues to be considered, namely how to create and deploy a grid service and what kind of service create, which would be a useful grid service. The way to create and deploy a grid service is explained in detail in Sotomayor's globus 4 tutorial (http://gdp.globus.org/gt4-tutorial/singlehtml/progtutorial_0.2.html) which describes the deployment process in five steps. For the service itself we used our background as AI students to come up with an idea that would combine grid computing with AI. We chose to create a language detector grid service, for which we thought it fits nicely with the grid concept for following reasons:

- Language Detection is a necessary first step in a multitude of applications like email filtering, information retrieval and spell checkers. These applications could access our service in a web service fashion, or, if they reside on the grid themselves, in a grid service fashion.
- Creating a language detector is a typical machine learning task that may require huge amounts of data for training. This can be very time consuming when done on a single machine. However, there exits lots of possibilities to distribute this task on several machines which is essentially what grid computing is for. Moreover, the big amount of data is difficult to obtain, but on a grid where data is shared among different scientific communities one may find it easier to get the required data. The same is true for testing a classifier.

The language detector that we implemented features 22 European languages (including Russian). The language detector is now available as a grid service and can be accessed via a simple client that we wrote. We also performed few tests to investigate the performance of our language detector, and we found that it works sufficiently well (with accuracy approaching 100%) for texts consisting of 4 words and more.
Obviously the service can now be easily combined with other services, but due to time constraints we were not able to demonstrate this.
In the rest of the paper we will first talk about the general steps to create a grid service. Then we will go into the basic design of our service including the service interface. Next we will discuss the implementation of a language detector and give some theoretical background. Furthermore we will present some results that we obtained from testing the service, including some typical scenario of using the client to connect to our service. Finally we will talk about future work and possible extensions and come to a conclusion.

Deploying Steps

The first step to deploy a service is set up a secure connection to the cluster. This is done by using a RSA private and public key system. Once this secure channel to the grid is established, we need a certificate for our remote account and a certificate to authenticate our host. Those certificates require signing from a CA (certificate authority) in order to be valid.

After dealing with all the grid security issues, it is possible to start to implement and test the service. Other than the source code of the service, additional definition files are required, which allows the deployment and the creation of the stubs and portTypes.

Those files are:

- WSDL: Describes the service interface in XML
- WSDD: to tell the Web Services container how to publish our web service
- JNDI: to organize and locate components of the java implementation of our grid service
- QNames: To facilitate the service implementation, providing shortcuts to qnames

All these files are then combined and compiled into a single Grid Archive (GAR) file using the ANT tool. We used the script included in the Sotomayor's tutorial to automate all the additional procedural steps to obtain the GAR file.
The last step is to deploy the created GAR file. This is done by running the command **globus-deploy-gar** followed by the name of the created GAR as the globus user. If the service is correctly deployed, the last step is to start the container through the command **globus-start-container –nosec**. After those few steps, the service is ready to be used by the respective client.
All the required files to compile and create a deployable gar of our service can be retrieved at http://staff.science.uva.nl/~rvalenti/uva/gc/LanguageDetectorServiceSRC.zip
A readme.txt file is included to illustrate the steps required in order to compile, deploy and use the service (grid security issues excluded).

Service Design

The basic design of our service is very simple. The main task of our service is to accept a request containing the unknown text and to return a response that contains the detected language.
Obviously there isn't any need for a more complex interface or added states for this service. However, for the learning factor we wanted to make a slightly more complex grid service that actually distinguishes itself from a simple web service. Thus we decide to add some "dummy" states just to learn the concept. The states we added are

- lastOp: the last operation of the detector, e.g. "*<text>* detected as *<language>*"
- timesUsed: the number of times the service was used since the first start

The following diagram illustrates a typical use case of our service and gives an overview of the general design.

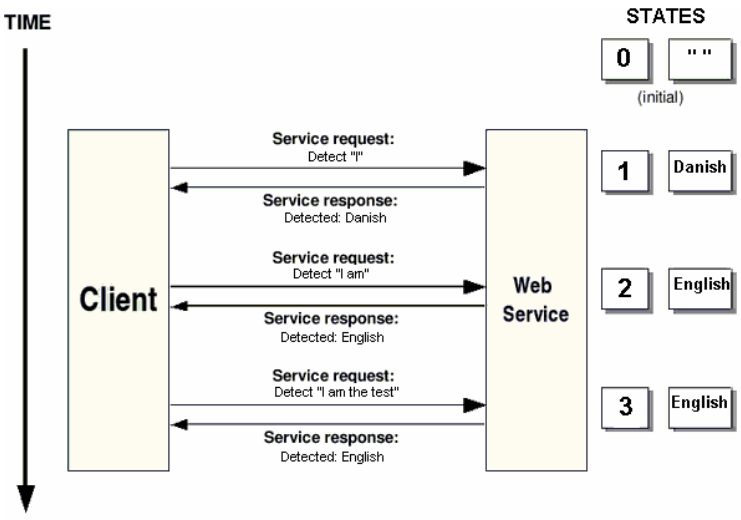


Figure 1 – ***typical use case of the language detector service***

So our service has one method called “detect” and two resource properties “lastOp” and “timesUsed”. This is defined in the WSDL file which can be found inside the downloadable source package. The WSDL file also defines the messages that are sent between the client and the service together with their content type. For our service the client sends a request containing a string (the unknown text) and the service responds with an answer also containing a string (the detected language).

Implementation

The implementation of the language detector itself follows a basic machine learning and Bayesian decision process. Having a set of classes C, each class is represented by a model that is defined by its model parameters. A training set of samples for each class is used to estimate or learn these model parameters. The task is then to assign an unknown sample S to one of the classes. This is achieved by comparing S with all class models and picking the one for witch S matches the most, or more precisely, for which S has the highest posterior probability given the class model. In the context of this paper, the set classes consists of languages that can be identified and a sample S is a piece of text whose language needs to be identified. Languages are modeled using a character level third order Markov model.

Markov Models

Markov models are used to describe stochastic processes with the limiting assumption that the probability distribution of a next state only depends on the current state. In a language modeling context, Markov models are a convenient way to represent stochastic grammars. Grammars essentially consist of rules that, given a set of symbols or words A, define valid strings or sentences S that can be constructed in a given language. In this context, the states of a Markov model consist of symbols or characters, so that the occurrence of symbol in a string determines the probability of the next symbol generated in a string. Hence the probability of a string S being generated by a certain grammar is given as:

$$\mathrm{P}(S) = \mathrm{P}(s_1, s_2, ..., s_n) = \mathrm{P}(s_1)\prod_{i=2}^{n} \mathrm{P}(s_i \mid s_{i-1})$$

The definition of a Markov model can be extended to not only the current state determining the next state, but also a history of k last states. Such extended Markov models are also referred to as Markov models of order k. The probability of a string S being generated by a k^{th} order Markov model is then given by

$$\mathrm{P}(S) = \mathrm{P}(s_1, s_2, ..., s_n) = \mathrm{P}(s_1, ..., s_k)\prod_{i=2}^{n} \mathrm{P}(s_i \mid s_{i-k}, ..., s_{i-1})$$

The distribution of the initial states $P(s_1,\ldots,s_k)$ and the so called transitional probabilities $P(s_i|_{si-k},\ldots,s_{i-1})$ are the model parameters that need to be learned from the training data. In practice it is convenient to pad all strings with k special characters so that $P(s_1,\ldots,s_k)$ is one for all strings. This also ensures that model also works for strings of length less than k.

Kth order Markov Model can give a more accurate description of the real world as they use a weaker independence assumption than a standard Markov Model. On the other hand, using a large k increases the number of probabilities that have to be learned and thus increase the computational complexity and space requirements. In many cases of language modeling 2^{nd} order Markov Models have proven to be a good tradeoff between accuracy and efficiency and so we also chose to use 2^{nd} order Markov models for our language detector.

Having decided on a 2nd order Markov models we can estimate transitional probabilities through simple counting methods. For each character s_i in the training set we have to consider its previous two characters s_{i-1} and s_{i-2} and calculate the probability of 's_i follows s_{i-1},s_{1-2}' as the total number of times we encountered the sequence s_i, s_{i-1},s_{i-2} divided by the total number of times we encountered s_{i-1},s_{i-2}. Formally

$$\mathrm{P}(s_i \mid s_{i-1}, s_{i-2}) = \frac{\#(s_i, s_{i-1}, s_{i-2})}{\#(s_{s-1}, s_{i-2})}$$

For the implementation it is convenient it store such transitional probabilities in the form of a rule triple

(CONTEXT, CHAR, PROBABILITY)

so for each character s_i in the training set we store ($s_{i-2}s_{i-1}$, s_i, $P(s_n|s_{i-1},s_{i-2})$). The set of rules of for characters in the training set then constitutes a language model L.

If we the training set for a certain language consists of the text '*Test text*' we would first split it into words and then pad each word with the two occurrences of the special character '^'. Hence we would learn following rules

```
( ^^, t, 1.0 )
( ^t, e, 1.0 )
( te, s, 0.5 )
( es, t, 1.0 )
( st,  , 1.0 )
( te, x, 0.5 )
( ex, t, 1.0 )
( xt,  , 1.0 )
```

Classification

We use a common Bayesian decision rule of assigning a class label C to an unknown sample S, which is given by

$$C = \max_c \mathrm{P}(S \mid L_c) = \max_c \prod_{i=3}^{n} \mathrm{P}(s_i \mid s_{i-1,} s_{i-2} \mid L_c)$$

After training the classifier we keep all language models in memory in form of nested hash tables. An incoming piece of text is then matched against each language model and the probability of the text being generated by the model is calculate. We then simply pick the one with the highest probability and return it as the detected language. Options to return several detected languages when the probabilities are similar or equal are not featured in the current implementation but are trivial to do.

Using the language model from the previous example, the probability of the string 'test' would be

$$P(test|L) = P(t|\text{^^})P(e|\text{^}t)P(s|te)P(t|es)P(\_|st) = 1*1*0.5*1*1=0.5.$$

Other Issues

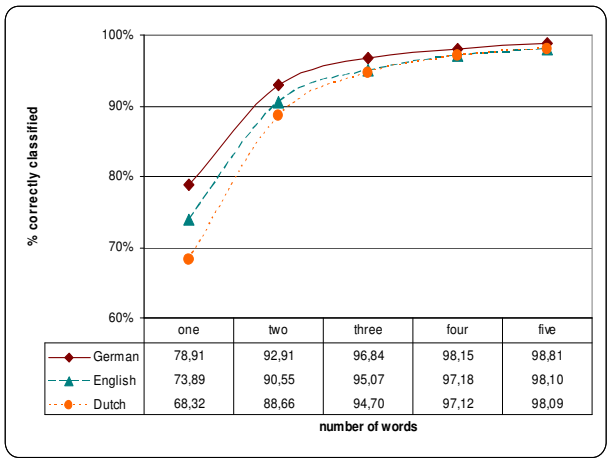
A common problem in k^{th} order Markov models is that of unseen or unknown k+1 sequences. This means that, when trying to match a k+1 character sequence of a new text with the rules in our language model, we could encounter a character combination that was not learned. Assigning 0 probabilities to such unknown character combinations causes the entire text to have 0

probability, which is usually not desirable. This phenomenon is commonly known as the “Zero Frequency” or “Sparsity” problem. As it is a common statistical inference problem, there exist several proposed solutions, such as discounting or smoothing methods. We use a very simple smoothing method for this language detector, but will not go into details here.
Another issue that needs to be considered is the character set and encoding used for different languages. One solution is to use UTF-16 encoding consistently to avoid any encoding switching and/or resulting errors.

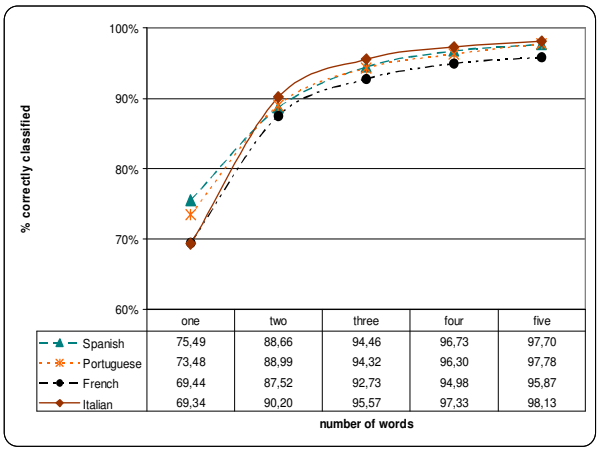
Experiments and Results

To evaluate the performance of our language detector we used a corpus of roughly 9MB containing 9 languages. Always two thirds of the corpus were used for training and one third for testing in a cross validation manner. While our actual implementation features 22 languages and was trained on a substantially larger corpus for some languages, this will still give an insight of the general performance of the detector.

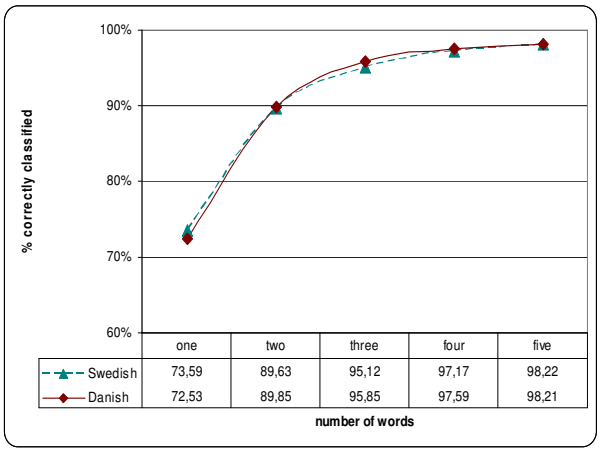
Figures 2 through 4 show the performance as the percentage correctly classified versus different length texts for all nine languages.



	one	two	three	four	five
German	78,91	92,91	96,84	98,15	98,81
English	73,89	90,55	95,07	97,18	98,10
Dutch	68,32	88,66	94,70	97,12	98,09



	one	two	three	four	five
Spanish	75,49	88,66	94,46	96,73	97,70
Portuguese	73,48	88,99	94,32	96,30	97,78
French	69,44	87,52	92,73	94,98	95,87
Italian	69,34	90,20	95,57	97,33	98,13



	one	two	three	four	five
Swedish	73,59	89,63	95,12	97,17	98,22
Danish	72,53	89,85	95,85	97,59	98,21

Figures 2- 4 - *% correct classified vs. number of words for nine languages*

We can see from figure 1 that the performance on one word texts is fairly medium for all languages - German performing the best with 78,91% accuracy and Dutch the worst with 68,32% accuracy. For all languages we could observe a mayor performance increase when going from one to two word texts, jumping from 72,78% on average to 89,66% on average. Also we can see that the difference in performance between the languages diminishes as texts get larger. For five word texts the performance approaches 100% for all languages

The final image shows a screenshot from an interaction with the client.

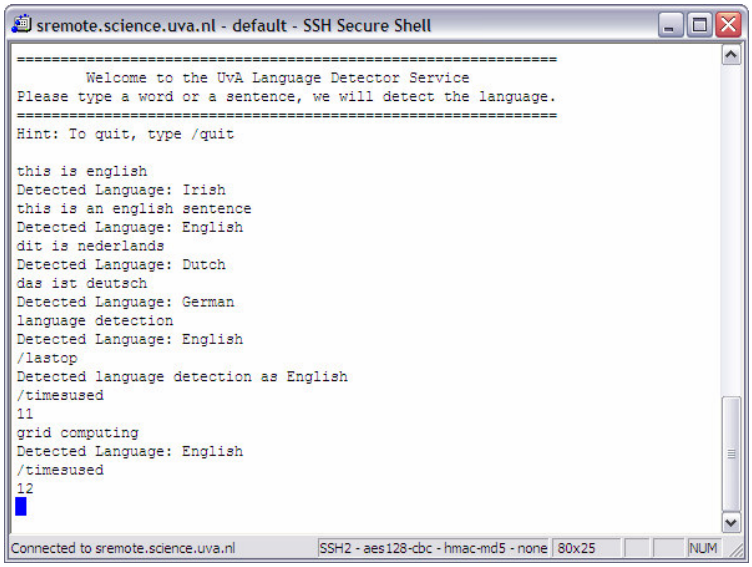


Figure 5 – *client screenshot*

Future Work

As stated in the introduction, the implemented grid service allows for straight forward extensions. Email filtering, information retrieval and spell checkers are applications or grid services which could contact our service to perform parts of their tasks.
We didn't manage to create a training and testing service for our language detector, which could have sped up the training and the testing significantly. In fact, this phase was performed on a single machine and required several hours OOL (Of Our Life). Another nice extension of the service could be a web interface that will allow user from internet to use the service for their purposes.

Conclusions

We successfully managed to create and deploy our own web service using the globus toolkit 4. In order to achieve this we followed the Sotomayor's tutorial, but we manage to use our own directory structure and naming, breaking loose from the tutorial's structure. Our service is a good example of an of Artificial Intelligence application that could be a useful grid service and benefit from grid technology. Even if we didn't use and exploit all the possibilities that the Grid offers, we feel that we've got a good hands-on experience by working on this project. We have also brainstormed on how our service could be use, and we point out a lot of possibilities to integrate and/or extend the implemented service.