ABSTRACT
Open source software (OSS) is becoming more and more popular and available for the average person. The key to success of OSS projects are the well organised communities. These communities are challenged by their unique way of working. Modularisation is used to cope with one of their challenges, working together while geography distributed. The modularisation into packages makes it easy to share code with each other but with this code sharing a few problems arise. These problems, like code that isn’t up to date and not knowing of new functionality, can be dealt with by checking if Conway’s law is satisfied. In this paper we try to find the correct communicational network by only using communication that is about the dependencies. The communication is sorted with the help of classification algorithms. Mailing lists from the OSS Apache HTTPd project are used to demonstrate the method.

Keywords
Open source software, Canway’s law, inter package, package dependencies, communication, Apache, text mining

1. INTRODUCTION
The development of open source software (OSS) is different compared to the development of traditional software. The geographical distribution of the developers and the organisational structure they work in are examples of these differences. For instance, most communities don’t have a single manager but are coordinated by a core group of developers [24].

To make it easier to control a project, projects are divided into modules [10][13]. All modules, or packages, combined make the software code. Between packages there are dependencies because of the concept of code sharing and reusing code. For example, a chat application uses the network package to make it possible to make a connection over a network [11]. The packages can be represented as nodes and the dependencies as edges [31]. This will result in a network with packages and the dependencies between them as shown before by MacCormack [21] and others.

The organisation structure of OSS projects can be represented by the communication between developers [30].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

This organisational structure can also be represented as a network with the developers as nodes and communication between the developers as edges, a sociogram [29]. The communication between the developers is typically over the internet by email, Internet Relay Chat (IRC), forums and other digital communication [17].

One of the problems the OSS communities have to deal with is the problems that arises with code sharing. Because of the modular construction, there are packages that depend on other packages. Code changes over time and the packages depending on the changing code have to deal with this. Keeping up to date in terms of functionality and security can be a challenge. Here Conway’s law [15] can help. Conway states that the organisational structure must be a copy of the design structure, as illustrated in Figure 1. So, when developers are working on code that is dependent on each other and something changes, they should communicate this with the developers working on the depend code.

While extracting the technical dependency structure can be done relatively easy and has been done before [5], representing the organisational structure of a community can be a bit harder. The organisational structure can be represented by the communication, but the only communication that is relevant when comparing it with the package dependency structure is the communication about these dependencies. Classifying every communication message as being about a dependency or not will help making a correct representation of the organisational structure.

The classification of the communication can be done by natural language classifiers [28][19]. Classifiers as the Naive Bayes classifier, a decision tree, k-Nearest Neighbour classifier and Logistic Regression can be used to determine whether the communication is about a dependency or not.

Using the OSS Apache we try to identify if there are any misfits between the organisational structure and the design structure. More exactly we will use the HTTPD project, which implements a ‘safe, efficient and extensible’ HTTP server [4] and is one of the core projects of Apache. Apache was chosen because all mailing list are openly available the use of text mining tools on these mailing list has been
The following questions will be answered:

1.2 Research questions

1. How does the communication network look among the packages within the Apache HTTPd community?
2. How does the structure of package dependencies look of the Apache packages?
3. Does this meet the expectations of Conway’s law?

1.1 Problem statement

So, Conway’s law states that the organisational structure and the technical structure should look the same. Because of misfits problems occur like not being abreast of code changes, using code that isn’t up-to-date functionally or using code that isn’t safe any longer.

With this research we hope to demonstrate that the communication network of the Apache OSS fits the inter package code dependencies. If there are any misfits, these mismatches can be a source of problems and project managers should look at these mismatches.

1.2 Research questions

The following questions will be answered:

1. How does the communication network look among the packages within the Apache HTTPd community?
2. How does the structure of package dependencies look of the Apache packages?
3. Does this meet the expectations of Conway’s law?

2. RELATED WORK AND BACKGROUND

2.1 Conway’s law

Conway’s law is in literature also known as the mirroring hypothesis [14]. The mirroring hypothesis is a more widely used hypothesis, not only for software development but for all kind of development projects. Cataldo introduced a socio-technical congruence framework, that states that the social and the technical component of a project need to be in line with each other [13]. Cataldo’s socio-technical congruence has a lot in common with Conway’s law and the implications are the same, there needs to be a fit between the technical and the social dimension [16].

Colfer and Baldwin made an overview in which they evaluated over a hundred studies in which the mirroring hypothesis was tested [14]. A fifth of the articles was about research done in open collaborative development projects, like OSS. Within this group of articles about open collaborative development only 39% supported the mirroring hypothesis.

Since the information about OSS projects is openly available, these kind of projects are very attractive for research [22]. A well known research about two OSS projects is about the development of Apache and Mozilla [24]. They checked within these project how they worked and how the work was organised around these projects. These studies gave a lot of insight in how OSS communities work.

It was shown that communication is very important especially in OSS communities [20]. This is required because of the modularisation of the software [10]. The socio-technical patterns have been a topic of research already. Hossain et al [18] looked at betweenness centrality match as is also used by Amrit and van Hillegersberg [8] in combination with Conway’s law.

Amrit and van Hillegersberg [5] developed a tool and method called TESNA for detecting the Socio-Technical Structure Clashes (STSC’s). With this tool software development can be evaluated and mismatches can be found. The tool has been used successfully in two cases with closed source software development [5] [8].

Another research from Amrit and van Hillegersberg [6] explored the effect on the organisational structure of the increase in software coupling. A changing design structure needs a management that copes with such changes and knows how it effects the organisation. They also showed that the health of a project can be measured with movements in the Socio-Tecnical Core-Phirepery [7].

2.2 Text mining and classification

The usefulness of text mining tools in OSS has already been proven. It can be used to classify bugs [25], other software bug reports [27] and development mails [9].

Classifiers that are often used are the Naive Bayes Classifier (the Boolean Multinomial variant proved to be the best in spam filtering [23]), Logistic Regression an Decision trees. Since the Naive Bayes classifier is the one with a high-bias [26], which means it works the best with a small training set, this one will certainly be used. There are in fact a lot of possible implementations of the Naive Bayes Classifier. Five of them were studied by Metsis et al [23].

There will be used in this paper, the Bernoulli classifier, a traditional implementation and one that is developed more.

Basically the Naive Bayes classifiers look at a message as a bag of words. A message that has to be classified is compared to a bag of words of a certain class and compared to an other. A probability will be calculated for both bags and the message will be classified as the class of the bag with the highest probability.

The n-gram model can also be used for classification. The n-gram model predicts the the likelihood based on the next n tokens, which results in a probability given a certain class. The class with the highest probability is the most likely to be the actual class.

Another option for classification is the k-Nearest Neighbour algorithm. This algorithm looks to the k nearest neighbours of an object, a message for instance. It will be classified based on the classification of the k neighbours that are the closest to the object itself. Neighbours are objects that look the same as the original object.

3. METHOD

The first research question will be answered by finding the organisational structure within a community. Apache has been chosen as the OSS project to identify this organisational structure. More particular, the HTTPd project which is part of Apache has been chosen to evaluate. The developers communicate over the mailing-list: dev@httpd.apache.org. First we will describe how the data is collected. Then the usage of the classifiers will be explained.

3.1 Collecting e-mails

All archives of mailing lists of Apache projects are available on the internet in mbox format [1], a file format standard for e-mail messages. E-mails can be downloaded per
month, the e-mails send over the list since 2002 till now, June 2014 were downloaded.

Some pre-processing was needed since one mbox file contained all emails send in one month. Also, for the classifier we only need the body of the e-mails. An other thing we wanted to do was grouping e-mails with the same subject in conversations. Often e-mails are responses on other mails. These responses can be recognized since the subject of these mails begin with Re:, which is short for reply, followed by the original subject. The responses and the original e-mail together are called a conversation.

A program was written in Python to do the separating of the mails, the extraction of the body and the combing of the e-mails in conversations. Some other minor processing was done like storing the e-mail address of all persons who send an e-mail per conversation and removing the lines of the original e-mail(s) in the responses, since those were already stored.

Now the e-mails are ready to be classified.

3.2 The classifiers

The classification of the mails was done by five classifiers. All these classifiers need training data, that means they need a set of e-mails that are already classified. For testing the performance of the classifiers we also need a set of data that has been classified already. For both sets, the training and the testing set, e-mails need to be classified manually. For these sets, 200 conversations were manually classified which means that these conversations were rated based on the body of the e-mails. The conversations about a dependency were separated from the conversations that where not. The first 200 conversations of 2014 were used for this.

This provides a training set as well as a set with e-mails to test the classifiers. For a validated result a method called cross-validation was used. 10-fold cross-validation is commonly used for this and will be used here. The original dataset with 200 conversations is split into ten equal sized subsets. This is done ten times with nine subsets used as training data and one subset for testing the classifier.

A popular toolkit which has implemented a few classifiers is NLTK[3] which is written in Python. But after some tests we found that training of the classifiers took a long time so we decided to use an other tool kit which is written in Java and is named Lingpipe [2].

The five classifiers that were used are: an implementation of the Naive Bayes classifier

```
NaiveBayesClassifier.java
```

the Bernoulli classifier, a variant on the Naive Bayes classifier

```
BernoulliClassifier.java
```

the k-Nearest Neighbour classifier

```
KnnClassifier.java
```

a classifier based on the n-Gram model

```
DynamicLMClassifier.java
```

and an other implementation of the Naive Bayes classifier

```
TradNaiveBayesClassifier.java
```

All these classifiers need training with the training set. All classifiers were used with the default settings. The k-Nearest Neighbour classifier needs a value for $k$. A simple test shows that $k = 1$ gives the best results. Usually $k$ is small so we tested the classifier with $0 < k < 10$ what showed us that $k = 1$ gives the best results. The classifier based on the n-Gram model also needs a variable to be filled in, $n$. A common value is found in the range $0 \leq n \leq 16$ so we tested these values the same way we found the $k$ and found that $n = 7$ gives the best results.

All classifiers needed a tokenizer which divides a sequence of characters, like a sentence, apart into words. The IndoEuropeanTokenizerFactory which is part of the Lingpipe toolkit does this, is ofter used for English text and will be used as tokenizer for all the classifiers since all e-mails on the mailing list are in English.

3.3 Performance measurement

To evaluate the classifier some custom code was written in Java to calculate the parameters that describe the performance of the classifiers. All performance identifiers are based on a four way identification, with true positives: the conversations about a dependency that were tested as one, true negatives: the conversations not about a dependency and the test result says either, false positives: conversations that were classified as about a dependency but are really not and false negatives: the conversations that are about a dependency but are not classified as one.

The parameters precision, recall and the $F_1$-score will be used to compare the classifiers. The results will also be plotted in a Receiver Operating Characteristic (ROC) space.

4. RESULTS

Within a minute the classifiers were trained and tested. The results, the performance identifiers, are represented in graphs. In Figure 2 the accuracy, the precision and the recall of the classifiers are represented. The higher the bar, the better the score. The most accurate classifier is the K-Nearest Neighbour classifier followed closely by the classifier based on the n-gram model. The K-Nearest Neighbour classifier is also the one with the highest precision but the Traditional Naive Bayes scores very high on recall. More than 90% of the e-mails classified as being about a dependency are actually about a dependency.

Figure 3 shows the $F_1$-Score of the classifiers. The $F_1$-Score is calculated with the precision and recall score which are shown in Figure 2. Here the Traditional Naive Bayes scores high but again the k-Nearest Neighbour classifiers scores the highest. This is as expected since the $F_1$-Score is calculated with the precision and recall scores and can be seen as a weighted average of those two.
The last graph, Figure 4, shows how the classifiers score in a ROC space. On the horizontal axis the False Positive Rate is plotted and on the vertical axis the True Positive Rate is plotted. The diagonal dotted line is for reference purposes, anything under this line is worse than a random guess, anything above better. The classifier with a score farthest away from the dotted line is the one with the highest predictive power. The Naive Bayes classifier is, based on the score in the ROC space, worse than a random guess. The differences between the classifiers isn’t very big, but the Traditional Naive Bayes classifier has the highest predictive power followed by the k-Nearest Neighbour.

To give an idea of the amount of conversations about a dependency in the past few years a classifier classified all the conversations that were available for now. In total 14028 conversations were identified since 2002 in the HTTPd developers mailing list. 2841 (20%) conversations were identified as being about a dependency, so 11187 (80%) were not. These results are also listed in Table 1.

4.1 Discussion

A weakness of all the classifiers is the manual selection which is used to get the training and testing data. This can partially be eliminated by writing some classifying rules, but this is very difficult. An other method is the use of one or two persons which also do the manual selection, separately, and compare these results by performing a kappa inter-rater reliabilities tests. This will give a more validated result.

An other problem with this measurement is that no thresholds were changed to optimise the result. Changing the default values of the classifiers will improve the accuracy of the classifiers since it’s not proven that the default value works the best in this case.

5. CONCLUSIONS

With these results, we can conclude that it is not fair to use all communication between the developers as communication relevant for the organisational structure as meant by Conway. The communication has to be about the dependency, otherwise one can not speak about mirroring the technical structure represented by the code dependencies and the organisational structure. Although the best performing classifier doesn’t have a perfect performance, a test with this classifier shows that roughly 20% of all communication is about dependencies. A lot of the conversations on the mailing list were about something else like how functionality must be implemented, votes for a certain implementation and even about a certain convention that was coming up.

The classifier that is the most appropriate in this case is the k-Nearest Neighbour classifier algorithm. This classifier has a the highest F1-Score and has the best score in the ROC space. Generally said, classifiers can be used to identify whether a conversation is about a dependency or not.

6. REFERENCES


Table 1. Conversations classified by n-gram based classifier

<table>
<thead>
<tr>
<th></th>
<th>Total</th>
<th>Dependency</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conversations</td>
<td>14028</td>
<td>2841</td>
<td>11187</td>
</tr>
<tr>
<td>Percentage</td>
<td>100%</td>
<td>20%</td>
<td>80%</td>
</tr>
</tbody>
</table>


