Imperial College London
Departement of Computing

Mobile CrowdSourcing Application and ML Processing for Barclays Bike Hire Scheme

Author : Jerome GUERET - JAG13 - CID: 00880077
Supervisor : Emil LUPU

Submitted in partial fulfillement of the requirements
for the MSc Degree in Advanced Computing of Imperial College London

September 5, 2014

Abstract

Mobile crowdsourcing has been increasingly growing over the last years thanks to the extroadinary expansion of smartphones - and more broadly mobile devices such as tablets - which now possess incredible sensing and computing capabilities. This opens up new perspectives about applications where users - that cannot be trusted individually - easily send back information to a centralised system taking care of the processing. These many pieces of information are useless when taken individually but can reveal really meaningful when gathered and processed adequately. Much of the boring and time-consuming work is finally done by the crowd with a minimum effort given its repartition among many taskers.

This project proposes such an application enabling the identification and the tracking of damaged Barclay's bikes across the city of London in view of a future intervention for mainteance and repairs. The application is built on top of the Hive framework, developed in a previous project, that prevents the developer reinventing the wheel and easily incorporing the new functionalities he needs within an existing structure. Apart from building and deploying the necesary tools that enable the collection of the data, this project focuses on the processing and the issue of outputing useful and relevant information to facilitate the intervention of TFL teams on damaged bikes. It also provides a modelling tool to test both the functionning of the whole solution and the degree of its efficiency, diverse scenarios can be run to build comparisons and estimate how much it would be worth deploying the system for real - i.e distributing the application to users.

Contents

Chapter 1

Introduction

Over the past few years the introduction of smartphones - and more generally mobile devices - has initiated a significant shift in people's way of life and their interaction with the digital world. From the use of a single desktop or laptop, the vast majority of people has gradually learnt how to jungle with multiple devices that can be used for doing, at times, the exact same task, they have been introduced to cloud services that can help synchronise the data between the whole set of devices they possess and facilitate the processes of saving and sharing of important files.
However these mobile devices also come with new functionalities and new perspectives for both users and developers. The main revolution is in the fact that users can personalise they devices by installing a bunch of applications that suit their needs. This flock of applications is all the more diversified that the sensing and computing capabilities of this devices always keep increasing: the number of sensors is every time bigger and the type every time more varied - GPS, several cameras, gyroscope, diverse kinds of accelerometers, light, pressure, altitude, proximity sensors, microphone, compass - the CPU are more and more efficient while consuming less energy and in the same time battery capacity and durability never stop getting better.
This jewel of technology gradually gave new ideas to developers: instead of developing applications only for the user, why not leverage the extraordinary abilities of these devices to help collecting huge amount of data through a growing crowd with absolutely no effort compared to the old fashioned way.

1.1 Project motivation and goals

This project forms part of this perspective: the Barclays Cycle Hire scheme, owned by Transport For London, has a fleet of more than 10000 bikes [18]. Hires can sum up to 47015 a day which is actually the record, achieved during the 2012 Olympic Games taking place in London.
The bikes are distributed across 720 stations which cover most of zone1 and a more imbalanced part of zone 2 if we have a look at the interactive map displayed on the TFL website [21].
In obvious ways, a non-negligible number of bikes gets damaged every day, according to the figures available on the internet, this goes up to 30 bikes a day being repaired in TFL depots [20]. The system put in place to locate the damaged bikes is fairly simple: if a user assesses that the bike he has just used is damaged and in some way dangerous for the safety of future users, he can decide to lock the docking point at the station where he leaves the bike. The docking point being locked prevents other users to hire the bike at the station.
However, here is the main issue with this kind of solution: what prevents the user to lock a bike and make it unavailable for future users for wrong reason: we can easily imagine someone overexaggerating the truth and estimating that the bike is completely out of order when in fact the damage is minor - and the bike is still serviceable - or still worse inexistent - in this case the user might have had malign intents when he decided to lock the docking point or might also be honest.
On the other hand, should a bike be slightly damaged but still useable, it might happen that nobody decides to report it to TFL which has thus no knowledge at all that this bike is to be

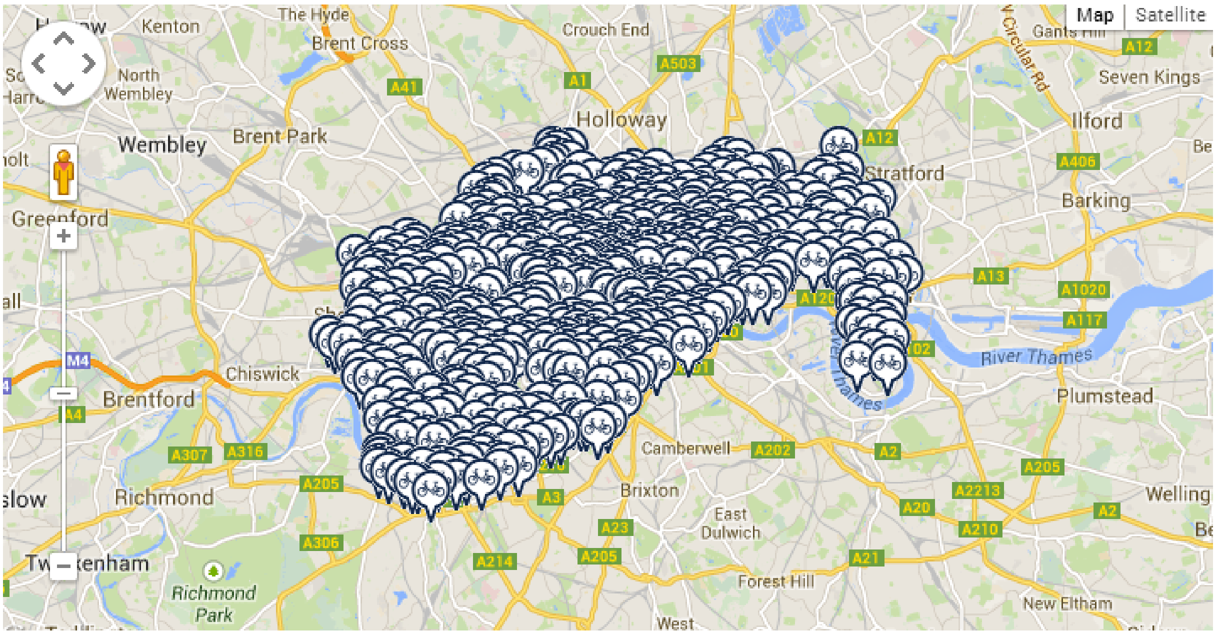


Figure 1.1: Interactive map of stations from TFL [21]

monitored because likely to break completely in the near future.
Therefore when a docking point is locked, TFL teams rely on the opinion of one and only one user, when we all know that taken individually, there is no means to get to know the degree of trustworthiness of each person. On top of that, the maintenance teams that go around the city must come to the docking point to collect the reported bike and bring it back to the warehouse, taking the risk that the bike is not damaged for real and thus wasting time and energy - docking points cannot be unlocked from the distance.

This being said, why not imagine a reporting system leaning on the formidable workforce that represents the crowd and the ease with which it is now possible to access it using the growing capabilities of mobile devices, as stated above.
Such a system would enable users to report slightly damaged to heavily damaged bikes to TFL via their mobile device in a first step. Users participating to the experience would also be asked in a second step to confirm or deny information collected from other users, when the opportunity presents.
A back end server would realise the collection and centralization of all the data and processing via ML algorithms to output the ground truth, i.e. the real state of each reported bike: is it damaged for real? If yes what is the true damage? The registration of users on the server would enable to assign each of them a degree of reliability or performance that would be taken into account when estimating the ground truth. The server would also play the role of displaying all the necessary and useful information required by maintenance teams so that they know where to intervene.

Both TFL and users would gain to use such a scheme: TFL would save precious time not getting around the city to collect a bike that is not damaged and on the other hand would have an overview of bikes to collect in a near future, based on the prediction of its final breakdown. By assigning priorities to the different types of damage and their degree of gravity, they would even be able to determine ahead the optimal journeys to be done so as to pick all bikes requiring the most urgent maintenance.
As for users, they would not anymore restrain themselves when thinking of warning TFL of a damaged bike - in the case of honest users - as the bike would still be available and serviceable for others. Maybe some of them would express an excess of zeal and report non damaged bikes with benevolent intents, but that would not be an issue considering the fact that the system would be

able to set aside these mistakenly reported bikes.

1.2 Contributions

The first contribution of the project is the literature review presented in the next chapter and which results from the background research that was initially done. It deals with the mobile crowdsourcing field and outlines the tool that has served as a starting brick in the elaboration of the final solution. It also outlines useful ML algorithms related to the goals of the project: assessing ground truth from many a-priori untrustworthy annotators.

The main contribution of this project is the final system. It boils down to two distinct parts:

- On the one hand, there is the mobile application. This reuses the work done in the Hive framework: the application used to list the tasks created on the dedicated Google App Engine application as well as the plugin taking a file saved by another plugin and uploading it on the server [12]. The work done consists in the creation of a new plugin that implements all the functionalities needed for the goals of the project on the client side: it enables users to easily locate themselves into London, retrieve interesting data about surrounding stations, select the bike station where they are standing and fill the different types of forms (either for reporting a bike or confirming or denying information collected via other users).
  The plugin has been coded in accordance with the requirements imposed by the Hive framework so that it becomes a new brick that extends the set of functionality proposed by the whole framework, as it was conveyed in the spirit of its creation: "an extensible framework.
- On the other hand, the back end server application is also built next to the initial implementation for the Hive framework. The original work implements the user interface that enables the developer to create new tasks and designs the distinct stages into each of them. In addition to that, there are the methods used to communicate with the part of the framework located on the mobile devices (forward the tasks and stages to the user, store the data uploaded on the server).
  What has been done in the project is reorganising and making sense of the crude data retrieved from users and stored into the blobstore in order to process it thanks to the ML algorithm implementation. In addition a web user interface has been devised to enable the display of the data, and the output results once processed by the algorithm.
  Finally, as we needed to test the good functioning of the system when subjected to the input of huge of data, a tool for modelling the dynamics of the reality has been added to the web interface. It enables the simulation of diverse scenarios and an easy way to change the parameters of the model.

The last but not the least contribution of this project is this report, it aims at explaining and outlining all the work that has been undertaken. It describes the main ideas that helped designing the system and conveys the subtle details and trickiest points that occurred and raised issues in the implementation step.

1.3 Outline of the report

The rest of the report is organized as follows:

Chapter 2 reflects the work that has been done before starting to design the solution, as a background search to get to know existing tools that were of interest for the project's purposes. It gives a general overview of what mobile crowdsourcing is, with the main problematics associated and also focuses on more technical aspects such as the variety of ML algorithms employed and the reuse of an existing framework built for such applications - Hive framework.

Chapter 3 aims to outline the main ideas, concepts and principles that have been followed all the project long and which altogether helped to design the final product.

Chapter 4 gives details about the implementation: the main ways used to realise the goals and the technical subtleties that had to be employed to overcome some issues encountered while implementing.

Chapter 5 evaluates the functioning of the implementation as well as how attractive could such a system be for TFL. It plays with parameters and tests the limit cases that could theoretically happen should the solution be deployed on a large scale.

Chapter 6 tackles the evaluation of the product as it is, presenting the advantages and downsides as well as the missing features that could be of interest to the system and would worth being considered for future work.

Lastly, chapter 7 concludes on the work that has been done and gives some suggestions for future work.

Chapter 2

Literature review and background

Crowdsourcing has become an increasing trend to collect huge amount of data and get it labelled at the lowest effort possible. The workers are the thousands or even millions of easily reachable people for which mini task are trivial while much more complicated to assign to a computer. Another important aspect of crowdsourcing is the time it takes to collect huge amount of data, it is reduced to hours or even minutes when it took much longer to get it labelled by more classic ways in the past.
Such an increase in the field is recently due to the fast growing market of mobile devices such as smartphones with powerful capabilities and extraordinary sensing abilities.

However, as this field has been dramatically growing and many applications have been developed, the issues and limitations have started to appear and be clearer. One of the most recurrent and important issues is about the quality of the labels we get from the many annotators. These anonymous workers are completely unknown to us, unlike employees in a company would be to their boss; and still we would like to trust the work of annotation we require from them. How can we be sure that the task accomplished by someone we do not know a priori is to be considered as the right answer we are looking for? The answer is we cannot. Should we have to do the task by ourselves, maybe we would have provided a complete different answer. Even if we ask many different and supposedly independent annotators to do the task, how are we going to distinguish who is giving back the right answer from who is wrong? One might be tempted to state that the truth lies where the majority puts it, but we could envisage cases when the majority is wrong because of some prejudice, or because the task is difficult, requires some expert knowledge to answer it properly and there are only a few experts and a lot of novices in the set of annotators. This search of measuring the reliability of each annotator and be able to find the ground truth among noisy data returned by an unknown set of annotators is a major issue raised in the field.
Another important problem is how to find the most optimal set of annotators to complete a task. A very simple task will require only a relatively small number of annotators who will most probably answer back in the same way confirming the problem was easy while it might need a larger set of annotators to answer rightfully to a more complicated issue. How do we adapt the number of annotators to the difficulty of the task? And if we have enough of annotators, which ones do we select? Can we find a way to determine some annotators more appropriate to deal with that specific task?

Among the goals of this project, one is to extend the server side of the Hive framework (which is outlined in the following review), with Machine Learning algorithms to get useful, intelligent and reliable data out of the many small amount of data collected from the crowd, where each person is unknown and for that reason supposed untrustworthy. The ultimate goal would be to tend towards a scheme which is described in [1] : an iterative way where data collected from workers is quality assessed before being classified and in the case where the classification process is not confident enough in its answer, some more labels are required.

2.1 General knowledge on crowdsourcing

The involvement of humans in crowdsourcing raises some important trust issues. Each worker cannot be trusted individually. No assumption can be made about a worker who is a complete unknown being with unknown motivation and degree of knowledge.

Motivation is a key aspect in crowdsourcing [10], the challenge is to find a way to make potential workers participate in the tasks we require them to do. At first, participation is highly facilitated by means provided by technology such as the internet; people can be tasked without requiring them to move to a specific place. Thanks to computers and now the huge rise in the use of mobile devices, they can participate whenever and wherever they want, which is a great advantage. Even so, to participate in such tasks instead of doing something else, these potential workers need a strong enough reason to motivate them.

The financial reward is the easiest way to get people to participate as we can notice with the popularity of the Amazon Mechanical Turk which is merely an online market of small tasks to be completed. In the same spirit, participating persons might also be rewarded with gift certificates or virtual money that can be used to buy specific rewards on dedicated stores. A more economical way to get people to participate is also to inform them of the possibility of winning a gift through a lottery where every participant (generally to a survey) is present, if the reward is attractive enough it might suffice to get people to complete the task, even if they know there is a high probability to get nothing in return in the end.

Altruism is another form of motivation, even if it may sound odd at first, if the community think that they can help considerably by performing the task and that it requires a minimum effort. A recent example of this is the incredible number of internet users who have been mobilized to search through thousands of satellite images to help and locate some evidence of where the wreckage of the Malaysia Airlines plane could be in the ocean. A last important feature that can motivate people is the enjoyment they can benefit from completing the task, for example if the task takes the form of a game as it is the case with the ESP Game [13] (game which consists in labelling images) and gives some entertaining to the participant.

Reputation and implicit work are another two aspects that can arouse motivation among workers. Reputation is the hope of getting any public recognition for the completion of the work while implicit work is the way to introduce the task as the part of another activity undertaken by any user. A well-known example of the latest is the most commonly used ReCAPTCHA system [14] for the registering of a new user on a website. The primary purpose is to prevent automatic registration on the website (made by a computer) while at the same time it helps identifying words for which OCR has failed because the scan was of an old book or newspaper.

Beyond the aspect of motivation is the one of quality control. Many cases are possible and the main issue is that we do not have any idea which one is correct: the worker may be cheating to complete the task faster and be paid more at the end of the day, but he might also be honest and simply have misunderstood the guidelines given. This underlines the fact that the instructions given to the annotators have also a key importance [2]. The words used to describe the task and what is expected of the worker can be decisive on the quality of his answer. Finally the worker may also be a complete novice about the field it is being tasked with, which raises the question of why he has been selected to complete it.

There are many theoretical ways to assess the quality of the control, some basic ones and other more advanced.

Among the simple ways to check the truthfulness of an annotator's answer is the processes of output agreement and input agreement.

Output agreement is mostly known for being used in the ESP Game [13], it consists in making two or more workers complete the same task and compare their answers. If they agree at some point on the answer we can consider the label as true. It is quite efficient on small easy tasks as labelling an image which is straightforward to describe as a living being (animal, human) or a specific feature widely known in the community (as would be a building for example) but it has some serious limitations with more complicated task.

Conversely, input agreement let the contributors interact on the input they are given and let them decide if they were given the same input. If they agree it is the same input, it is taken as being correct.

In a more advanced way, the label given by a first group of annotators can be checked by a second group, it is the basic idea on which relies the find-fix-verify pattern outlined by Soylent on AMT [15]. Still further the workers reviewing the work that has been done can be experts. There is then the need to know who is an expert in the community.

A more iterative approach is to this is making a third worker decide which answer of two previous workers is the right one (if they are different).

Another idea is to put in place a voting scheme where every participant vote for the idea he thinks is correct among the set of all the labels given back previously (there is a redundancy). This way of doing enables to identify the poor workers and get them out of the system so that the final quality is not affected by their work.

Ground truth seeding is another approach that consists in giving tasks for some of which we know the ground truth or at least we have an answer from a trusted worker. This way it is easy to filter out the poor annotators and keep only the annotations made by the best workers.

Some more original approaches focus on designing tasks in a defensive manner, for example, so that the worker is not encouraged to cheat or provide a poor label because he is not interested in the task.

2.2 Probabilistic models

Majority voting is often referred to as a fast and simple way to determine a close value to the ground truth that we are looking for [4]. It consists in considering that the true label is the one provided by the majority, or, if binary or discrete values classification is used, the closest label to the average of all the collected values.

However such a method relies on the strong hypothesis that the performance of the annotators is distributed in an equal manner which is far from being the case: there may be a small proportion of experts among a very large number of novices in our pool of workers, hence in all likelihood the process of majority voting will favour the novices over the experts and produce a wrong result, far from the gold standard. A simple solution to the issue would be to assign a weight to each annotator based on its reliability, but now is raised the problem of how determine the performance of each worker when we do not know the true label and cannot compare it with the answer given back.

In the paper [4] is proposed a probabilistic approach to adapt supervised machine learning algorithms when the annotators provide noisy labels and a fair amount of disagreement due to the inherent bias of human nature and if the ground truth is possibly not reached in the answers provided. Its purpose is to estimate the extent of reliability of each annotator and find the closest evaluation of the absolute gold-standard.

Given the features and the corresponding labels, it uses a maximum-likelihood estimator to learn a classifier which takes into account the accuracy of each annotator considered and return the actual true label. Specificity and sensitivity with respect to the ground truth (normally unknown) are used to rate the accuracy and thus reliability of annotators, a weight is then assigned to them, the higher the weight the better the annotator is in terms of expertise. The method can also include some priors about the workers via the specificity and sensitivity, allowing putting some trust in a specific annotator or conversely some mistrust, and a Maximum A Posteriori estimation is then considered. The way it is done is by applying an Expectation Maximization algorithm which iteratively refines its answer based on the accuracy of each annotator. The method is derived mathematically using logistic regression, but is also valid for categorical, ordinal and regression problems. An interesting feature of the method is that it can obtain the ground truth for data which has no features, just labels. In this case, no classifier can be learnt but the prevalence of the positive class (in the case of a binary classification), the algorithm is a little simplified.

The overall method assumes that the annotators answer in an independent way from each another, meaning that the errors they may provide are independent. However in practice the hypothesis does not hold, on some cases, workers might answer in a similar way (whether the label turns out to be right or wrong). It may be due to multiple reasons such as the answer is easy and commonly known by the community.
Another important assumption made in this paper is that the annotator performance is independent on the case he is dealing with. This also does not hold in practice, the worker might be more familiar and expert in a specific instance and thus provide a better answer than he is used to doing overall.

In the paper [8], the authors outline the wisdom of crowd through multidimensional models. Workers are modelled as multidimensional objects, where the dimensions may represent their competence, accuracy, expertise and bias. The model which is proposed enables to classify the workers in distinct groups of annotators inside which the workers have the same way of thinking and therefore tackling and completing a task. The reason for the multidimensionality is that a worker may have varying expertise regarding the tasks he is asked to perform and thus his reliability may be better on a specific subset of tasks compared with his overall performance. Furthermore, from one worker to another, there may be a different bias in the sense that one does not react in the same way when we are not sure of an answer, some might risk an unlikely answer while others would prefer to reason properly before answering. This results in different types of bias among the pool of annotators.
The method also tends to propose a way to reduce the number of annotators as far as the lowest number needed to get an acceptable and supposedly valid label.
A generative Bayesian model is put in place for the annotation process with an inference algorithm to assess the proper characteristics of the data and the annotators. Latent variables represent the true characteristics of the images (binary classification of images is the main subject of interest in the paper) and the different dimensions of the annotators while the only observed variables are the labels given by each annotator for a set of images.
Although the authors claim that their model works better and achieve finer performance for their specific need (binary classification of images) than the ones present in the literature at the time [4] [5] [11] and from which they have based their work, they note that other approaches might be more suited to non-binary annotations. The multidimensionality of the workers enables finer results and provides the ground truth label of an image by combining the labels given for that same image of several distinct groups of annotators with different skills. Another interesting feature of the model is that images are also considered as multidimensional entities and that the resulting vectors produced by the method enables the training of classifiers which rely on the possible values of each dimension.

Whitehill et al. describe a probabilistic model for inferring image label [11] which demonstrate that the commonly used majority vote scheme can be outperformed. The model, more commonly known as GLAD, is robust to noisy labellers as for the previous ones, but is also anti adversarial-labellers. It provides the expertise of the workers, the difficulty of the images and infers a label for the image which is hopefully close to the true label. In the model, only the labels are the observed variables while image difficulties, true labels and labeller accuracies are described as latent hidden variables.
As for the method used by Raykar et al. [4], maximum likelihood estimation is done by computing an EM algorithm which yields the parameters of interest. Performance in terms of proportion of correctly labelled images is compared to both Dawid & Skene model [16] and the majority vote scheme and reveals the small rate of errors in the classification made by the GLAD model.
This model is interesting in the sense that it is operating in a completely unsupervised manner. It provides the true label, the image difficulty and the worker accuracy and reliability when the only input observed are the collected labels issued by the annotators. The model can be used further to combine labels from an automatic labeller such as a classifier learnt by a machine learning strategy and human labellers.

Welinder et al. describe a similar way of approaching in his paper [8]. The general model outlined comprises of an online implementation of an EM algorithm. The number of labels asked per image varies for each image, and adjusts so that extra labels are no longer required once an acceptable certainty of the true label value has been established. It has the particularly to favour the participation of annotators classified as expert and avoid the participation of novice annotators.

The advantage of probabilistic models is that they are well adequate for an implementation of active methods which consists in determining if and what objects are to be re-labelled or be given to more annotators for an answer so that the certainty of the deduced label in the end is more correct and the probability of making an error reduced. This helps to produce a better approximate of a labeller expertise which is important for determining the accuracy and reliability of the labelling tasks he will do in the future.

There exists an approach [6] which focuses on examining the way annotators perform the tasks they are given, considering that the way with which they work may be of great help when it comes to determine the skills of each worker, its accuracy, reliability and seriousness to do the job.

The algorithm selected, at least for the first implementations, is the one of Dawid & Skene [16]. The main steps, features and notations are recalled below. A description and comments as well as an improvement of the algorithm are present in the paper of Liu [17].

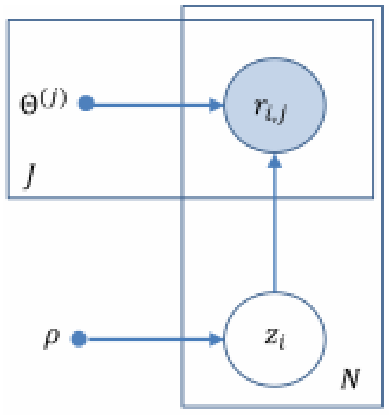


Figure 2.1: Graphical model representation of Dawid & Skene model [17]

We suppose that we have N items, J annotators of different but unknown expertise and that we have K distinct classes or labels possible for annotators to choose.
The label assigned by the jth annotator to the ith item is noted: $r_{i,j}$ and belongs to the set $\{1, 2, .., K\}$ if the annotator labelled the item. In the case when no label is assigned $r_{i,j} = 0$.
These ratings are observed and thus in input of the algorithm.

The notation t_i is used for designing the true label of item ith. The corresponding estimated label is noted $\hat{t}_i$. As shown in Figure 2.1 the model supposes that the true rating of each item is sampled from a multinomial distribution parameterized by ρ.
The degree of competency of the jth annotator is modelled with a confusion matrix of dimension KxK : $\Theta^{(j)}_{k,t}$ where its (k, t) element models the probability that this annotator (jth) will give the label t when the true label is $t_i = k$.
In all the following description of the algorithm the symbol hat is used to precise that we speak of the estimated value of the corresponding parameter.

The purpose of the inference is to get the value of the estimate parameters $\hat{\Theta}$ and $\hat{\rho}$ and the N true labels t.

The initialisation uses the following formulas:

$$\begin{aligned}\hat{\rho}_k &= 1/K \\ \hat{\Theta}^{(j)}_{k,t} &= \begin{cases}\lambda/(\lambda+K) \text{ if } k=t \\ 1/(\lambda+K) \text{ otherwise}\end{cases}\end{aligned}$$

And then we iterate until convergence the two following steps (succession of E-step and M-step):

E-step:

$$Z_{i,k} = \frac{\rho_k \prod_{j=1}^{J} f(r_{i,j} \neq 0)\Theta^{(j)}_{k,r_{i,j}}}{\sum_{k=1}^{K} \rho_k \prod_{j=1}^{J} f(r_{i,j} \neq 0)\Theta^{(j)}_{k,r_{i,j}}}$$

where

$$f(r_{i,j} \neq 0) = \begin{cases}1 \text{ if } r_{i,j} \neq 0 \\ 0 \text{ otherwise}\end{cases}$$

M-step:

$$\begin{cases}\hat{\Theta}^{(j)}_{k,t} = \dfrac{\sum_{i=1}^{N} Z_{i,k} g(r_{i,j}=t)}{\sum_{t=1}^{K}\sum_{i=1}^{N} Z_{i,k} g(r_{i,j}=t)} \\ \\ \hat{\rho}_k = \dfrac{\sum_{i=1}^{N} Z_{i,k}}{\sum_{t=1}^{K}\sum_{i=1}^{N} Z_{i,k}}\end{cases}$$

where

$$g(r_{i,j} = t) = \begin{cases}1 \text{ if } r_{i,j} = t \\ 0 \text{ otherwise}\end{cases}$$

To compute in the end:

$$\hat{t}_i = argmax_{k=1,2,\ldots,K}\ Z_{i,k}$$

This gives us the predicted label for each item.

The paper commenting on the Dawid & Skene algorithm [17] emphasizes that their model has $(J * K + 1) * (K - 1)$ free parameters and that this gives the model a high model capacity but on the other hand this can also lead to overfitting when there is not enough data. However this should not be our case with the Mobile Crowdsourcing application we want to build given the high number of bikes and users that we should have to process in practice.

The value of λ is closely related to the degree of competency of the users. It is to be superior to 1 and close to 1 when annotators do label correctly and increases as taskers start improving their performance [17].

2.3 Mobile crowdsourcing

Performing crowdsourcing on mobile devices enables new ideas to be brought to daylight; thanks to the incredible abilities of smartphones in terms of sensing, a distinction has to be made between opportunistic and participatory sensing [2].
Opportunistic sensing is a way of collecting data without the direct intervention of the user, except for his initial authorization of the process being carried out. On the other hand participatory sensing involves the user and requires his human capabilities of mind to carry out the task given. Given that many applications only relying on opportunistic sensing are performance wasting for the device, it appears that most of the applications have to be a balanced hybrid of the two approaches.

2.4 Technology used

The technology used comprises of the Hive framework which has been developed recently and Android based devices.

Web services run on the server which collect the data from the workers and provide the UI to the developer. The web services are based on RESTful web services which are centred on the representation and exchange of the state of the data. They only allow 4 types of operations which are the GET, PUT, DELETE and POST operations of the HTTP standard. Since it seems that the RESTful approach is more suitable for a point to point interaction that other technologies that were considered, they have been selected.

2.4.1 Hive framework

Hive is a framework developed last year by D. Valentinov [12]. This framework was created under the need that mobile crowdsourcing applications have shared functionalities and that it might be frustrating for developers to start from scratch for each new application. In the analysis of this student, no other framework existing was providing the real needs that such a tool should be solving. The main advantage of his work is that it enables third party developers to add their own brick to the existing infrastructure of the framework and share it to the remaining community so that one does not need to rewrite the code of a commonly used functionality if it already exist.

The tool is flexible enough so that a whole range of applications can be built on it. It consists of a (in appearance) basic and blank android application which is distributed to the users through the application store of Google (Google Play).
On the other hand, the developer hosts the server code of the framework on one of his own servers. Originally it was designed to run on Google App Engine. The User Interface that is accessible via the webpage displayed by the server is used by the developer to build the tasks he wants the workers to fulfil. It is relatively well guided and enables the developer to specify every stage that may occur during a task (for example 1-take a picture 2-classify a text).

The workers which are registered to the server of the developer (usually via their GMAIL account) can see on their device the list of every task which is available on the server, ready to be completed. They can either subscribe to a task or choose not to participate. The android application is built such that it knows how to communicate with the server, check for notifications and transfer data collected from the device back to the server. What a developer has left to do is developing his own plugins that will be the stages of a task. 'Take a picture' or 'classify a text' are plugins that D. Valentinov has developed to illustrate the way the framework is built and can be enriched with a lot of functionalities from many third party developers.

The developer can in theory know the number of participants to a certain task via the UI. He can also access the produced data.

One of the key advantages of the framework is the way it tackles some serious challenges that arise often when dealing with Android development on mobile devices. A primary challenge is to take into account the fact that the mobile might be in a place where data connection is seriously affected or completely unavailable. The different exchanges that needed to be done must be restarted either by the application or by the user who has been informed of the failure.

Privacy is also a key aspect when developing such applications since the data that can be of interest for the developer might contain private information of the user. The framework deals perfectly fine with this challenge since it asks the adequate permission to the user and informs him about the current operations and their consequences whenever such an access to private data is required. The design process of the framework has also taken into consideration that performances of the mobile device while being used by the application must not be compromised in any case, or it would lead to a negative experience for the user who might be less willing to use again the application. Operations and processes need to run softly and respect the parallel tasks that the user might be undertaking.

Chapter 3

Design

This chapter describes the main ideas that drove to the design of the final product as it exists now. It emphasizes the main principles and concepts on which the project has been built on. As a first step, it describes the ideas that led to the development of the mobile application: the different situtations in which the user might led to use the application, the necessary functionalities that should have the application, the technical constraints imposed by the fact we build it using the Hive framework - extending its functionalities - and some concepts about how to make the user interface more intuitive and easy to use.
Secondly, we focus on the server design. We analyse what has been implemented in the Hive framework and what is missing for the need of this project: we especially focus on how it would be convenient to process and store the collected data about bikes.
In a short part, we expand on the fact that during this project we had the privilege to have access to real data, released by Transport For London about his Barclays Cycle Hire Scheme.
Finally, we discuss the idea of building a modelling tool for simulating the dynamics we might expect should we deploy the system for real. The model in question is described in details, explaining what real data and figures we consider and how we model unknown parameters.

3.1 General purpose of the solution

The main idea on which the project is based is to use the crowd to collect information about potentially damaged Barclays bikes, and process this collected data so as to identify bikes that are really damaged (and what is the nature of the damage) from the ones that are mistakenly reported as damaged.
The processing of the collected data is done via the ML algorithm outlined in the background chapter.

3.2 Client side

3.2.1 Cases of use

On the client side, that is on the mobile devices used by the users to complete the task: either a smart-phone or a tablet, what is needed is an application that can enable them to report a damaged bike in the most intuitive and simplest way possible.

For this purpose the interface must be easy to apprehend and the navigation between the different windows have to be well indicated so that the user can know or find easily how to execute the next step(s).

There are two main different occasions when the user might have to report damages on bikes.

The first one is the most obvious one: after the user uses a bike, he might notice a slight problem with one of the elements of the bike, in this case he simply reports the bike once arrived at the station: he fills the form asking for the bike's ID and the damage and sends the output.
As the user might be using the Barclays bikes with other people (family or friends), the interface needs to implement the ability for the user to report several bikes in the case when several of the bikes used by the group of people might be faulty.

The second opportunity occurs when a user just reported a bike at a station: this means he has just completed a journey with the bike and assessed that there is damage on it, upsetting enough to report the bike. This tells us that this user is a reporting user. The assumption is made that this person takes the time to report a bike when he rides one which is faulty. Among all the people using Barclays bikes such a person is relatively rare. The idea is then to ask this user to do some extra work for us: if the station he arrived at contains bikes that have been reported by other users previously, he is asked to check for the presence of the bikes at the station and confirm or not the fact that these bikes are damaged. Of course the user is free not to complete this extra work as he might not have the time or the envy to do it. If the nature of the damage of the bikes is known with a good certainty when asking the user, it can save him some time to indicate the direction to have a look. However if it happens that the predicted nature of damage is wrong, it can also have the reverse effect and lead the user in a false direction. Finally the user has also the choice to answer that the bike is not damaged (according to him the bike will have been mistakenly reported as damaged) or to answer that he does not know meaning that he is not sure of what to answer and prefers not to give back a wrong indication (from the server side though, this situation does not bring more information than a simple "I don't have the time to complete this extra work).

For maximising the chances that the user actually completes this extra work, the interface has to be well designed. The user receives a push notification from the server indicating him the bikes to check at the station.

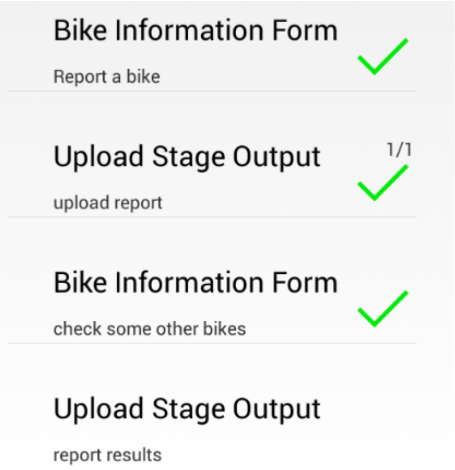
3.2.2 Building on top of the Hive Framework

The hive framework has originally been designed to be the foundation of such mobile crowd-sourcing applications and prevent the developer to re-implement all from scratch [12].

On the client side it consists of an application already developed which registers the user on the server via one of its GMAIL account and lists all the tasks that are available for completion. One task is composed of several stages. These stages are usually executed via plug-ins. There are several plug-ins already existing (sensor capture, image capture, upload a file from the phone to the server, ..).

In the case of this project one extra plug-in is needed: the one which enables a user to report a bike. The other plug-in which was already available with the Hive framework and which will be used is the "Upload the output" plug-in: it takes the written files of a designated stage (plug-in) and upload it to the server.
The plug-in in charge of collecting the report from the user (plug-in called "MCSBikeInformationForm") has to issue a file containing all the interesting data needed for the processing on the server and this file will be saved, at first, on the hard drive of the device. The save of the file is only possible if the user allows the application to do so through the notification he receives (this is a functionality implemented in the framework: each time a plug-in has to access a file or a resource of the phone or save one on the hard drive, it asks the permission to the user via a notification).

Therefore the task required for the project will be divided into four stages: the first stage uses the new plug-in to enable a tasker to report damaged bikes after a journey and once arrived at the station. After this stage is completed, the data has to be sent as soon as possible: this is the goal of the second stage which uses the "upload the output" plug-in. The data is immediately analysed and saved on the server storage space - called blobstore for Google App Engine - and in the case when the station at which the user is standing already contains previously reported bikes, the user receives back from the server a push notification with the IDs of the concerned bikes: the action of pressing on the notification opens the first plug-in again but on a different activity dedicated to check easily and intuitively the state of the bikes indicated (third stage). The fourth and last stage consists of sending the second report with the answers of the third stage.



Bike Information Form ✓
Report a bike

Upload Stage Output 1/1 ✓
upload report

Bike Information Form ✓
check some other bikes

Upload Stage Output
report results

Figure 3.1: Four stages of the mobile application

The advantage of splitting the task into four stages and using the "upload" plug-in is that if the user were to lose his data connection at any point, he would still be able to save the form for a damaged bike onto his device and try to send it later. It also gives the impression to the user to be "in control" since, due to the architecture of the Hive framework, writing data from the new plug-in onto the phone and accessing that data to send it to the server requires the user's permission.

3.2.3 MCSBikeInformationForm plug-in

This new plug-in needs to facilitate the user's work who wants to report a bike which is, according to him, faulty. What is important for the posterior processing is to get the location of the bike, i.e. the station where the user left the bike after reporting it. For this purpose an access to the user's location is required. Knowing his location, the nearest stations of Barclays bikes can be indicated to the user. He then selects the station where he is standing by and completes the mini-form that asks for the bike's ID and damage.

In order to facilitate the selection of the station, a Google Map view is inserted in the main activity of the application and the stations are pinpointed on the map so that the user can find the station he is standing by more easily.

However we need to take into account that geolocalisation accuracy on mobile devices can vary a lot depending on whether the device is connected to a Wi-Fi spot or may only have access to a 3G - or even worse - connection, and also whether the settings of the phone concerning the geolocalisation are set to enable a very precise localisation or a poor one - given that providing a more precise localisation often consumes more battery and that battery use is a recurrent problem on mobile devices, we cannot ignore the fact that some users might set the parameters of localisation as low as possible to save as much battery as it can.

Consequently, on the one hand we want to avoid sending the location of all the stations of London to the user because of the time it would take and the consumption of data connection for the user and on the other hand we might need to vary the number of stations displayed around the user if the localisation is poor.

The simplest way to prevent the user from not finding the station he is standing by is then to authorise him to refresh his location on the Google Map view and also allow him changing the radius of the circle centred on his location and into which are displayed all the bike stations.

The new plug-in must also ensure that the minimum information required is filled by the user: at least one bike with ID and damage.

3.3 Server side

Google App Engine (GAE) [25] services have been used on the server side. It enables the free hosting - even though free often rhythms with quotas and frustrating limitations - of Java Web Applications on Google servers and makes it accessible very easily for everyone wishing to deploy projects of reasonable size.

As the storage via SQL requests is not free on GAE, we use instead the NoSQL service provided. Objectify [24] is the tool used to save the instances of Java objects into the Datastore of the application.

3.3.1 Building on top of the Hive Framework

What is already in place on the server side in the existing Hive framework is a light web interface to create new tasks comprising multiple stages. This is the only functional web page that is accessible for the developer.
On the other hand, what is hidden is the whole structure that stores the files sent by users on their mobile devices (via the "upload plugin). These files are stored into the blobstore of the application.

In the Datastore of the application is saved the details of the tasks created (their different stages, their IDs) and the details of the users that have registered with the server. Users register via their GMAIL account, and what serves as their ID is the part before the "@" (ID: jerome for the address jerome@gmail.com). The tasks a user is participating in are also listed in the Datastore.

In order to achieve the goals of this project, there needs to be an implementation on the server side that centralizes all the data sent by taskers and processes it. Processing implies extracting the data and presenting it in a form that is easy to manipulate from the files saved by the new plug-in on the clients' devices; storing this data and running the algorithm seen in the literature review to identify which bikes are damaged and the nature of damage, and which bikes are not damaged (mistakenly reported).

3.3.2 Keep track of reported bikes and reporting users

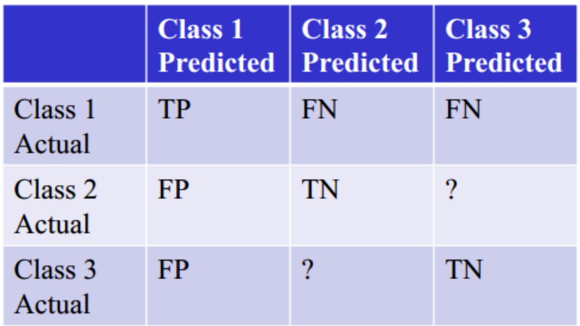
The first task that must complete the server is reorganising the data as it arrives from the many users reporting bikes.

A reorganising step is needed because the incoming files contain the report from one user for one or several bikes and on the server side the algorithm used to label the estimated true damages of the bikes takes as inputs the bikes.
Therefore, for each bike that has been reported at least once, there exists an instance of the class representing the bikes containing a history of its reports. Each report contains the timestamp of

the report, the nature of damage, the user reporting it and the station where the bike was left.

In parallel we also need to keep track of the different users: as the algorithm is going to run and label the bikes, it is also going to output who are the users that deserve to be trusted and who are the ones doing mistakes more or less frequently and thus less reliable. A ”performance field can then be assigned to each user in order to assess the degree of reliability of their answers. There are several ways to make the performance of a user evolve: the simplest one consists in calculating the proportion of right answers the user has given out of all the answers he provided and calculate the performance to assign from a given function - this function can be a straight line or a more elaborated one such as an exponential function if we wish to increase by a little more the influence of reliable users compared to untrustworthy ones.

A more complicated method is to calculate at the end of each run of the algorithm the recall, precision and F1-measure of each user from his confusion matrix (the confusion matrix is an output of the algorithm as seen in the background chapter [17]) and base the performance on the overall F1-measure of the user.



	Class 1 Predicted	Class 2 Predicted	Class 3 Predicted
Class 1 Actual	TP	FN	FN
Class 2 Actual	FP	TN	?
Class 3 Actual	FP	?	TN

Figure 3.2: Confusion matrix and notations used [23]

Given the notations in the previous figure (3.2) which respectively stands for True Positive (TP), True Negative (TN), False Negative (FN) and False Positive (FP), the formulas for these classification measures are the following [23] :

$$\begin{cases} Recall & = \dfrac{TP}{TN + TP} \\ \\ Precision & = \dfrac{TP}{TP + FP} \\ \\ F1 - measure & = 2\dfrac{Precision * Recall}{Precision + Recall} \end{cases}$$

There also must exist a way to save the location of the currently reported bikes (reported as damaged but not repaired yet) so that when the server processes the data from a report file, it can send back the IDs of the bikes which are at the same station where the user just sent his report and save this new bike at the current station - which implies removing it from the previous location if it was already being tracked.

Different counters can be set for each bike as the number of times that the bike has been reported, the duration it was used from the first time it was reported to the moment it was classified with

good certainty as damaged and repaired, the number of times it has been repaired or identified as non-damaged though it had been reported as faulty.
These counters can then be used to make some statistics for TFL about the bikes.

3.3.3 Datasets

For practical reasons, it is easier to introduce the use of multiple datasets in the web interface: each dataset has its own bikes and users. This way it seems more convenient to test particular situations and behaviours of the algorithm and the whole solution in itself with fresh data without having to delete the existing inputs of data - that may take time to collect or enter manually.

3.3.4 About the labelling

Not all the bikes need to be taken into account in the algorithm: we can set up a minimum number of reports for a bike before it is processed by the algorithm. Indeed if the number of reports is too low for a bike (say 1 to 3 reports), even if the user has been reliable until now, he can still be mistaking and ”doing some excess of zeal” by reporting a bike which is not damaged. The spirit is to wait for a couple of other taskers to give their opinion about this bike before predicting anything about its state.

Once a bike is labelled by the algorithm and the degree of certainty about the predicted label is high enough, the bike needs to be taken out of the system that keeps track of the reported and not yet repaired bikes (which are awaiting more labels from the users before being processed by the algorithm).
Given the degree of certainty of the labelling, a bike can be classified into different categories. If the bike is classified into the category which gathers the most certain predictions - the one with the highest certainty, then it means the prediction is reliable and that we do not need to waste more time bothering more users about this bike: we already have enough opinions on its state to predict a good label.
The degree of certainty of a prediction is derived from the Z matrix associated: it is the probability corresponding to the predicted label for the given bike. The higher this number, the more certain the prediction.

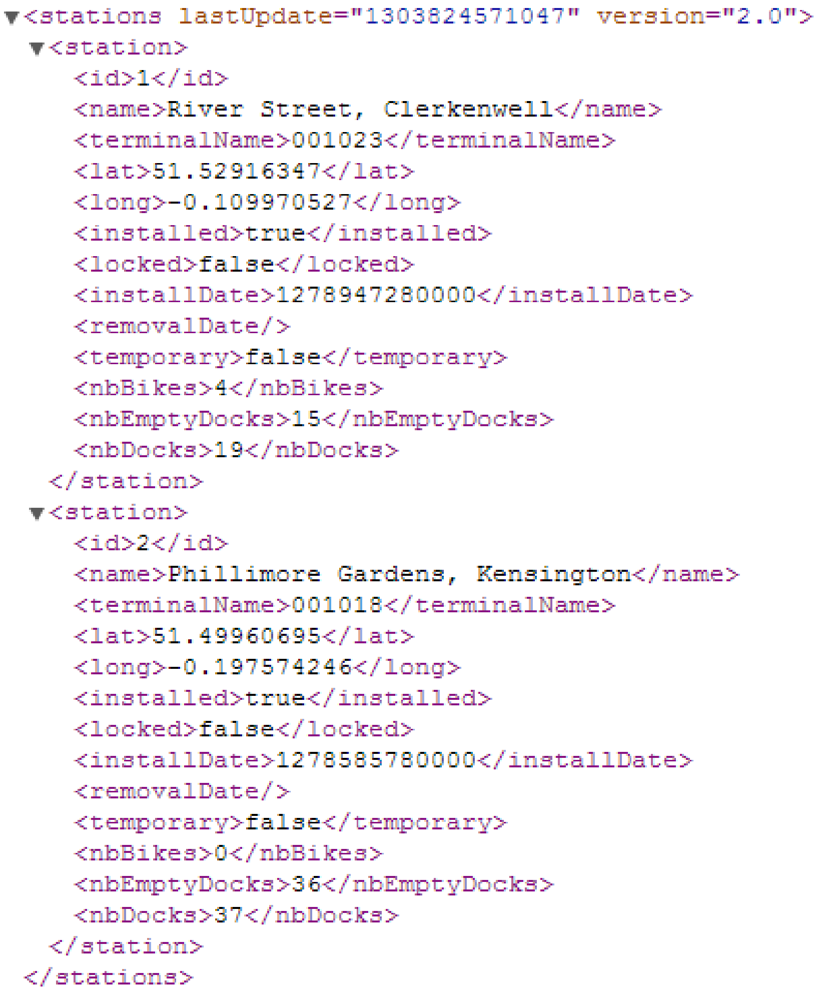
3.4 Use of TFL real data

For this project some real data is used both for the client application and the modelling of the behaviour of the users. There exists two ”feeds” released publicly by TFL.

3.4.1 Stations status feed

This feed is not really a live feed in the sense that it is a XML file (Figure 3.3) refreshed every 3 minutes at most (most of the time a new file is available every minute).
It provides the coordinates (latitude - longitude) of all the stations of bikes across London, their ID and name as well as the number of bikes available at the station, the number of empty docks and the total number of docking points. A straightforward calculation gives the number of locked docks: docking points where a bike is locked and unavailable users because someone assessed that this bike could no longer be used safely and should be reported to TFL maintenance teams.

With each new file provided by TFL there is a time-stamp to precise at what time the file has been updated and this enables us to know to what extent the information is likely to be outdated: even if the velocity with which the situation changes between two successive files depends undoubtedly



```
▼<stations lastUpdate="1303824571047" version="2.0">
  ▼<station>
     <id>1</id>
     <name>River Street, Clerkenwell</name>
     <terminalName>001023</terminalName>
     <lat>51.52916347</lat>
     <long>-0.109970527</long>
     <installed>true</installed>
     <locked>false</locked>
     <installDate>1278947280000</installDate>
     <removalDate/>
     <temporary>false</temporary>
     <nbBikes>4</nbBikes>
     <nbEmptyDocks>15</nbEmptyDocks>
     <nbDocks>19</nbDocks>
   </station>
  ▼<station>
     <id>2</id>
     <name>Phillimore Gardens, Kensington</name>
     <terminalName>001018</terminalName>
     <lat>51.49960695</lat>
     <long>-0.197574246</long>
     <installed>true</installed>
     <locked>false</locked>
     <installDate>1278585780000</installDate>
     <removalDate/>
     <temporary>false</temporary>
     <nbBikes>0</nbBikes>
     <nbEmptyDocks>36</nbEmptyDocks>
     <nbDocks>37</nbDocks>
   </station>
 </stations>
```

Figure 3.3: TFL Data feed about bikes stations

on the affluence at the observed time, overall we know that information which is 1 minute old is more likely to be accurate than information dated of 3 minutes.

The data comprised in this file is saved as an instance on the server and when a user starts to use the new plug-in and (automatically) sends to the server his location, the server parses the instance to return to the user the information about the closest stations to the user.

Therefore the strict minimum of information is forwarded to the user to save as much time and data connection as possible. Only the coordinates of the station matter for the plug-in on the mobile device: it enables positioning the stations on the Google Map view.

Originally, the number of bikes and empty docks at each station were also of importance since the user was also to fill this information (he was asked to count the empty docks and the bikes at the station) and the comparison between TFL-provided data and the user-provided data would serve to initiate the degree of reliability of the user. However this idea was put aside as this tiny gain of information about the user would imply making him work more, all the more so as counting empty docks and bikes can be a long and boring action at some stations with up to 60 docks.

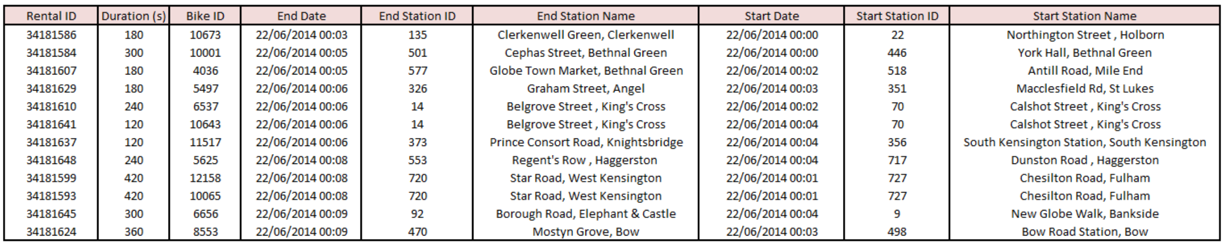
3.4.2 History of hires

Each month, TFL also makes public a CSV file (Figure 3.4) where every hire made in the month is an entry.

Each entry contains the ID of the bike used, the duration of the journey, the station's ID from where started the journey and the end station's ID.

One of the main downside of this data provided by TFL is that it is not released in live, thus it is not possible to use it in combination with the client application to help tracking the bikes that have been reported once and are moving across London without being reported all the time (recording the sequence of reports-non reports of a bike).
Another important drawback of the file is that no information is provided about the users. Only a rental ID is provided with each entry so there is no way to know how many times a month a given user uses the scheme nor be able to make the distinction between member users and more casual ones or even tourists.

However this file is still useful as it enables us to build a model simulating the dynamics of what happens in the reality: the paces with which bikes are damaged and get reported so as to measure the efficiency of the solution and if it is worth deploying on a large scale for real.



Rental ID	Duration (s)	Bike ID	End Date	End Station ID	End Station Name	Start Date	Start Station ID	Start Station Name
34181586	180	10673	22/06/2014 00:03	135	Clerkenwell Green, Clerkenwell	22/06/2014 00:00	22	Northington Street , Holborn
34181584	300	10001	22/06/2014 00:05	501	Cephas Street, Bethnal Green	22/06/2014 00:00	446	York Hall, Bethnal Green
34181607	180	4036	22/06/2014 00:05	577	Globe Town Market, Bethnal Green	22/06/2014 00:02	518	Antill Road, Mile End
34181629	180	5497	22/06/2014 00:06	326	Graham Street, Angel	22/06/2014 00:03	351	Macclesfield Rd, St Lukes
34181610	240	6537	22/06/2014 00:06	14	Belgrove Street , King's Cross	22/06/2014 00:02	70	Calshot Street , King's Cross
34181641	120	10643	22/06/2014 00:06	14	Belgrove Street , King's Cross	22/06/2014 00:04	70	Calshot Street , King's Cross
34181637	120	11517	22/06/2014 00:06	373	Prince Consort Road, Knightsbridge	22/06/2014 00:04	356	South Kensington Station, South Kensington
34181648	240	5625	22/06/2014 00:08	553	Regent's Row , Haggerston	22/06/2014 00:04	717	Dunston Road , Haggerston
34181599	420	12158	22/06/2014 00:08	720	Star Road, West Kensington	22/06/2014 00:01	727	Chesilton Road, Fulham
34181593	420	10065	22/06/2014 00:08	720	Star Road, West Kensington	22/06/2014 00:01	727	Chesilton Road, Fulham
34181645	300	6656	22/06/2014 00:09	92	Borough Road, Elephant & Castle	22/06/2014 00:04	9	New Globe Walk, Bankside
34181624	360	8553	22/06/2014 00:09	470	Mostyn Grove, Bow	22/06/2014 00:03	498	Bow Road Station, Bow

Figure 3.4: TFL CSV file of history of hires

3.5 Reality dynamics modelling

In order to test that the processing from one end to another on the server works fine with huge amount of data as we expect there will be in reality, we need to build a model from the real TFL data we have access to.

This modelling will also enable us to vary some parameters - such as the proportion of reporting users among the crowd using the bikes - and get a first estimation of how efficient is the solution and if it can be put in place and developed further.

We need to build a model as close as possible to the reality, however there are many figures we do not have access to and which are not publicly released, we can set these parameters and provide the developer an easy way to change them via the web interface.

As stated previously the main piece of real TFL data used to build this modelling is the monthly release of all the journeys that have been done in the month. However as it has been emphasized, the amount of information disclosed is minimal (no information at all on who is using the bikes nor on the frequency with which the bikes get damaged).

A great advantage of modelling the functioning of the scheme is that we are going to know what are the true labels of the reported bikes and thus we are going to be able to estimate the efficiency of the algorithm and adjust some parameters so that when the algorithm say that a bike is damaged or not actually damaged, it can be trusted with a very high certainty. If we think in terms of a potential future deployment, it means that this would prevent TFL maintenance teams collecting bikes not damaged for nothing, and thus would save both energy and time - and money!

3.5.1 Modelling damaged bikes

The different types of damage that have been considered all along this project are the following: tires, front brake, rear brake, saddle, pedalongs and direction. Other types of damage may be added to allow a more precise approach. This is not a crucial detail as the number of different damages does not impact the way the ML algorithm works, or the other mechanisms of processing and storage on the server side.

One of the key issues is to estimate when a bike is going to become faulty. The more the bike is used, the more likely the chances are that an element of the bike gets faulty. For this reason, we can model the probability that the bike becomes faulty for one element according to an exponential distribution with parameter λ .

For example for the tires, we can decide that the chances that one of the tires is going to become a minor problem (aggravating afterwards or directly serious enough so that the bike becomes out of service) after 100 hours of using the bike are equal to 1 %:

$$p(\text{tires damaged after 100h of use}) = 0.01$$

Given the cumulative distribution function (Figure 3.5) of the exponential distribution is:

$$f(x) = 1 - e^{\lambda 3600x}$$

x being the cumulative duration in seconds of use for the bike - hence the multiplication by 3600, we can solve the equation to find the value of λ.

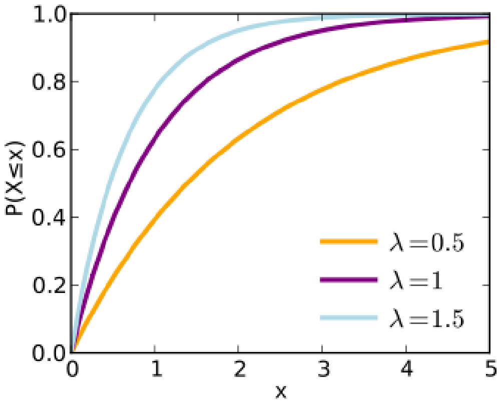


Figure 3.5: Cumulative distribution function of the Exponential distribution [19]

Each type of damage follows an exponential distribution with its own λ. Besides we make the assumption that the chances for an element to get damaged do not depend on the state of the other elements. For example if the brakes get faulty at some point, it is not going to increase or decrease the chances that the tires get faulty as well.

This was the first approach when designing the model statistics to manage the way and frequency with which bikes would get damaged. It seems the best one as we had accessed easily to the cumulative duration of use of each bike and because the exponential cumulative function is usually an interesting and accurate way to model the relationship of an ageing object and the probability it breaks down.
However, in practice, once set in place and the first tests run, the amount of bikes getting damaged was amazingly unlikely with this model: for instance, in 6 weeks there would be about 6000 bikes getting damaged out of the 10000 to 12000 bikes in the fleet: the number of them would increase in an exponential manner along the weeks.
This was confirmed by some figures found on the internet [20] stating that about 30 bikes are repaired a day in the depots of TFL. Therefore if we make the assumption that approximately the same number of bikes gets damaged and repaired in one day - for balanced flows of input / output - then in one week we should get between 200 and 250 bikes damaged.
This is the model that we adopt finally in the modelling for the bikes getting damaged and this is probably a good enough modelling given that in the end the number of bikes damaged is approximately the same each day, this seems to better model the reality since all the bikes have not been put into circulation at the same time, but progressively so it is expected that the flow of damaged bikes in the reality is approximately constant.
In terms of parameters set in the model, it boils down to get a bike damaged every $\frac{10000}{12}$ entries browsed in the CSV file.

Consequently for each entry of the CSV file (provided by TFL), we know the bike used and keep track of the total duration it has been used and then estimate the probability one of its elements gets faulty.

3.5.2 Modelling users' behaviour

Among our taskers the level of expertise varies from the user who never make a mistake (expert) to the one returning 100 % wrong answers - users that may answer honestly but who do know anything about bikes mechanics or users that want to spoil or false the studies, in any case users with unfriendly intentions. The issue is to assess how many distinct categories there is and the proportion of users in each category.

As a default parameterization, we work with four categories:

- the first category of users answers every task making no mistakes (experts). 10 % of the reporting users are in this category.
- the first category of users answers every task making a mistake in 15 % of the cases. 50 % of the reporting users are in this category.
- the first category of users answers every task making a mistake in 40 % of the cases. 35 % of the reporting users are in this category.
- the last category of users answers making a mistake each time. 5 % of the reporting users are in this category.

We assume further that a user cannot change from one category to another: his reliability is constant as the time goes by.

3.5.3 Case of non-damaged bikes

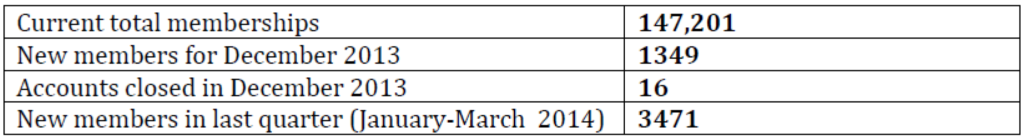
An expert cannot mistakenly report a bike as damaged if the bike is not damaged in reality (as the user is supposed to be an expert). However all the other users can do such an error either

because they honestly think an element of the bike is getting faulty, or because they take too seriously the task of reporting bikes and tend to become too exigent and too prompt to work.
The statistics that manage the pace at which bikes not actually damaged get mistakenly reported is similar to the one for generating damaged bikes: a constant number of these 'not damaged but reported bikes' is generated. The default value is set so that it happens every $\frac{10000}{3}$ entries - of the CSV file.

3.5.4 Number of users

This is a real issue and the current version of the solution is not quite realistic about it.

TFL provides annually the number of members for Barclays bikes. According to their last release of figures, it amounts at nearly 150k members (Figure 3.6) [22].



Current total memberships	**147,201**
New members for December 2013	**1349**
Accounts closed in December 2013	**16**
New members in last quarter (January-March 2014)	**3471**

Figure 3.6: TFL statistics of March 2014 [22]

On the other hand TFL also provides a bar chart (Figure 3.7) with the number of member hires and the number of casual hires (users that are not registered as members) for each month of the year.

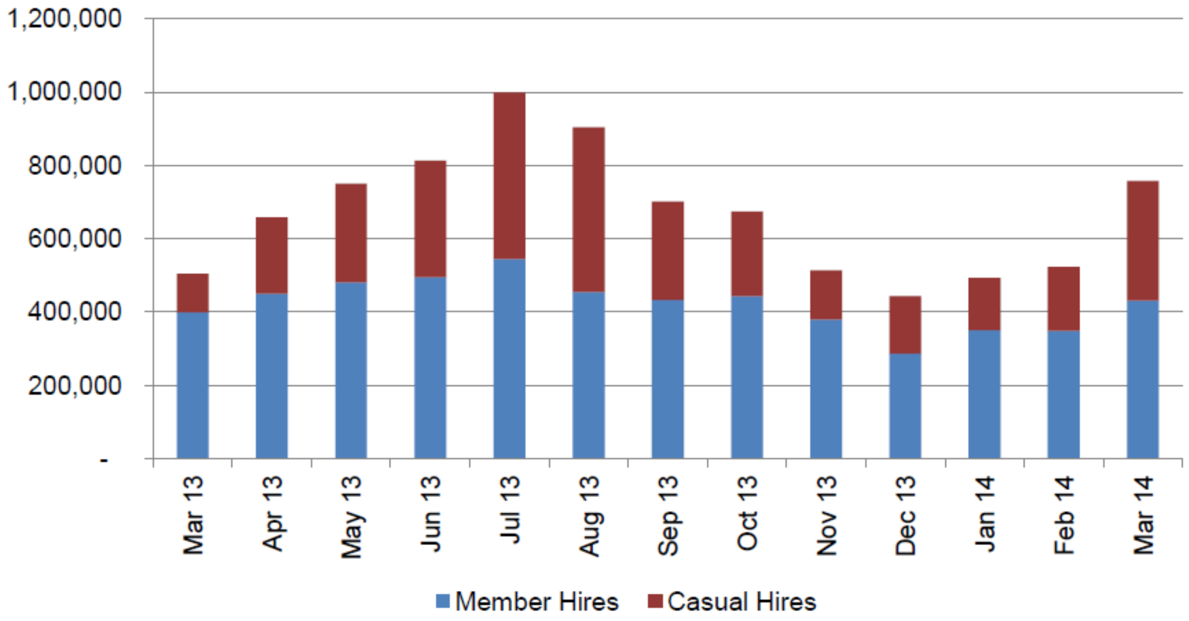


Figure 3.7: Member and casual hires by month [22]

However there is no means to find out the average activity of users throughout the year (how many times do they use the bikes on a monthly basis?). It is even most certainly more complicated than that: different types of behaviour must be existing among the members: members riding every day of the week to go to work, others riding only during weekends, and many other types
It gets also complicated when thinking about the different types of users among the ones that make casual hires: some are tourists when others are Londoners using the bikes on a very irregular basis.

Another issue to consider is the variation of the number of hires throughout the year: during the summer there are more users and proportionally more casual hires (tourists) than in the winter.

On top of that, we need to set up the proportion of users that are reporting users - users who installed the mobile application and participate actively in reporting damaged bikes across the city - and it is legitimate to ask whether some tourists might download the application (some tourists always download a bunch of more or less useless applications when visiting a new city) and use it and how important this trend is compared to reporting users among members: indeed a casual reporting user has not got as much history as a reporting user member of the Barclays Cycle Hire Scheme at TFL, and thus there is less information from him to evaluate his degree of performance and reliability.

Facing all these problems led us to choose a very simple way to tackle things at first, in the hope that we would have enough time to improve the model later: we choose to set the total number of users at 200k and the proportion of reporting users at 15% of them.

3.5.5 Reporting at station

As described previously, users reporting a bike at a station (after using it) are asked about bikes that may be at the same station and which have previously been reported.

However it would be unrealistic to consider that all users reporting a bike after using it take the time to answer a push notification asking them to check for extra bikes. There are multiple reasons that can be imagined for why users would not do it: either there are too much bikes to check, the user is in a rush and decides that he is not going to have the time to answer; or the user simply does not want to complete extra work; or another option which has the same result from the server point of view: the user is unable to tell if the bike is really damaged or not - checking a bike attached at a dock is more complicated than riding it and realising there is something wrong - and does not want to give back what seems to him a completely random answer.
We can also think about the fact that if a user is already taking the time to help TFL by reporting a faulty bike after using it, he might also be inclined to help a bit more and make the effort to answer back about bikes already at the station.

In the case when there are several bikes to check at the station we can imagine a mixed situation: despite being willing to answer back about these bikes, the user might find easy to give a feedback about a subset of these bikes and conversely encounters some of the difficulties stated above about the remaining subset.

Accounting for all these considerations, a proportion of how many feedbacks we will get out of all the ones asked - in the sense that one feedback is for one bike - has to be set initially: 75 % of bikes which require more feedback at a station actually get one. In the other cases, the user is unable to tell with a fair certainty or does not want to.

Among the feedbacks we get, we need to consider that there is going to be mistakes because all users are not experts.
In the case when the user is an expert it is actually really simple: either the bike is damaged and the expert reports the right damage, or the bike is not damaged (but has been reported mistakenly) and the user reports a ”not damaged” bike.
For the other three categories, it is a little more complicated: they might do a mistake - some even do mistakes all the time - while answering. The issue is: if the bike is damaged (for real), isn’t it a better chance that the user report a damage (even if it is the wrong damage) rather than a ”not damaged bike? If it is a ”not damaged bike, it is less complicated: when mistaking, the report will mention a random type of damage.

Therefore, if the bike is damaged, we set that there are 80 % of chances that a mistaking user report another random type of damage, and 20 % of chances that he report it as ”not damaged.

3.5.6 Tracking reported bikes

Using the CSV file, we need to be particularly careful when saving the location of reported bikes.

The advantage we have in the modelling part is that we can simulate the use and the dynamics of bikes getting damaged with the history released by TFL. Thanks to these files we can track a damaged bike from one station to another even if it is not reported and thus get a sequence of reports and non-reports for the bike.
In the reality this is impossible to do as this history provided by TFL is only released monthly and not in live ; therefore say a bike is reported at a given station and the following user does not report it, then our server has no means to know whether the bike has moved or is still at the previous station. It raises the following issue: when the server is going to send a push notification to a user to ask him to check for some bikes at the station where the user is standing, some or all the bikes might not be at the station and there is no means to know where the bikes are until there are reported again.

This is going to enable us to test different schemes of our solution:

- the first would be when we do not ask to users to check for already reported bikes that might be at the station: the percentage of reports at station is down to 0.
- the second scheme would be to test the efficiency of the scheme which is actually developed in reality (the one described so far): users are asked to check the bikes at a station but we are not sure whether the given bikes are still at the station or have already moved elsewhere. If one of the user's choice of answer is that the bike is not at the station anymore, we do not need to keep asking to all users arriving at the station whether this bike is here : we can get it out of the tracking system (we lost its real position).
- the third approach would be to suppose that we get the data provided in the CSV files in live and then we get access to the movement of a bike even when it is not reported (and we can ask a user to check the bike while being sure that the bike is at that station). This allows us to make the sequences of [reports, non-reports] for each reported bike in view of the CBR implementation.
- there is actually a last approach that I can think of: we could imagine saving somewhere the reported bikes that we do not know anymore where they are and asking each time a user is reporting a bike whether one of the missing reported bikes is at the station where he is. However the list of ”lost” bikes might get large and asking a user to check if one of the 50 bikes missing is at his station might bore him.

The third is the most attractive one but we need to be careful about the tracking of the bikes in the modelling. Indeed when we read the history of hires and come across a bike that we know is damaged, we need to ensure that the starting station of the journey is the same as the ending station of the previous journey of the bike, otherwise we might get the same bike saved at several different locations across London which is going to have an impact on the reports made by users: the bike might be reported more than it should as the users also make reports at the stations.
Such a situation - the previous ending station different from the starting station - is likely to happen as TFL teams regularly move bikes across London from full stations to empty ones to regulate the flows of users from suburbs to centre.

All the statistics figures that have been set up in this chapter can be modified in the tab settings of the web interface on the server.
There are lots of unknown parameters in the modelling that had to be set up to approach a realistic enough representation of the behaviour of users in the reality.

Figure 3.8 is a flow chart describing how each entry from the CSV files is logically processed. The model statistics being referenced are the ones we discussed all along that last section of the chapter.

3.6 Summary

In this chapter, we discussed of the system design: first for the mobile application on the client side, then on the server side: what needs should respond the implementation and also what constraints it should take into account. Finally we described the goal of modelling the reality dynamics (paces with which bikes get damaged and users report) thanks to the real pices of data realeased by TFL.

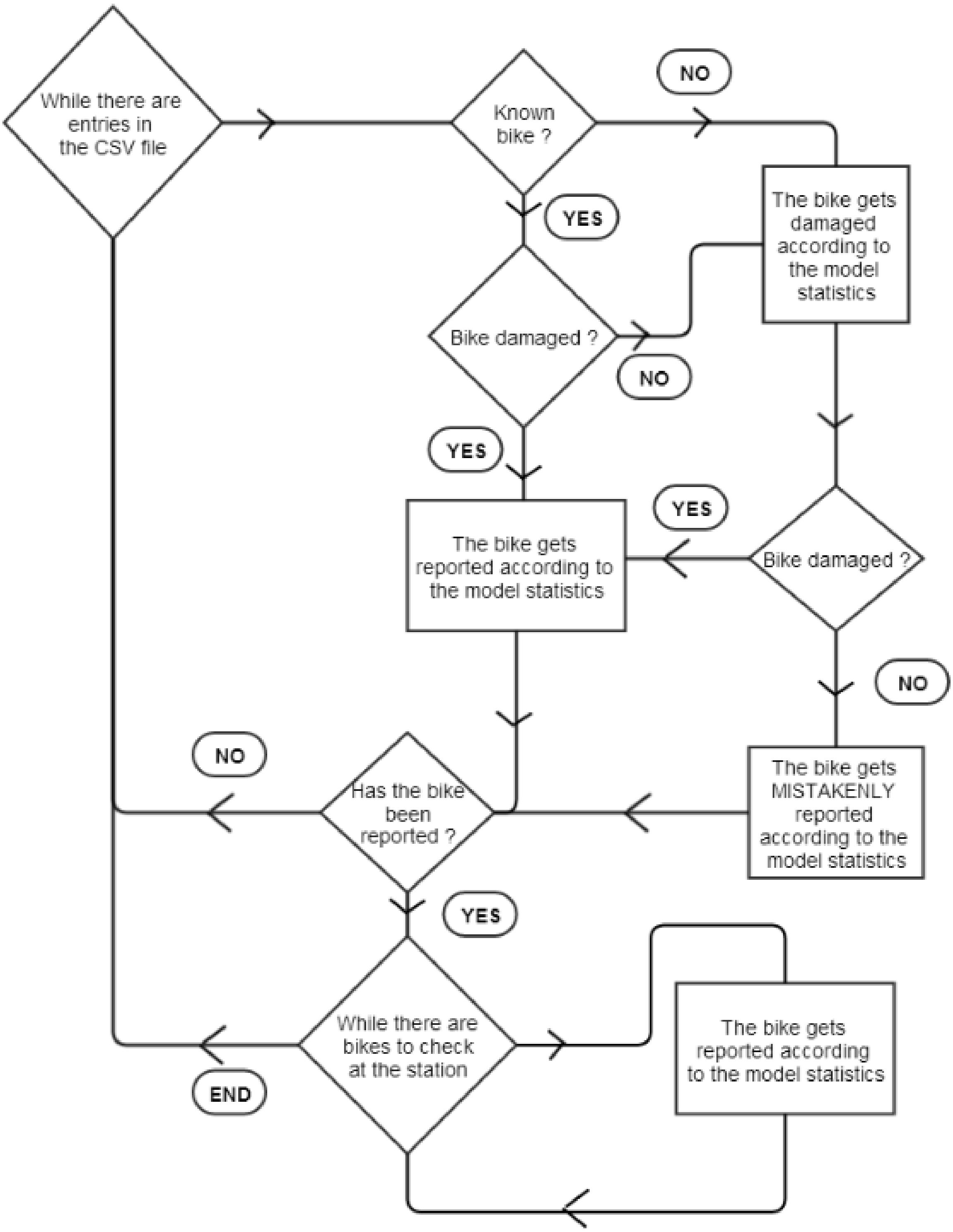

While there are entries in the CSV file
Known bike ?
NO
YES
The bike gets damaged according to the model statistics
Bike damaged ?
NO
YES
The bike gets reported according to the model statistics
YES
Bike damaged ?
NO
The bike gets MISTAKENLY reported according to the model statistics
NO
Has the bike been reported ?
YES
While there are bikes to check at the station
The bike gets reported according to the model statistics
END

Figure 3.8: Flow chart of the processing of CSV files from TFL

Chapter 4

Implementation

The implementation follows the principles described in the previous Design chapter. This chapter focuses on how the code has been organised in practice and what is the overall structure and relations between the diverse components. It also relates the main technical difficulties encountered and the way they have been tackled and resolved.
In a first part, the implementation of the new Android plug-in that extends the Hive framework is explained. The general structure is presented and the situation when each of the different activities play a role is described. All the specifities and details that are slightly particular compared to standard implementations on Android are outlined.
The second part deals with the details of the implementation on the server side: what new features have been added to the original code of the Hive framework (and where these modifications have been made) as well as the different sub parts of the additional functionalities: the process that fetches TFL data about stations to forward it to the mobile application, the web interface which is used by the developer to display diverse contents, the process of extracting the uploaded data - from the mobile application, the process of running the ML algorithm and saving the results - it consists in categorising the predictions according to the certainty returned for each prediction - and the processes of testing the whole solution.

4.1 Client side

On the client side, the only new piece of code is the new plug-in which enables the user to fill the form to report a bike after using it and fill the form to report the state of bikes already at the station.
The same plugin is coded to achieve both purposes. However, depending on the situation, the form changes and the actions and processes realised in background are different.

In the first situation, the user needs to select the station where he is standing by so that the server knows where he is, thus when the user launches the plugin to complete the first stage of the task, the following steps are executed:

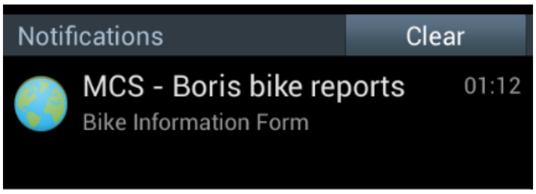
- we determine the location of the user
- we send the GPS coordinates of the user to the server, along with the radius of the circle (distance in meters, by default 300m) in which we want to get the stations of Barclays bikes.
- we receive the data about the stations from the server and display both a list of stations and a google map view with pinpoints at the places where the stations are
- the user needs then to select a station (either via the map by clicking on one of the icon or via the list of stations)
- the user fills the form to report a bike and saves (to be able to save the output file on his device, he needs to give the plug-in the right to write on his hard drive and thus accept the notification he has received)

The user can change the radius of the circle (searching distance around his location) by pressing the Settings item in the menu bar when he is in the MainActivity (the one displaying the Google Map view).
In the second situation, the server knows where the user is because he normally just sent a report with his location in the data. Thus there is no need for the plugin to find the location again and request information about surrounding stations. The server initiates the stage:

- The server sends a push notification (Figure 4.1) back to a user if there are bikes to check at the station the user is standing by. This notification contains the IDs of the bikes to check.

- If the user clicks on the notification, it starts the stage with the new plug-in for the second situation: it goes directly to the form.

- The user can fill the form completely or partially (or quit) and save the output (permission needed to save the file, as before).



MCS - Boris bike reports 01:12
Bike Information Form

Figure 4.1: Push notification on the mobile device

In both cases, the stages of the task that come after the form has been filled and saved get the upload plugin perform its role: it is designed to access the file saved by another stage and send it to the server. The functioning of this plugin has not been altered from the original version made in the Hive framework [12].

When building the task on the server side via the user interface created with the Hive framework, the ID of the stage for which the upload plugin needs to access the file has to be specified.

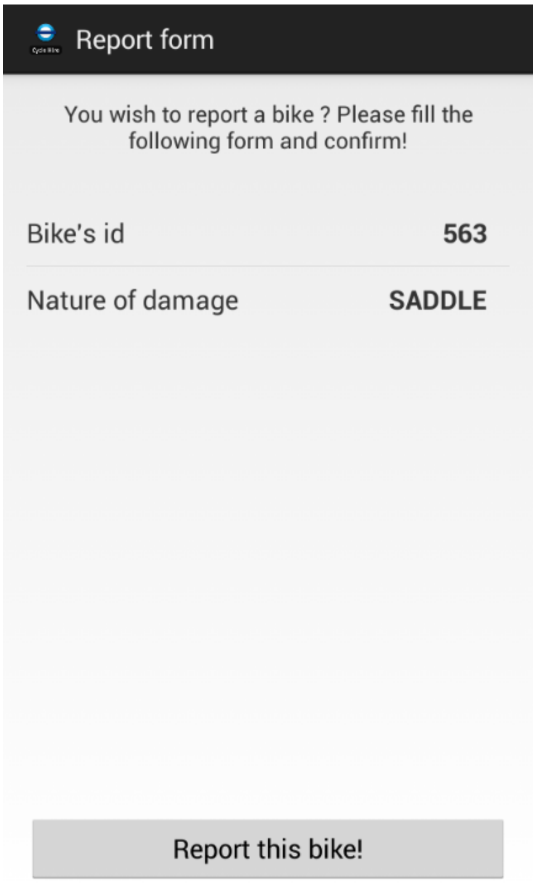
In the new plug-in code, there are three main activities:

- MainActivity displays the Google Map view and the list of surrounding stations. It is also the activity from where the communication with the server is made.

- DamagedBikeActivity (Figure 4.2) is the one where the user fills the form when he wishes to report a bike after using it. It is launched when clicking on a station (either on the map or in the list) in the MainActivity.

- CheckBikeActivity (Figure 4.3) displays the form to fill when receiving the push notification to check some extra bikes. In reality the MainActivity is at first opened when clicking on the notification, though it detects that it has been opened by the notification and directly redirects towards the CheckBikeActivity.

The SettingsActivity is only used to set the value for the radius of the circle centred on the user's location.



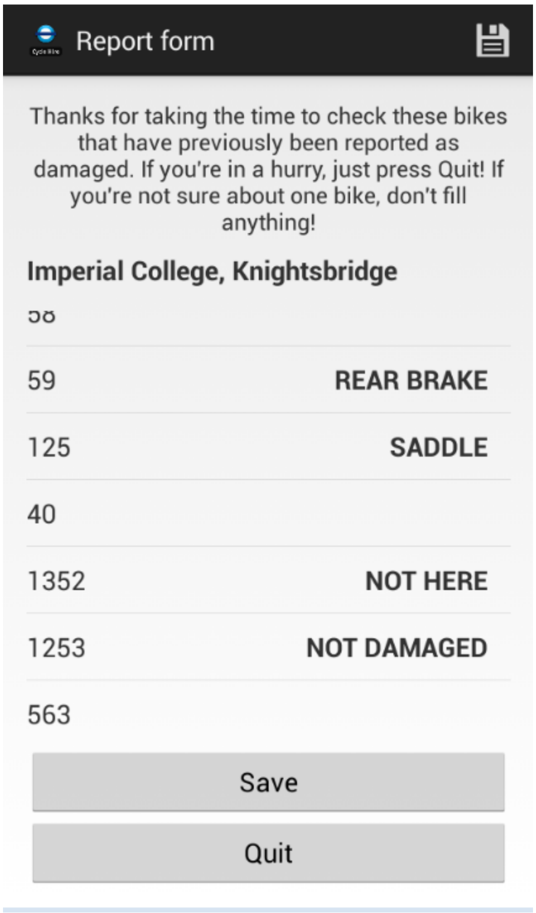
Report form

You wish to report a bike ? Please fill the following form and confirm!

Bike's id **563**

Nature of damage **SADDLE**

Figure 4.2: Form to fill after using a bike



Report form

Thanks for taking the time to check these bikes that have previously been reported as damaged. If you're in a hurry, just press Quit! If you're not sure about one bike, don't fill anything!

Imperial College, Knightsbridge

58

59 **REAR BRAKE**

125 **SADDLE**

40

1352 **NOT HERE**

1253 **NOT DAMAGED**

563

Figure 4.3: Form to fill for checking bikes

4.1.1 Building on top of the Hive framework

For this new plug-in to work well in accordance with the existing framework managing the plugins, some specific points need to be respected.

As described in the report of Dimitar [12], the plugin needs to have a class that implements MCSStageApp.
This class has been called BikeInformationApp and is placed in the main package of the plugin.
This is the main link to the framework and enables us to get many pieces of information about the task and the current stage from within the plugin.

The MainActivity of the plug-in also needs to extend the StageActivity of the framework which (among other things) takes care of managing the permission's answer of the user (allow or deny) to the notification about accessing or saving a file on the hard drive.

4.1.2 Details about MainActivity

This activity has got a crucial role in the functioning of the overall plugin.
First of all, it initiates the process of asking the permission to the user (via a notification) for saving a new file on the hard drive, which is something that is going to happen if the user fills one of the two forms.
As described shortly before, the activity is launched in both situations at first and for this reason knows in which situation the user is using the plugin (either after using a bike or for checking new ones at a station after receiving a push notification from the server).
In addition to that, in the case when the plugin is used to report a bike after using it, this activity manages all the communication with the server: it sends the coordinates of the user (and the radius

of the circle) and gets back the information about surrounding stations.
Finally, the other two activities consistently send the information collected via their respective form to this activity so that the information can be processed and put in the required format (XML) before being saved.

Finding the location

Determining the location of the user (i.e. determining the couple of GPS coordinates) happens via two different ways - in any circumstance, finding the location is only possible when the stage of the task matches with the case of reporting a bike, in the case of checking a bike, we already know where the user is: no need to determine its location.

- either when the plugin is launched, or recalled from the background (activity resumed - the user did something else after starting the plugin). The geolocalisation is then automatically launched.
- or it can be launched manually by the user by clicking on the refresh item in the menu bar of the activity.

In both cases, the location is refreshed and a new request is made to the server about stations surrounding the user.

The Google Maps Android API v2 is used to get access to the location - at first I had developed my own version of code to determine the location from the sensors of the device, but it turns out that the Google API is optimised for saving battery on the device and provides the location faster and with more accuracy [26].
The code that had been developed at first is in the class GPSTracker, I left it here though it is not used should someone want to use that instead of the Google API. Indeed the only disadvantage of the API is that in the settings of the device, the user has to authorise Google to have access to his location (and who knows what they do with that..).

In order to get the Google Maps Android API v2 work on the mobile device, the Android manifest has to be modified and the key generated through the Google App Engine console has to be present in it. The documentation is clear about the modifications to make so that everything works fine [26].

The activity implements both interfaces GooglePlayServicesClient.ConnectionCallbacks and GooglePlayServicesClient.OnConnectionFailedListener.
On the start of the activity, we try to connect to Google services, and if the listener redirects us to the connected method, then we can start looking for the user's location. Otherwise it is likely there is an issue with the internet connection.

The location is made with the instance of LocationClient of the package com.google.android.gms.location which contains a method getLastLocation.
After determining the location, two things happen: the request to the server to get information about the surrounding stations and the positioning of the user on the google maps view.

Receiving information about stations

This is done via an asynchronous task. We send a GET request to the required URL of the server - we use here the RESTful services to transmit parameters via an HTTP request and get information back:

`https://mcs-imp.appspot.com/api/bikes/`

with the needed parameters: ID of the task, ID of the stage, GPS coordinates and radius of circle centred on the position.
This GET request is done in the doInBackground method of the AsyncTask. It waits for the answer from the server and if the server's response is successful (HTTP response 200), the ObjectMaper instance parses the data formatted in JSON to transform it into a list of Station objects; then the list is passed to the onPostExecute method.

The onPostExecute method of the AsyncTask analyses the content of the stations variable and decides what to do next: if the list is null there has been an issue when retrieving the data from the server, otherwise it calls the processData method of the activity which is going to display the stations on the map and list them into the activity.

The time required to get the data from the server depends on many factors and thus can vary a lot:

- the server may have to refresh its own data about the stations - i.e. get anew the data from TFL feed, this takes time.
- the number of stations to get varies in function of how large the user set the radius of the circle
- the quality of the data connection of the device can be very poor

For these main reasons, the implementation of AsyncTask was absolutely necessary. Besides it prevents the user from experiencing a freeze of his screen while the request is processed.

Positioning on the map

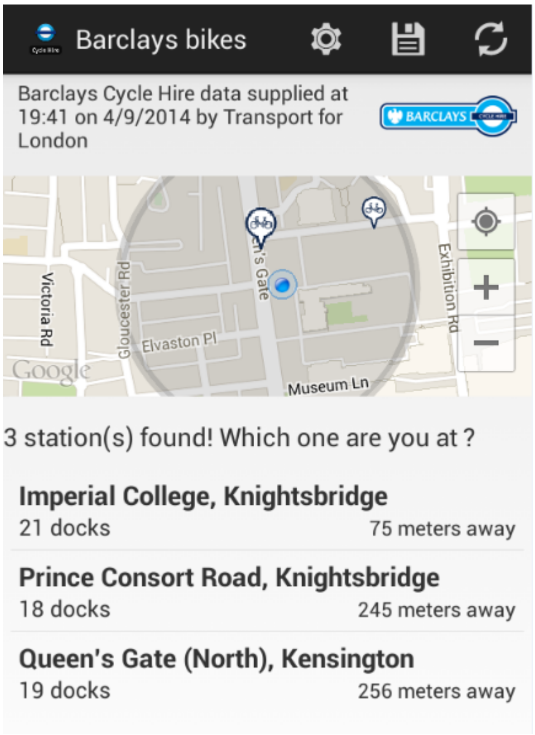
A zoom is made on the position of the user (Figure 4.4). It can happen at times that the circle which should be centred on the position of the user is actually not centred: this depends on the level of precision of the geolocalisation. The result is that stations fetched on the server might not be the ones desired.
A manual press on the refresh item in the menu bar generally re-centers correctly the circle on the user's location and fetches again the stations.

Saving the data

Both BikesDamagedForm and BikesCheckForm activities send the information of their forms back to the MainActivity.
If the user has given the plugin the permission to save the file on the hard drive, the data is saved and the user does not have the time to realise there is a call-back on this activity.
However if the notification for the permission has not yet been accepted by the user, a dialog windows is opened on the MainActivity to ask the user to do it before saving.

Saving is then done by pressing the corresponding item in the menu.
The action calls the static method implemented in the DataToXML class (in the tools package). This method formats the data correctly into a string and returns the string to the MainActivity which can write the data into the XML file onto the hard drive (method writeInfos).
All this processing is done into another AsyncTask. The onPostExecute method finishes the activity and notifies the framework (the main application managing the plugins and the progress in the stages of the task) that this stage is finished - the framework then manages itself the fact that the next stage will be launched when pressing the "Begin" button on the task screen.

The output file saved on the hard drive of the device is of XML format. It contains:



Barclays bikes

Barclays Cycle Hire data supplied at 19:41 on 4/9/2014 by Transport for London

3 station(s) found! Which one are you at ?

Imperial College, Knightsbridge
21 docks 75 meters away

Prince Consort Road, Knightsbridge
18 docks 245 meters away

Queen's Gate (North), Kensington
19 docks 256 meters away

Figure 4.4: Geolocalisation of the user and positioning on the map

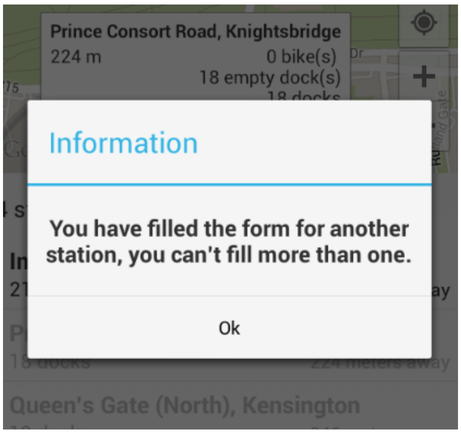
- the station ID
- the station name
- information about number of bikes and empty docks at the station at the time when the user reported the bike
- the timestamp coming from the moment where the XML feed on TFL website about stations was last updated
- the ID of the task and ID of the stage
- the ID of the user and a list of damaged bikes reported.
- a string that contains either ”use or ”report and serves for the server: as both activities for forms use the same structure for saving information, there needs to be a way to differentiate between information from a check form or a report form (the processing on the server is different).

The names of the attributes in the XML file are (when possible) the same as the name of the fields in the DataStation class so that the parsing can be done more easily on the server.

4.1.3 Protecting against unfriendly users

As it is always required when there is the need for untrustworthy users to fill information and send it, some protections have been implemented: for both types of forms, the user cannot save the form (Figure 4.6) until the minimum of information has been filled - basically the user cannot send back to the server a useless file with no consistent data in it.
In the case when the user has filled the form for reporting a bike after using it, validates the form and comes back to the MainActivity screen, if he has not allowed the plug-in to save the file he will be ”blocked” on this screen until he accepts the notification asking for the permission. Being on this screen, we need to prevent that he clicks on another station - either in the list or on the

Google Map view - and reloads the form 4.5. In the same time if he wishes to modify the data he just confirmed, this should be logically possible: if and only if he clicks on the station for which he has just filled the form, he can reload the form with the saved data he had previously confirmed.



Prince Consort Road, Knightsbridge
224 m 0 bike(s)
18 empty dock(s)

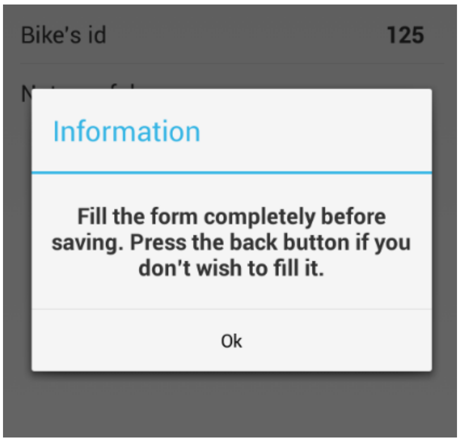
Information

You have filled the form for another station, you can't fill more than one.

Ok

Queen's Gate (North), Kensington

Figure 4.5: Protection user: form already filled for one station



Bike's id 125

Information

Fill the form completely before saving. Press the back button if you don't wish to fill it.

Ok

Figure 4.6: Protection user: form incomplete

With the same spirit, it is impossible for the user to access the form for checking bikes without receiving the corresponding push notification: when the current stage of the task is the one for checking bikes, if the user opens the plugin it closes itself automatically because there is no dynamic input associated with its opening - dynamic input that normally comes from the content of the push notification.

However, I am sure there are still things I did not think of that should be protected. One can never imagine what the user is going to do and how maleficent he is going to be.

4.1.4 Additional tools

Some useful tools have been implemented and placed in the package tools of the plugin code.

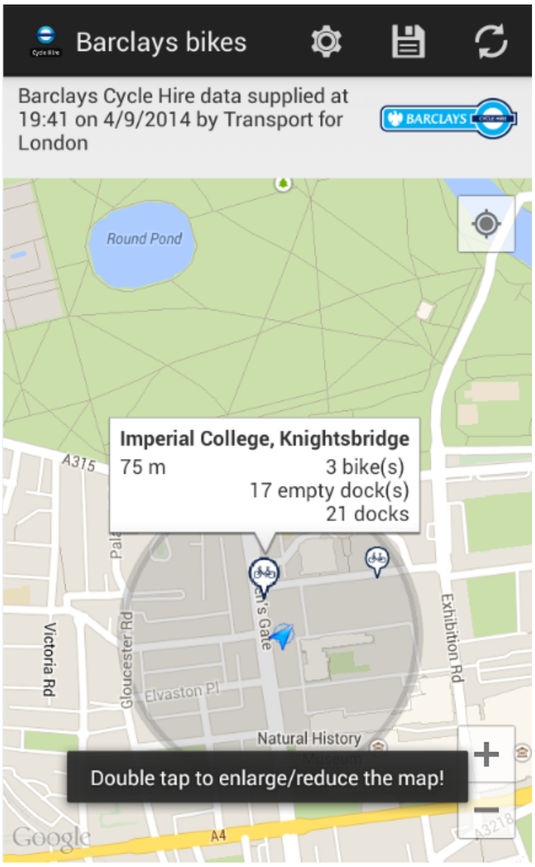
There are three classes that are personalised implementation of ListAdapter that enable to display information in ListView widgets with a personalised layout.
They are associated with the ListView widgets which are in both forms and the one that lists the stations in the MainActivity.

The SimpleGestureListener is implemented by the MainActivity and only the doubleTap method is used to alternatively enlarge or reduce the Google Maps view (Figures 4.7 and 4.8).
This interface also implements methods to listens to simple swipe gestures in the four directions (up, down, right and left).
It was originally used in the plugin but has been left aside for practical reasons - the navigation between the activities would get confusing at times and the user could also accidentally triggers the swipe listener while moving on the map.

4.2 Server side

4.2.1 Building on top of the Hive framework

The implementation of the new plugin we developed requires a matching implementation on the server side so that when we create the task with its stages on the web interface (already developed



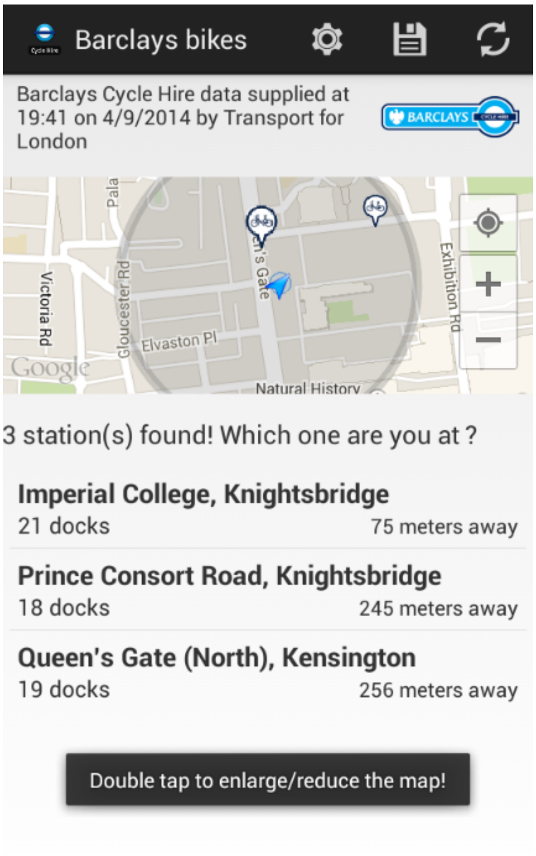
Barclays bikes

Barclays Cycle Hire data supplied at 19:41 on 4/9/2014 by Transport for London

Imperial College, Knightsbridge
75 m 3 bike(s)
17 empty dock(s)
21 docks

Double tap to enlarge/reduce the map!

Figure 4.7: Enlarged map view



Barclays bikes

Barclays Cycle Hire data supplied at 19:41 on 4/9/2014 by Transport for London

3 station(s) found! Which one are you at ?

Imperial College, Knightsbridge
21 docks 75 meters away

Prince Consort Road, Knightsbridge
18 docks 245 meters away

Queen's Gate (North), Kensington
19 docks 256 meters away

Double tap to enlarge/reduce the map!

Figure 4.8: Reduced map view

into the Hive framework) it proposes the new plugin as a possible stage. This is done in the class BikeInformationDescription of the package com.dp.mcss.meta. The MetaSpecDAO class into the package com.dp.mcss.storage needs also to be updated so that everything works fine. Implementations on these files have been inspired from the code already present for other plugins.

Besides these slight additions of code in the original code of the Hive framework, the method uploadStageFilesCB of the class TasksResource of the package com.dp.mcss has been modified so that when a file is uploaded coming from a user - some form data - the adequate processing of it is called: this calls the ProcessRealUpdateData methods that manages the processing and storage of bikes and users provided by real mobile devices.

4.2.2 Get the TFL live feed

One purpose of the server is to get the information publicly released by TFL about the stations of bikes across London, save this data and make it easily accessible for the users on their mobile devices.

This information is provided in a XML file accessible at the following URL (Figure 3.3) : `http://www.tfl.gov.uk/tfl/syndication/feeds/cycle-hire/livecyclehireupdates.xml`

RESTful services are used to process the requests of the users concerning the information of the stations they need (stations included in a circle centred on their location and of radius a distance they can set up themselves).
The class implemented for this purpose is BikesResource. When a user sends his location to the server, the parameters: longitude, latitude and the radius of the circle are sent as header parameters of the GET request to the server.
If the instance of Stations that is stored in the cloud is too old compared to the present moment

(older than 60s as set at the moment) then the server fetches the XML file and parses it with JAXB to transform it into a Stations object.

The Stations object is made of a field that stores the timestamp at which the last update about stations was done by TFL and another one which is a list of Station objects.
Important fields of instances of the class Station are its coordinates, the number of empty docks and the number of bikes that can be hired (some bikes are locked to the dock, it happens when a user wants to report TFL a damaged bike).
JAXB uses the name of the field to do the mapping between the XML attributes and the fields of the class. If the field's name is to be different from the name of the attribute in the XML class then the annotation @XmlElement(name = " .. ") needs to be used to precise the name of the attribute in the file that has to be mapped with this field.
The positive point of using JAXB is that it enables us to embed the Station objects into the Stations one as a list of Station.
The annotation @Embed needs to be indicated before the name of the class embedded, i.e. in this case the class Station.

The Stations object is also an entity and thus contains the annotations used by Objectify (@Id, @Entity, @Cache, @Index) to store and access the data into the Datastore of GAE.
As there is no special need to store more than one instance of the Stations class into the Datastore, each time the live feed is analysed from TFL website, the existing instance is replaced by the new one. The reason is quite simple : we could imagine a further development where two instances are compared to provide a certain functionality (in which case it would not be complicated to slightly alter the way the objects are stored : an unique ID by instance would need to be put in place instead of the only ID possible for now), but at the moment such a process does not exist and as the Datastore free space is something precious, there is no need to keep useless and outdated instance of an object in the cloud.

The coordinates and searching distance provided by the user (radius of surrounding circle) are used by the StationsDAO class to calculate the distance to every station in London. A sub list of all the stations which distance from the user coordinates is shorter than the radius of the circle is made and sent to the user formatted in JSON.

4.2.3 Web interface

There is a main servlet implemented in the HomeServlet servlet that is used to display information and process the requests of the developer.
The servlet is associated with the dynamic URL /home. The URL can take a tab parameter that enables the developer to go from one tab to another.

Bootstrap has been used to organise the display with a minimum standard quality [27].
The HTML and JSP code is written into the files that can be found in the war/WEB-INF directory of the project. The general structure (tabs and contents called in each tab) is written in the homePortail.jsp file.
Depending on the tab, an adequate JSP file is called from the homePortail.jsp file and displays the specific content that we want to be displayed in the given tab. Therefore there is at least one JSP file for each tab. In the ML tab there are sub tabs:

- tab predictions with predicted labels for each bike and the recall, precision and f1-measure of each user.
- tab data (with the matrix of labels)
- tab details with the confusion matrix of each user

Hence in the case of the ML tab there are three distinct associated JSP files.

All the JavaScript code which exists is written at the end of the homePortail.jsp file. The few functions written serve to hide or show the details of the reports in the reported bikes tab, and also serve in the simulation form tab to add or remove rows in the form that can be used to report new bikes.

4.2.4 Managing the bikes and users

All the processing done on the DataStation object to sort and reshape the data that is going to be stored into the ReportedBike and ReportingUser instances is done into the only instance of DataStationProcess. This instance is linked to the HomeServlet instance and the ProcessUploadData (used by the modelling) or the ProcessRealUploadData (used in reality to extract the data from the blobstore files uploaded by users via their mobile devices).

The instance of DataStationProcess takes a DataStation object (and also needs to know in which dataset working) and consults the list of existing instances of ReportedBike and ReportingUser (stored into distinct hashmaps) to determine if the new information that brings the DataStation object needs to be added to a particular instance or if new objects need to be created (a new instance of ReportedBike, a new ReportingUser or both).

When the DataStation object comes from the ProcessUploadData instance, true damages about the bikes are known and the dedicated field in the ReportedBike objects can be set.

Once a bike is considered as repaired, the history of its reports is copied into another field to clean the field reports used to determine whether the bike can be processed into the algorithm.

Similarly its category is reset as if the bike was reported for the first time so that when after being repaired or identified as ”not damaged the bike still gets reported: indeed when a bike is classified in the category with the highest certainty, it cannot be processed by the algorithm as we are sure of the result.
This way in the case (this should happens very scarcely but still) when a bike damaged is labelled as ”not damaged with the highest certainty, no TFL team will be assigned to fetch it but as the bike will still be reported it is going to be reprocessed shortly after and hopefully this time it will be well classified.

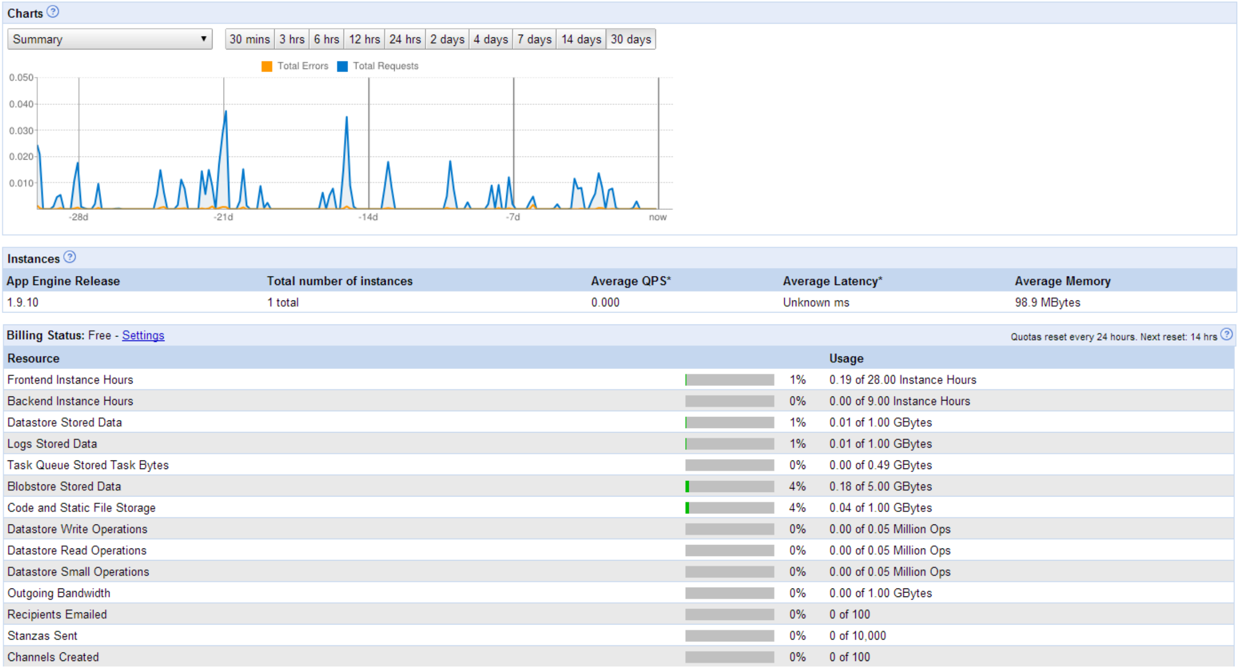
For users, we just need to update the field that stores the IDs of the bikes that the user has reported as this field is used to select what bikes are going to be taken into account and processed for the upcoming run of the algorithm. Bikes that are in this field must remain only the bikes that are reported but not yet repaired or identified as not damaged (bikes that have not been classified with the highest certainty of prediction by the algorithm).

4.2.5 Datasets

Two types of datasets have actually been implemented. While testing the good functioning of the server side elements, I started uploading huge amounts of data coming from the history of hires provided by TFL.

I quickly realised that given the average number of entries these files contain (about 35k entries on average for one single day) and the total number of distinct bikes (10k to 12k), fetching and saving each time the bikes and users concerned in the cloud represents a huge amount of requests to the Datastore. The problem is that when using GAE for free, there are some quotas existing and among others the number of small operations on the Datastore is limited to 50k a day (Figure

4.9).



Instances

App Engine Release	Total number of instances	Average QPS*	Average Latency*	Average Memory
1.9.10	1 total	0.000	Unknown ms	98.9 MBytes

Billing Status: Free - Settings

Quotas reset every 24 hours. Next reset: 14 hrs

Resource		Usage
Frontend Instance Hours	1%	0.19 of 28.00 Instance Hours
Backend Instance Hours	0%	0.00 of 9.00 Instance Hours
Datastore Stored Data	1%	0.01 of 1.00 GBytes
Logs Stored Data	1%	0.01 of 1.00 GBytes
Task Queue Stored Task Bytes	0%	0.00 of 0.49 GBytes
Blobstore Stored Data	4%	0.18 of 5.00 GBytes
Code and Static File Storage	4%	0.04 of 1.00 GBytes
Datastore Write Operations	0%	0.00 of 0.05 Million Ops
Datastore Read Operations	0%	0.00 of 0.05 Million Ops
Datastore Small Operations	0%	0.00 of 0.05 Million Ops
Outgoing Bandwidth	0%	0.00 of 1.00 GBytes
Recipients Emailed	0%	0 of 100
Stanzas Sent	0%	0 of 10,000
Channels Created	0%	0 of 100

Figure 4.9: Dashboard and quotas of GAE

As a consequence, when creating a dataset (tab settings in the web interface) there is a check box which enables the developer to authorise the storing of some often-used data in the Datastore.

The data concerned is:

- the instances of ReportedBike
- the instances of DataStation
- the instances of ReportingUser

This way of doing is particularly useful when testing the solution with the history provided by TFL for several weeks: quotas of small operations on the Datastore are not affected and completely wasted with the upload of only one day of history.

The major disadvantage is that objects have to be saved in memory in the case when the developer wants to work on another dataset.
For this reason the class SavedDatasetOutsideCloud has been implemented. When there is a change of dataset via the tab settings and if the developer was previously working on a dataset where the storage in the cloud was disabled, the following three hashmaps are saved in a new instance of SavedDatasetOutsideCloud :

- the hashmap containing the reporting users for the datasets
- the hashmap containing all the reported bikes (damaged, not actually damaged, already processed and repaired thanks to the algorithm)
- the hashmap containing all the stations at which there are bikes that have been reported but not yet classified by the algorithm in the category with the highest certainty on the prediction.

The DataStation instances are not important because once the information extracted from it has been dispatched in both the matching instances ReportingUser and ReportedBike, there is no need to keep track of it, there is no further processing about these instances.

The instances of SavedDatasetOutsideCloud are stored into a hashmap of the only instance of DataStationProcess that works with the servlet HomeServlet.
If the developer wants to switch back on this same dataset after having worked on another, then we look into this hashmap to check whether information about this dataset had been saved and if it is the case, we load the data into the instance of DataStationProcess.

The major downside of this way of doing is that for each new deployment of code for the application on the GAE server, there is a complete reset of all the objects in memory: all instances of objects that are not stored into the cloud are lost.

A dataset needs to be selected to go on the majority of the tabs of the web interface : indeed the simulation form tab and upload tab are a way to input some data and in the same spirit, tabs displaying bikes, users or statistics need to be told from which dataset the data has to come.

4.2.6 ML processing

The whole algorithm used for predicting the damage labels for each bike is implemented in the class MLBox in the package com.jg.ml. A similar code has beforehand been written with Matlab - before realising the implementation in Java so as to play a bit with the algorithm and understand its functioning - and is available in the archive of code submitted electronically.

The algorithm needs two inputs:

- one hashmap containing ReportedBike instances and which keys are the IDs (long) of the bikes concerned
- one hashmap containing ReportingUser instances and which keys are the IDs (string) of the users concerned

Selection of the bikes and users

As stated in the design chapter, we cannot consider all the bikes and all the users possible to run the algorithm: some bikes have only been reported once or twice and the predicted label for these bikes would not be sure enough.

On the other hand, given the way of functioning of the algorithm, we need to set a minimum of reports for each user taken into consideration in the algorithm so that the outputs have a good chance to be correct.
For example, it would be absurd to take into account a user who would have answered for only one bike among those selected, as the algorithm compares multiple answers made by one user with the answers for the same bikes made by all the other users (that did give a non-null annotation) to assess which subset is most likely right about each bike.

However we cannot expect that each user has an answer for a majority of the bikes also taken into account : the number of both users and bikes in the reality makes it already very improbable that each reporting user will have ride a significant subset of the reported bikes in a small timeframe.

Let's not discuss too much about the actual minimum number of reports a user has to be taken into account, but let us keep in mind that the first constraint in this step of selecting bikes and users is set on the users.

Once we set the minimum number of reports a user has to get to be considered in the algorithm, we need to browse the list of all the bikes he has reported (field bikesReported into the class ReportingUser) and that are still to be classified with the highest certainty and ensure that none of these bikes has already been classified definitively (in the category with the highest certainty), recount the number of bikes that respect this constraint and if the number is still superior to the minimum number imposed, we can add the id of these bikes in a list of relevant bikes that will be taken into consideration for the algorithm and add the user to the hashmap that will be passed as an input.

Once all the users have been processed, and the list of relevant bikes is completed, we can build the second input: the hashmap with the relevant instances of ReportedBike.

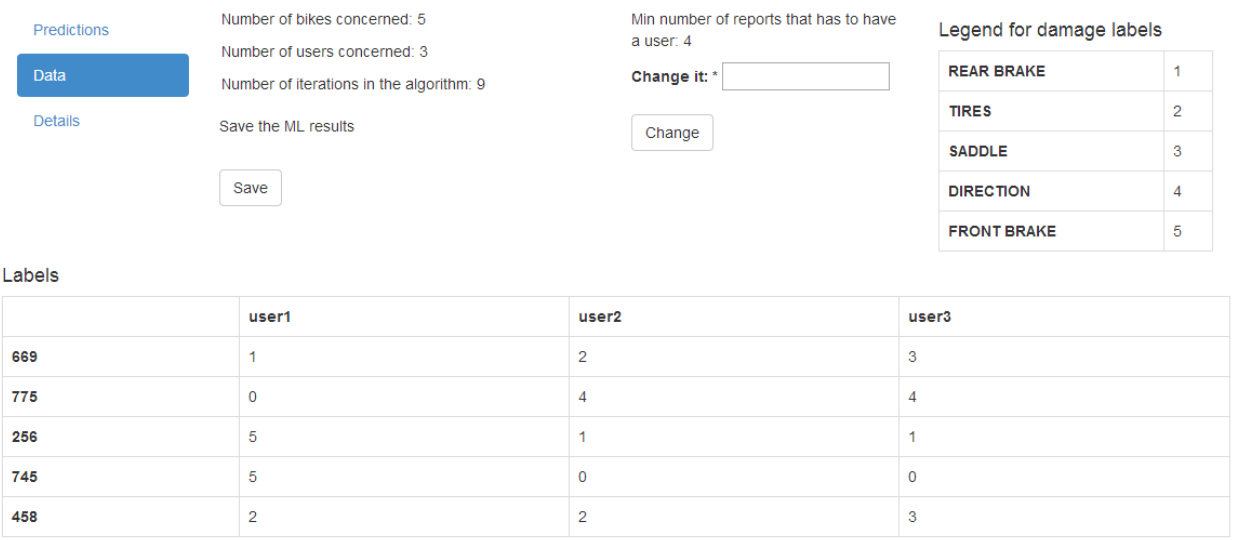
The methods for the selection are written in the DataStationProcess class as it is where we store as fields the lists of both reporting users and reported bikes for the current dataset.
Instead of duplicating these variables that can be heavy to manipulate we access the instance of DataStationProcess linked to the HomeServlet servlet to create the two desired subhashmaps.

Initialisation of the algorithm

The initialisation of the algorithm consists in setting the required parameters:

- total numbers of annotators which is the size of the hashmap containing the users J
- total number of bikes considered which is the size of the second hashmap N

Thanks to both hashmaps, we can start building the matrix called *labels*: for each bike we parse the list of reports (field reports) and check that the user associated to each report of the bike is present in the subset of users selected for the present run of the algorithm before assigning the damage to the user into the matrix (cf Figure 4.10).



Number of bikes concerned: 5
Number of users concerned: 3
Number of iterations in the algorithm: 9
Save the ML results

Min number of reports that has to have a user: 4
Change it: *

Legend for damage labels

REAR BRAKE	1
TIRES	2
SADDLE	3
DIRECTION	4
FRONT BRAKE	5

Labels

	user1	user2	user3
669	1	2	3
775	0	4	4
256	5	1	1
745	5	0	0
458	2	2	3

Figure 4.10: Labels matrix on the web interface

There is a hashmap mapping the different types of damages (formatted as strings) into a number. This hashmap is gradually filled when building the matrix of labels, so from one run of the algorithm to another the number assigned to a certain type of damage may change. This number is necessarily superior strictly to 0 as the number 0 is set in the matrix when the user has not given any feedback about the bike.

Once the matrix of labels is built, we know how large is this hashmap and it thus gives us the number total of classes (total number of different types of damages) K.

Before launching the algorithm, the last think that is needed is parsing all the users to get the performance of each one (which varies along the successive runs of the algorithm) and fill the λ matrix with these values as it used to initialise the confusion matrices Theta used to calculate the values in the Z matrix in the first E-step of the algorithm.

Post treatement of the algorithm

Once the algorithm is finished, we use the confusion matrix for each user to calculate the recall, precision and F1-measure of the user based on the formulas given in the design chapter.

The average F1-measure across all the classes may serve to redefine the user's performance (or degree of reliability) if it has changed during the algorithm (which is quite likely).

A slight and subtle modification I had to do in the calculation of recall and precision: say a user has not used all the different labels possible while labelling the bikes he has given an answer for; then there is going to be a problem in the calculation. Indeed, each row of the confusion matrix of the user necessarily sum up to 1 thus for the classes the user did not use, there is going to be figures in the corresponding rows and columns which are going to false the results of recalls and precisions for the other classes.

Therefore a simple way to avoid that is to save in a variable the whole set of labels that the user has used for the bikes present in this run of the algorithm. This is done in the initialisation part into the variable $damagePerUser$. Thus when calculating the recall and precision for a user in the end, we can skip all the rows (for the recall) and all the columns (for the precision) associated with classes the user did not use and the values of recall and precision (and thus F1-measure) for other classes are not affected.

Eventually as explained in the Design chapter, the performance of a user is determined based on the proportion of right answers among all the answers (labelling tasks) that have served at least once in a ML run of the algorithm.
The function that transforms this proportion to the performance is an exponential function that takes the proportion between 0 and 1 and outputs a value between 1 and 10 (Figure 4.11). This way we give even more importance and more weight to experts and trustworthy annotators, comparatively to users performing poorly. If a user has got a performance of 1, the consequence is that his answers will not weight in the algorithm, it is as if the annotator was not participating to the algorithm.

Categories of bikes

As it has been stated many times: there exists a category where we put bikes processed by the algorithm and for which we are sure about the predicted label, or at least the certainty of the result is very high (about 99 %).
In total there exists four categories; the reason for the existence of categories is that due to the huge amount of 0 in the labels matrix (corresponding of an absence of annotation by the user for the bike) make the algorithm perform with less good results than if the matrix was full of annotations other than 0s. The four categories are:

- the first one is labelled "SURE": a bike is classified in this category if the probability associated to its predicted label is superior to 0.9 and if the bike has at least 3 reports (we already

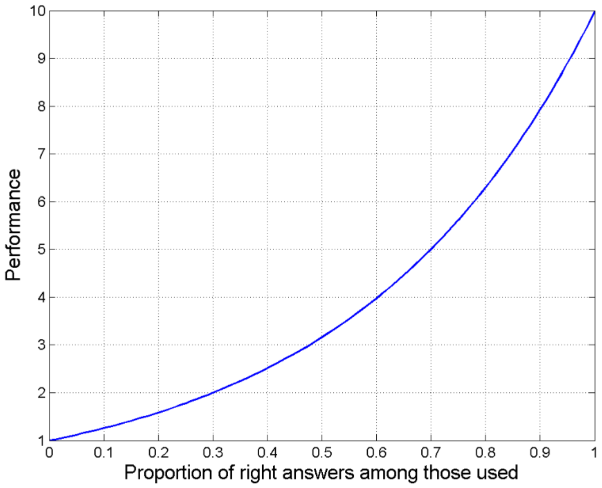


Figure 4.11: Function mapping proportion of right answers of an annotator to his performance

discussed this last point: a bike should not be processed by the algorithm if it has not got enough reports, we wait for more points of view from other users).

- the second one is labelled "NOT QUITE SURE" and corresponds to the interval 0.75 to 0.9 for the probability associated with the predicted label.
- the third category contains all the bikes that have been processed but since the probability associated with the predicted label is equal to 0, we know for sure the algorithm did not perform well on these and we put them aside for another run later.
- the last category comprises all the bikes not processed by the algorithm - not selected as inputs for the algorithm.

The figures of 0.9 and 0.75 can of course be changed. However they have been determined empirically so that the algorithm classifies bikes in the first category only when the predicted label is almost always equal to the true one: this way we minimize the error rate in the perspective where TFL teams would waste time, energy and money fetching too many ill-labelled bikes.
Determining empirically these key values was possible because in the modelling part we know perfectly the true damages of the bikes - since we generate them ourselves, so we could compare with the predicted labels.

In addition to the checks done so that a bike can be classified in the SURE category, there is an additional check which prevents a weird behaviour from the algorithm: we realise a quick check on the proportion of predicted labels among all labels for the bike: if less than 25 % of the labels are equal to the predicted one, it looks suspect and we place the bike into the second category. This second check is done into the DataStationProcess instance just before taking account the results and updating the state of the bike and the users concerned.

Limitation of GAE

When there are too many bikes and too many users considered for the run of the algorithm, it takes more time to build the whole matrix of labels and compute all the iterations of the algorithm.

The main issue is that all the processing must be completed within the maximum interval of time that can last the request. The maximum duration of a request is of 1 minute, hence from the moment we click on the tab ML , it has to fetch all the users and bikes needed, fill the matrix of labels and compute all the steps of the algorithm and return the results in the display zone before 1 minute has elapsed. When the number of bikes starts to exceed 800 or 900 and the number of users 300, this gets quite improbable.

Therefore the solution put in place consists in making several runs of the algorithm with keeping these two numbers low enough to be able to complete all the steps.
In the HomeServlet instance, we set a while loop just before the ML process is launched and we decide that if one of the two numbers is too high (the number of the users selected and the number of bikes associated), we increase by a given step the minimum number of reports that a user need to get to be taken into account.
Both the step and the starting value for the minimum number of reports by user are values that can be modified by the developer in the Settings tab.

4.2.7 Testing and modelling

There are two tools for testing the good functioning of the server side implementation without having to waste huge amount of time entering data from a mobile devices.

The first one is accessible via the tab Simulation form : it displays a form that simulates the action that could encounter a user on his mobile device. A random station is selected, the developer can select a user (currently there are 10 users that can be used, but in practice there can be as many as the developer wants: the change has to be done in the JSP file simulationForm.jsp where a larger number of users can be set in the select input) and report as many bikes even introduce as many types of damage as he wants.
As described in the design chapter, initially the mobile application contained a form for the user to fill information about the number of empty docks and available bikes that would be at the station; this is the reason why such information is also asked in this form but is prefilled so that the developer does not waste time with useless information.
This way of testing was used at first to test the good functioning of storage of bikes and users into the cloud and also different scenarios more or less complicated to test the limits of the algorithm. In particular, it helped realising the subtle error that could happen about recall and precision when users did not use all the labels possible.

The second one has largely been described in the design chapter, it consists in modelling the dynamics of the solution by simulating the way bikes get damaged and the reporting behaviour of users.
CSV files can be uploaded to the server via the web interface and the tab upload . The file must contain only 4 columns which are in the following order:

- duration of the journey (in seconds)
- ID of the bike
- ID of the ending station
- ID of the starting station

The CSV file provided by TFL has thus to be slightly modified: the columns displaying the name of the both stations (starting and ending stations), the data and time of the start and end of the journey and the rental id (which just consists in a number incremented for each rental and cannot be linked to any user) are useless and removed.
The reason for these deletions is simple: original files can weigh up to 120 MB and removing all these bits of information that are of no use at the moment enables us to decrease the weight down to about 30 MB. As space in the blobstore (where the files are stored after the upload on the server) is also subject to some quotas, this is a non-negligible gain.

In addition to these deletions the files need also a reordering based on the ending date of the journey (so the reordering needs to be done before the deletion) : ordering has to put the oldest ending date first and the newest ending date in the last line of the file.
The reason for the reordering is that it is more representative of what happened in the reality and thus it is better for our modelling. There is also a more technical and practical reason: this way the starting point of the journey is supposed to be, in the vast majority of cases, the same that the ending point of the previous journey. This is important because we track the moves of reported bikes across London and we save the ending locations, if the ordering was not done, then these two stations would be most likely different and a lot of modifications in the variables used to save the locations of the bikes would have to be made. This takes time in the processing of the CSV file and it is going to be underlined further, there is a limited amount of time to execute each request to the server. Therefore the more we save time computing things, the better it is.

Once the file is uploaded on the server, the blobkey of the file is given to the ProcessUploadData instance (which is supposed to be unique and linked to the HomeServlet instance) so that the file can be retrieved from the blobstore and read one entry at a time.
The mechanisms applied to each entry have already been described in the design chapter, and summed up on the figure at the end of that chapter (Figure 3.8).
As the file is being processed, some counters are put in place and being incremented to get some significant figures about the modelling. Among those significant numbers, we pay attention to:

- the total number of entries processed: that is the total number of hires
- the total number of reports made: this number does not include the reports made by users at stations
- the total number of reports made at the stations
- the total number of damaged bikes: this counter is incremented each time a bike gets damaged
- the total number of reported bikes: this counter is incremented only once for each bike, if the bike is reported several times (as it should be as time goes by) the counter is incremented only once. However if the bike gets repaired at any point and then gets reported again, in this case the counter is re-incremented.
  In this number there are both bikes that are actually damaged and bikes that reported even though they are not really damaged (mistakes).
- the total number of reported bikes that are not damaged in reality. From this number and the previous one, we can deduce the total number of reported bikes that are actually damaged (difference of the two).
- the total number of reports at station concerning bikes not actually damaged but whose report indicates damage (amplification of the mistake already made by another user: the bike keeps being reported as damaged when it is not in reality).
- the total number of reports at station concerning bikes not actually damaged whose report is right: the bike is reported as not damaged.
- the total number of bikes repaired (bikes which were actually damaged)
- the total number of bikes identified as not damaged and thus saving teams the effort of fetching the bike to finally realise it is not damaged.
- the total number of distinct bikes that have been used in the total interval of time browsed by processing the CSV files.

These two last numbers are available once a run of the algorithm is done. As it is going to be describe in more depth into the following chapter the idea is to alternatively upload data and run the algorithm across time (for example one run for each day of data).

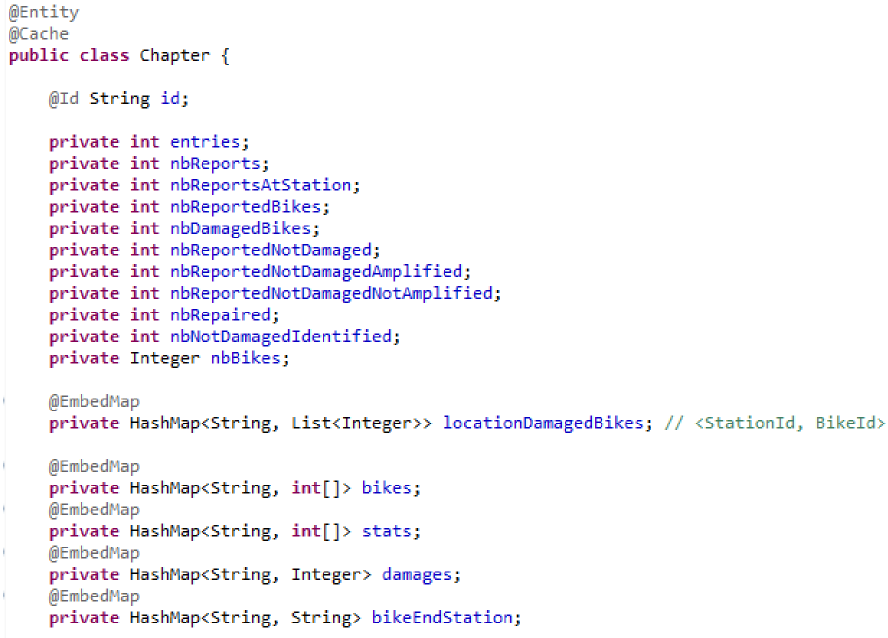
The ProcessUploadData instance also uses a int[] variable of length 4 for each bike in which is respectively stored:

- the cumulative duration of use of the bike
- the number of times the bike has been used up to now
- the integer associated with the nature of damage affecting the bike: this integer is set to 0 as long as the bike is not damaged or 1 to 6 for real types of damage. The integer corresponding to the category not damaged cannot be stored in here since the bike is not damaged in reality: not damaged is not a nature of damage even if it is used so in the algorithm.
- the total number of times the bike has been reported: if the number of reports is strictly positive but the damage is set to 0 this means that the bike is not damaged in the reality and this is how we can recognise a mistakenly reported bike and ask users at stations to give some feedback about it.

When one bike is repaired (via the public method removeDamagedBike called from the DataStationProcess instance), the damage is reset to 0 as well as its number of reports. Also the tracking of its location in the dedicated variables is cancelled.

A hashmap having for keys the IDs of the bikes and for values the matching int[] variable is built in the instance of ProcessUploadData. This hashmap, the one that match the different types of damages (strings) to integers, the other one that saves the last end station of all the reported bikes, and all the counters previously stated are stored into the cloud once the entire CSV file has been parsed.
The storage is done via the bean Chapter. The ID of the chapter is the same as the name of the dataset currently selected in the web interface (so the ID is a string).
The reason for this storage is that if a change of dataset were to be done between two successive uploads in the same dataset, we need to save all the variables present in the ProcessUploadStation instance.
Unfortunately there is one that we cannot store into the cloud because of its particular form: the hashmap *hBikes* where the keys are the IDs of stations where there are reported bikes and the values are lists of int[], each variable of type int[] is of length 2 and contains the ID of the bike and the integer matching its damage - this time the integer can be matching the not damaged label. This hashmap is the one read when a user is reporting a bike after using it at a station: all the reported bikes already at the station are read and asked about to the user.
Therefore this hashmap is not saved into the cloud for the modelling process because of its particular form not supported by Objectify and needs to be passed to the DataStationProcess instance so that it can be saved into the dedicated SavedDatasetOutsideCloud in the case when the dataset were to be changed.

We can notice that the Chapter class contains a field locationDamagedBikes. This has the same purpose than the hashmap described just above and that I said cannot be stored into the cloud. Except the values of this field are lists of integers and not lists of int[] : indeed in the reality we do not know the true damage of the bike and there is no need to store it as we do in the modelling process. Consequently in the ProcessRealUploadData instance (which processes the data uploaded in the blobstore by users from their mobile devices), the variable locationDamagedBikes will be slightly different from the one into the ProcessUploadData instance and will be saved into the cloud



```
@Entity
@Cache
public class Chapter {

    @Id String id;

    private int entries;
    private int nbReports;
    private int nbReportsAtStation;
    private int nbReportedBikes;
    private int nbDamagedBikes;
    private int nbReportedNotDamaged;
    private int nbReportedNotDamagedAmplified;
    private int nbReportedNotDamagedNotAmplified;
    private int nbRepaired;
    private int nbNotDamagedIdentified;
    private Integer nbBikes;

    @EmbedMap
    private HashMap<String, List<Integer>> locationDamagedBikes; // <StationId, BikeId>

    @EmbedMap
    private HashMap<String, int[]> bikes;
    @EmbedMap
    private HashMap<String, int[]> stats;
    @EmbedMap
    private HashMap<String, Integer> damages;
    @EmbedMap
    private HashMap<String, String> bikeEndStation;
```

Figure 4.12: Chapter fields

because the form is supported by GAE's tool Objectify.

In brief, both classes ProcessUploadData and ProcessRealUploadData use the same implementation of Chapter class, the difference is that location of the reported bikes is stored into the cloud for the reality case whereas it cannot be when modelling the data because of the particular form the hashmap takes.

However, because of this slight difference, the way of removing the bikes repaired (or identified as damaged) once the algorithm run and sorted some extra bikes into the category with the highest certainty is different. In the case of the reality, we need to access the Chapter instance stored in the cloud and delete the bikes in the locationDamagedBikes variable while for the modelling situation, we remove the bikes in the ProcessUploadData by calling the method removeDamagedBike from the DataStationProcess instance.

4.3 summary

In this chapter, we described in details the implementation both on the client side (mobile application) and on the server side (processing and storage of the data collected). The main issues encountered have been discussed as well as the way they had been tackled.

Chapter 5

Tests & Results

This chapter focuses on testing the whole scheme when subjected to the dynamics of what is expected to happen in the reality: input of a large amount of data on a regular basis (weekly) and the processing that is done with it.
It aims at testing the modelling tool and running with its help various scenarios to assess the limit values of crucial parameters that we should aim towards for the scheme to work properly should it be put in place for real.

As explained in the previous chapter, the limitations of GAE about quotas are annoying for running a battery of tests. It reveals more convenient and faster to create an "offline test centre" - a Java application - that replicates the functioning of the processing of the data which is made on the server on a local machine of the lab (as what we use for these tests are the CSV files released by TFL we do not need the data uploaded to the cloud by real mobile devices). This way we save time because we do not need to upload the data history on the server: we can directly parse it from the working computer; also the computer used in the lab of the DoC has way better performances than the Google server allocated with the free service of GAE (128Mb RAM, 600MHz against 16Gb RAM, 3400 MHz in the lab).

Therefore, this chapter is organised as follows: first we launch a 20 weeks run, where we alternatively input data for each week and run the ML algorithm on the data generated (data generated by our modelling tool), with the parameters set up in the previous chapter (called default parameters). This enables to test the good functioning of the system subjected to the input of huge amount of data and gives us a first overview of how efficient the whole solution is to detect, track the bikes and enable TFL intervention for repairs.
The second section focuses on playing with the values of the parameters, essentially the ones we are not sure in the model - because we do not have access to them - and for which we expect a significant influence on the results. This allows to determine the limit acceptable values that make the system unefficient.

5.1 Results of the modelling

A priori, the most crucial parameters are:

- the total number of users
- the percentage of reporting users
- the percentage of reports at stations (how many feedbacks do we get out of all the ones we ask for)

First we analyse some results by fixing these parameters to values which seem reasonable: default values.

5.1.1 Evaluation of the well-functioning of the algorithm

This first test is for the fixed parameters: 200000 users, 15 % of them are reporting users (that actively use the mobile application each time they use a damaged bike - to the point that some even report bikes not damaged), and in total we get a feedback for 75 % of bikes at stations. Getting feedback means the user labelled the bike as damaged or not damaged. If the label is that the bike is not docked or that the user is not sure (or if the user did not even take the time to complete the checking form), this does not count as a feedback (25 % remaining).

As a reminder, other fixed parameters (in the design chapter) are: 5 % of users always wrong, 10 % of users always right (experts), 50 % making 15 % errors and the remaining 35 % making 40 % errors.
Also when reporting at stations, in the case when the user makes a mistake and the bike is damaged for real, there are 80 % chances that another damage is reported against 20 % the report is that the bike is not damaged.
Finally there are 0.0003

The test is a succession of data-input and ML-run. The data corresponding to 1 week of history is parsed and both damaged bikes and reports are generated according to the model statistics, and a run of the ML algorithm is launched to classify bikes into the different categories and update the performance of the users. The ones with the highest ones are repaired or ignored according to their predicted state (damaged or not). The repairs are supposed instant as bikes might be encountered anew in the next input of data.

The four following figures represent interesting variations about some parameters across the run of 20 weeks in a row:

- Figure 5.1 is the most important, as it represents the percentage of bikes damaged (for real) that have been reported. At the end of the first week, this percentage is already fairly high (70%) and as weeks go by it quickly goes up to tend towards a value which is slightly above 95%: this means that the scheme in place enables a good detection and coverage of actually damaged bikes.
  On the same plots, we find respectively the percentage of bikes repaired (for those damaged) and the percentage of bikes ”not damaged - mistakenly reported - identified and thus that will not need to be retrieved by TFL teams and taken back to reparation warehouse.

- The second figure (Figure 5.2) shows the error rate of the algorithm for bikes classified into the category with the highest certainty. This value remains very low along the run and even tends to decrease as the algorithm sorts the users by assessing their degree of reliability for each one of them.

- Figure 5.3 represents the variation of the proportions of the two types of reported bikes: the ones that are really damaged and the others ”not damaged among all reported bikes. Overall the proportions of each type keeps constant over the time and this is perfectly logical as we changed the modelling of damaged bikes when implementing and doing the first tests (as explained in the Implementation chapter): the number of bikes getting damaged each week is in a way proportional to the number of hires there are in the week, that and also the statistical randomness introduced into the modelling part explain the slight variations we can see.

- The last figure (5.4) plots the variation of the proportions of the two types of reports: reports made after using a bike and reports made at stations. At the beginning there are more reports made after using a bike than reports made at station, however this rapidly inverses and tends to a stabilised situation. The behaviour is here also normal: as one report at station is technically the report for one bike at a station, given the number of damaged bikes to be labelled increase at first and then stabilise, the number of bikes a user may have to label at a station is likely to be superior to 1, thus for one report after use a user makes, there may be more than one to make once at the station.

The first two figures are the most important ones to look at to get an idea about how good the results are and how well the algorithm and the whole solution works, whereas the last two figures, even though it is interesting to have a look at the variations of these parameters, are more destined to check that the modelling part and the statistics set produce a model behaving in a logical and expected (to a certain point) way.

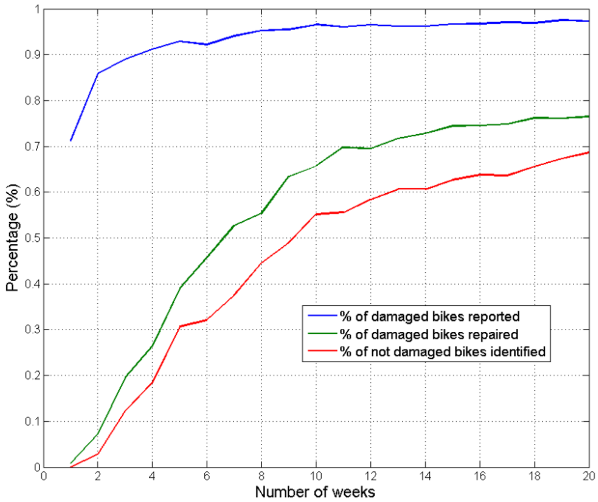


Figure 5.1: Proportions of damaged bikes covered, repaired and ignored (not damaged)

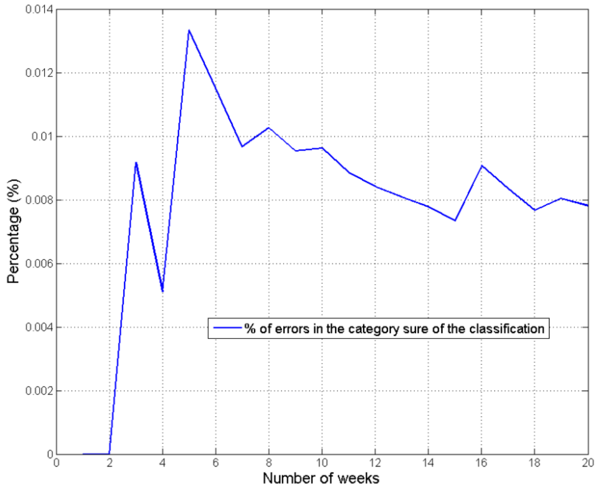


Figure 5.2: Error rate for the category with the highest certainty

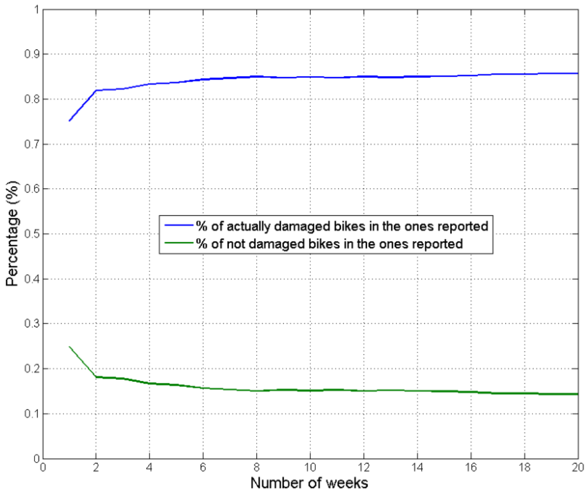


Figure 5.3: Proportions of damaged and not damaged bikes among those reported

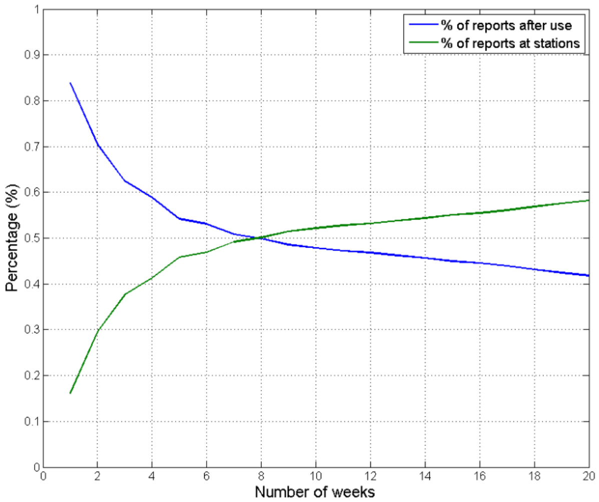


Figure 5.4: Proportions of reports at stations and reports after use

Evolution of the performance of the users

In order to evaluate how well the algorithm is able to determine the degree of reliability of each user and assign each of them an adequate performance value, we output a CSV file containing all the users that have been used by the algorithm at one point in the 20 weeks in a row run: we can then plot the final value of the performance for each one of them.

As we built ourselves the model, we perfectly know which user is supposed to be in which category, thus we just need to gather all users for each category and have a look at the overall trend of the performances assigned for each user in the category.
The figures of the corresponding plots are Figures 5.5 to 5.8, each one represents a category:

- Figure 5.5 is for the category where users are always wrong: as expected a large majority of users have a performance value of 1 which is the lowest value possible - as a reminder the scale goes from 1 to 10 and initially a user starts with a performance of 5.
- Figure 5.6 is for the one where users are always right and in this case, we notice that a large majority of the users get a performance of 10 (the maximum possible).
- Figures 5.7 and 5.8 are respectively for the users of the category making 15% and 40% of errors while labelling. In this case when having a look at the plot, it is far less obvious to realise if the algorithm performed well on average for the users of these categories.

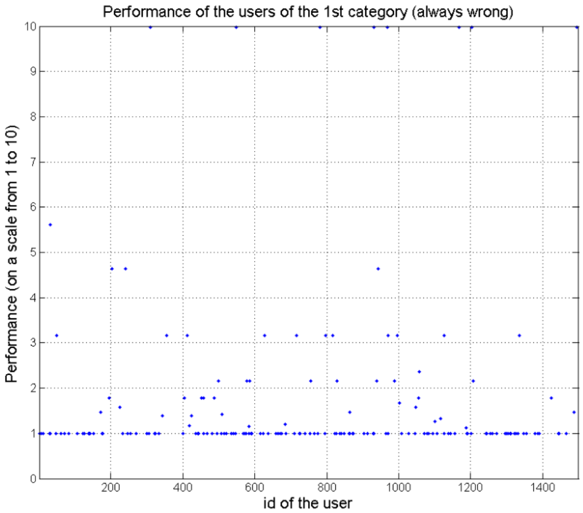


Figure 5.5:
Final performance assigned to users always wrong

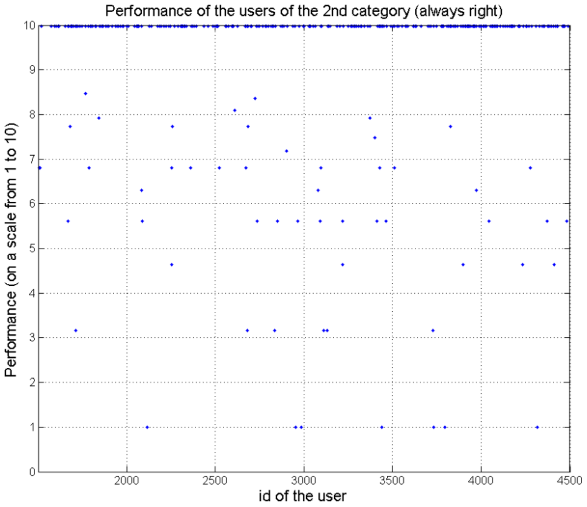


Figure 5.6:
Final performance assigned to users always right

For this reason, since the CSV file was loaded on Matlab and that we have some figures associated with the run (total number of users used, total number of users, total number of reporting users and the different theoretical percentages concerning the distinct categories), we can calculate some interesting figures from the output data.

In the first table - - we get the average percentage of good answers given by users in the category and compare it to the theory with the calculation of the absolute error. As we were already expecting it from the previous plots, the users from categories 1 and 2 have an average percentage of right answers consistent with the theory, though the one for the category of users always wrong is quite far from being 0 (17.63 %). Besides, we get the values for categories 3 and 4 and surprisingly

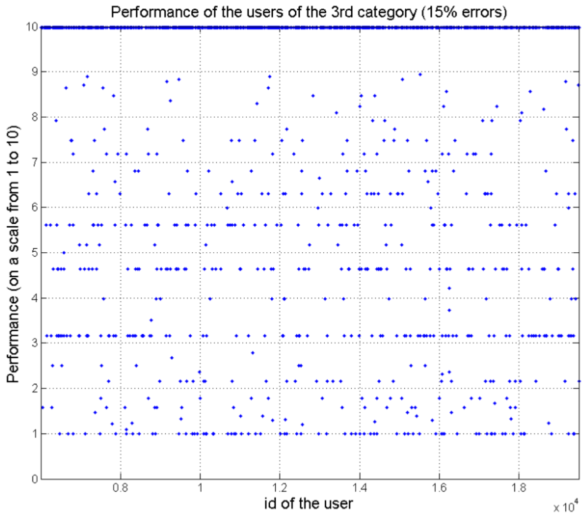


Figure 5.7: Final performance assigned to users making 15% errors

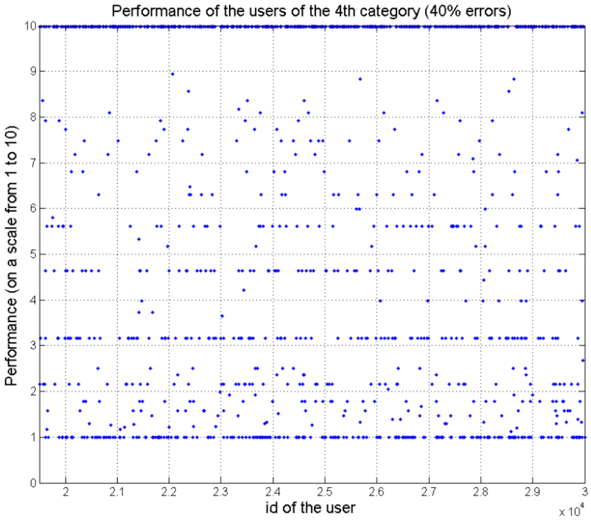
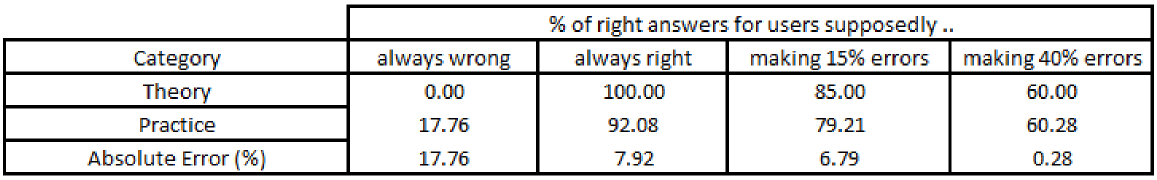


Figure 5.8: Final performance assigned to users making 40% errors

enough considering the corresponding graphs, on average the percentage of right answers is really close to the theoretical value for both these categories.

Therefore, on the long term the algorithm performs well when it comes to identify the degree of reliability of a user and consequently considers to what degree his work is likely to be true.



	% of right answers for users supposedly ..			
Category	always wrong	always right	making 15% errors	making 40% errors
Theory	0.00	100.00	85.00	60.00
Practice	17.76	92.08	79.21	60.28
Absolute Error (%)	17.76	7.92	6.79	0.28

Figure 5.9: Average percentage of right answers for each category

On the second table, which is Figure 5.10, we have a closer look on how many users have been taken into consideration at least once by the algorithm in each category across the 20 weeks run. In the end, we check that the ratio of users used in each category is the same (to 1% or 2% difference) than the ratio of reporting users used in total (3494 out of the $0.15 * 200000 = 30000$ reporting users there are in total).

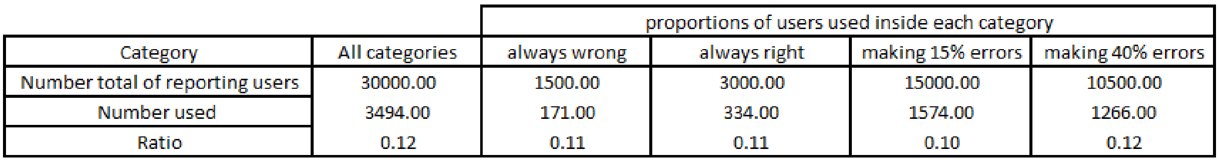
The ratio in itself (10 to 12 % of users used) is low considering that the run is on 20 weeks. On the other hand the fact that it remains constant from one category to another and all categories confounded proves that the modelling statistics works fine for picking at random users.

The third table, on Figure 5.11, is another way to check that the modelling statistics are respected as the final proportions of categories among all the users used is close to its theoretical value.

5.2 Influence of some parameters

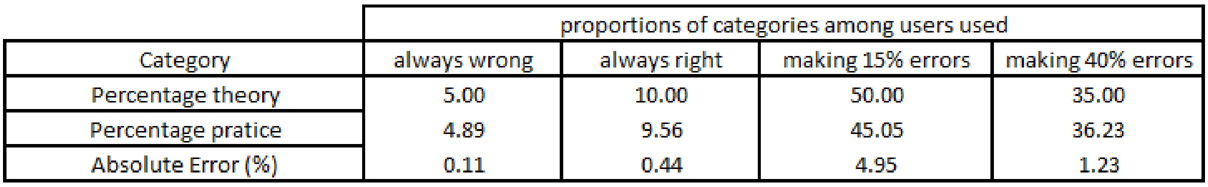
5.2.1 Influence of the variation of some key parameters

In this section, we analyse the impact of the varying the crucial parameters stated previously.



		proportions of users used inside each category				
Category	All categories	always wrong	always right	making 15% errors	making 40% errors
Number total of reporting users	30000.00	1500.00	3000.00	15000.00	10500.00
Number used	3494.00	171.00	334.00	1574.00	1266.00
Ratio	0.12	0.11	0.11	0.10	0.12

Figure 5.10: Proportions of users used in each category and for all of them



	proportions of categories among users used			
Category	always wrong	always right	making 15% errors	making 40% errors
Percentage theory	5.00	10.00	50.00	35.00
Percentage pratice	4.89	9.56	45.05	36.23
Absolute Error (%)	0.11	0.44	4.95	1.23

Figure 5.11: Proportions of categories for all users used

Influence of the percentage of the number of users

The modelling about the number and the type of users and the multiple issues raised were discussed in the design chapter. In the previous section, the run made was for 200000 users, which is a little low as we said in the design chapter that given the bar chart of TFL (Figure 3.7).

In the following plot (Figure 5.12), we represent the percentage of damaged bikes covered by the reports, it is the same curve than the blue one in Figure 5.1 except that we added the variation of the parameter 'number of users'.
The plot is thus a surface with on the X axis the number of weeks, and on the Y axis the number of users going from 100000 to 600000 users. Percentages of reporting users and reports at stations are still fixed (15 % and 75 % respectively).

The first remark we can make is that the coverage of damaged bikes does not seem to depend on the number of users there are in the model. Initially I had plotted the other two curves that are in Figure 5.1 (percentage of bikes repaired and percentage of bikes 'not damaged' identified), but these do not show any more information than the one we just deduced from this curve: the number of users does not have a consequence on the speed with which damaged bikes are spotted and reported to the system, nor on the speed with which they are either repaired or ignored if not damaged.

This is in some way a certain relief: since the number of users does not have as much consequences as we could reasonably expect, a better modelling of the users (essentially their type: modelling the fact there are members - and even sub categories of members - and casual hires as tourists would do) does not seem the most urgent feature to implement.
However, the scientific rigor would impose to do it just to ensure that this parameter is not crucial.

Influence of the percentage of reporting users

Let us vary now the percentage of reporting users. We set back the number of users to 200000 and keep the percentage of reports at stations at 75 %. The percentage of reporting users takes the following values (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 25, 30 and 35). When plotting the figures for the first time, it appeared necessary to set a lower step for the lowest values given the higher slope

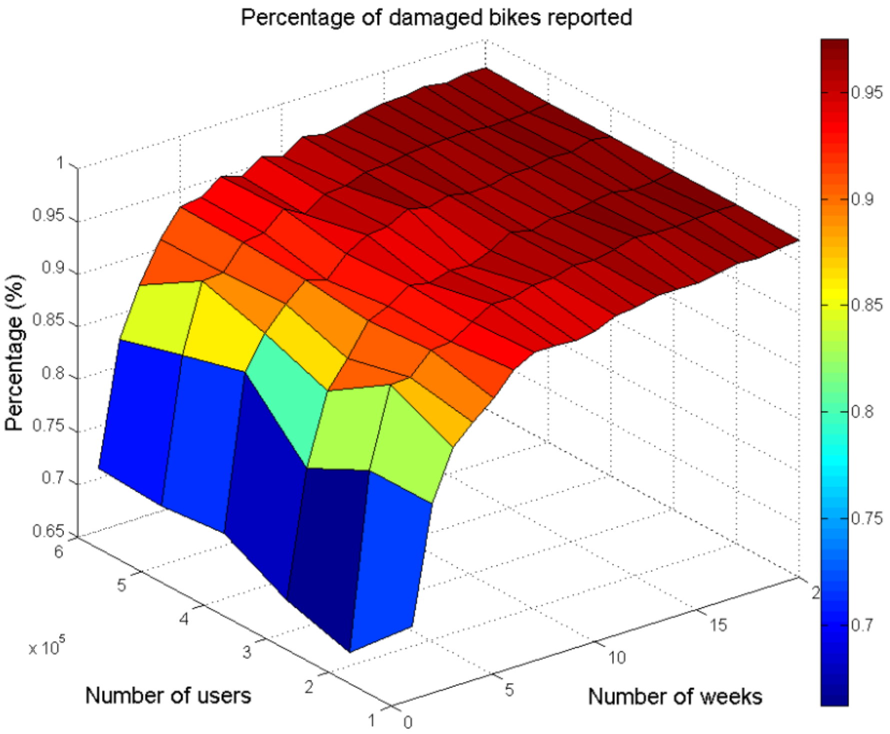


Figure 5.12: Influence across time of the number of users on the coverage of damaged bikes

for those ones.
On the other side, it seems unnecessary to go above 35 % of reporting users as this is already a value which might be very hard to reach in practice - unless taskers have something to gain from doing it. There is no sense in testing 0

Figure 5.13 is for the coverage of damaged bikes and Figures 5.14 and 5.15 are respectively for the percentage of bikes repaired and 'not damaged' identified.

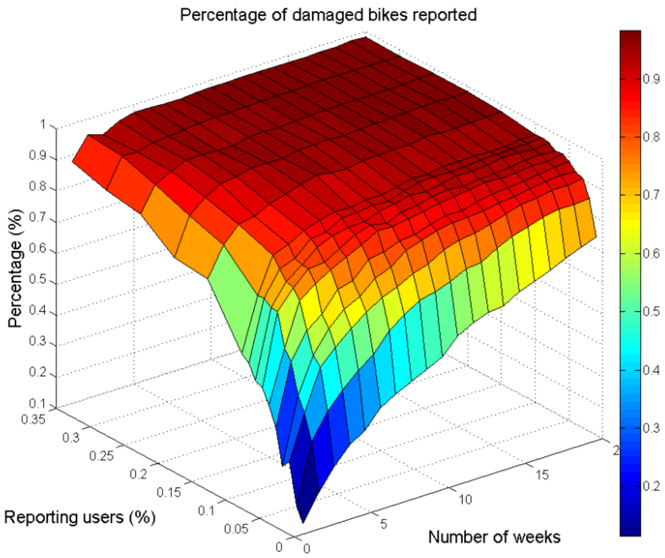


Figure 5.13: Influence across time of the number of users on the coverage of damaged bikes

Having a look on the figures, we notice that this time, the parameter has a significant influence on the velocity with which the resulting curves vary.

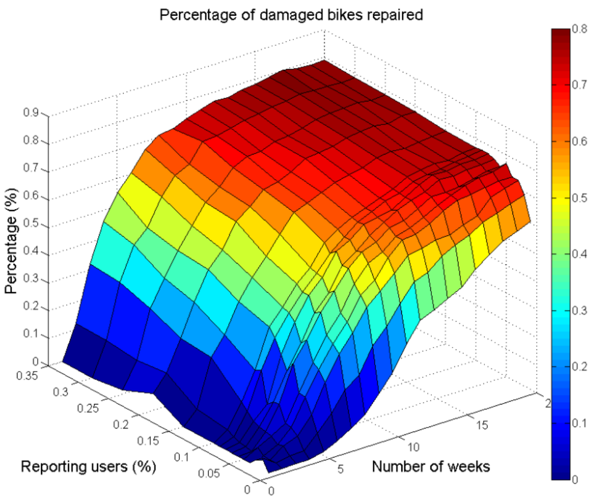


Figure 5.14: Influence across time of the number of reporting users on the percentage of bikes repaired

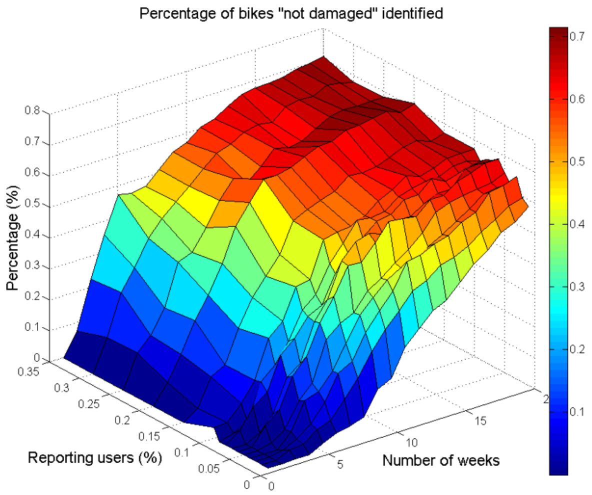


Figure 5.15: Influence across time of the number of reporting users on the percentage of 'not damaged' bikes identified

When the percentage of reporting users is down to 1 % (which is 2000 users in total), we observe the most striking case: the percentage of damaged bikes that are reported to the system has difficulties to increase and reaches a total coverage of about 70 % at the end of the 20th week when for all values of the parameter greater than 5 %, this value is reached at least at the end of the second week.
On the long run, whether the percentage of reporting users is 5 % or 35 %, we approximately reaches the same values (for the three curves we observe). What changes is the slope of the curve in the first weeks and thus the velocity with which is it reaches the final limit value (towards which it seems to tend).
This means that there is most probably more time that elapses in between the time when the bike is reported for the first time and the time when it is repaired or identified as not damaged. There is a 'latency' more or less important that depends on this percentage of reporting users: what could be interesting (but I do not know if there will be sufficient time to implement this as I write these lines) would be to analyse the average number of days elapsed in between the two moments described above and how much the parameters influence its variation: this would be a good way to measure a 'responsiveness' of the solution.

However we can still notice that values like 10 % or 15 % produce a good compromise in the sense that this latency is not too important, the final limit value (constant pace in which a constant number of newly damaged bikes are reported and approximately the same number of bikes reported a while ago are being classified for sure) is reached rapidly and reasonably reachable in practice (1 user out of 10 to 6 using the mobile app).

Influence of the percentage of reports at the stations

Let us set fixed again the percentage of reporting users to 15 % and study the influence of the variation of the percentage of reports at stations, we make its value vary from 0 % to 100 % with a step of 5 %. The number of users is still of 200000.

Once again, we notice how significantly this parameter can affect the curves. However, surprisingly enough, this only affects the curves of the percentage of repaired and 'not damaged' identified bikes. The first one which shows the evolution of the coverage of damaged bikes is not affected at all by the variation of the parameter (no latency as it could be noticed while playing with the percentage of reporting users).

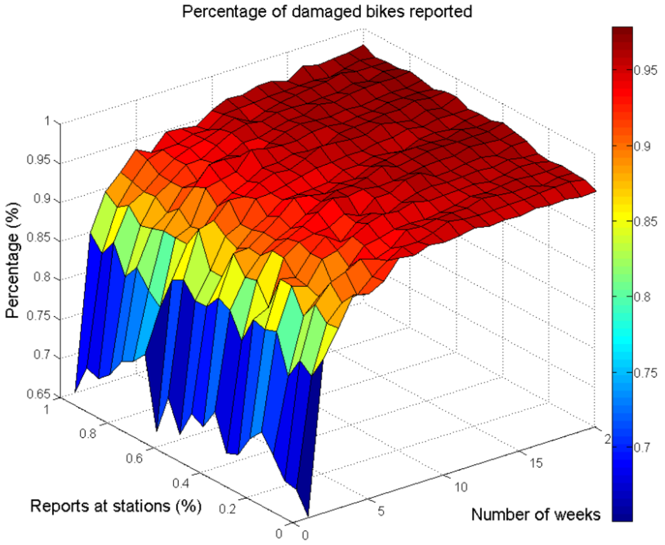


Figure 5.16: Influence across time of the reporting at stations on the coverage of damaged bikes

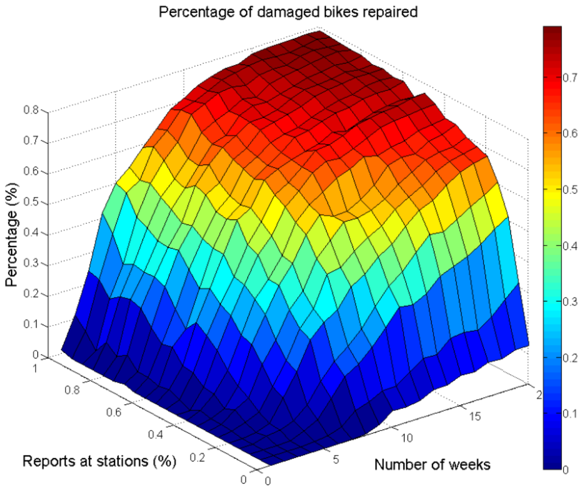


Figure 5.17: Influence across time of the reporting at stations on the percentage of repaired bikes

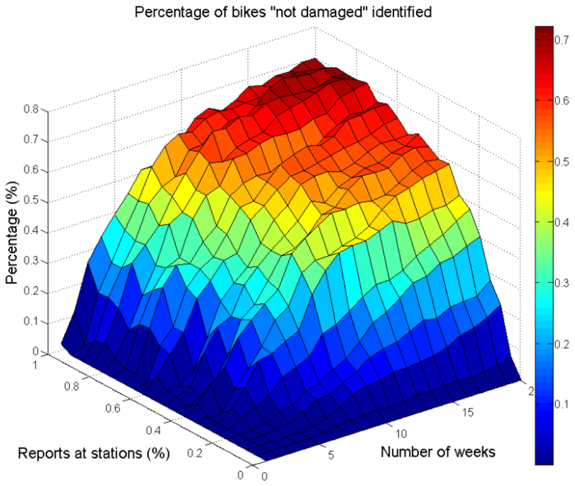


Figure 5.18: Influence across time of the reporting at stations on the percentage of 'not damaged' bikes identified

A more expected behaviour can be noted on the other two curves. The two limit case studies are:

- When the percentage of reports at stations is at 0, we observe how much the fact of reporting at stations is important for the solution to be efficient: a very low proportion of bikes actually damaged is repaired along the run, and the value barely reaches 30 % after 20 weeks. The same observation can be made for the identification of bikes not damaged.
- On the contrary, when the percentage is at 100 the system is well responsive and bikes rapidly start to be repaired or ignored according to the classification made.

In between, we can make the same remark as for the previous parameter: even though there is some obvious extra latency compared to the case when the percentage is at 100 %, some fairly low values produce amazingly reasonable results: with a percentage set at 40 to 60 % (which seems reasonably reachable in practice), the responsiveness is high enough for the system to be efficient and the limit value reached in the end is really close to the optimal case.

However, once again being able to measure this responsiveness would be better.

Influence of the scale of performance for the users

Setting back the value of reports at stations to 75 %, we take a moment to check that the width of the scale used for evaluating the performance of users do not have an influence on our results. On Figure 5.19, we vary the maximum limit of the scale (that was initially going from 1 to 10) to 20 and 30.

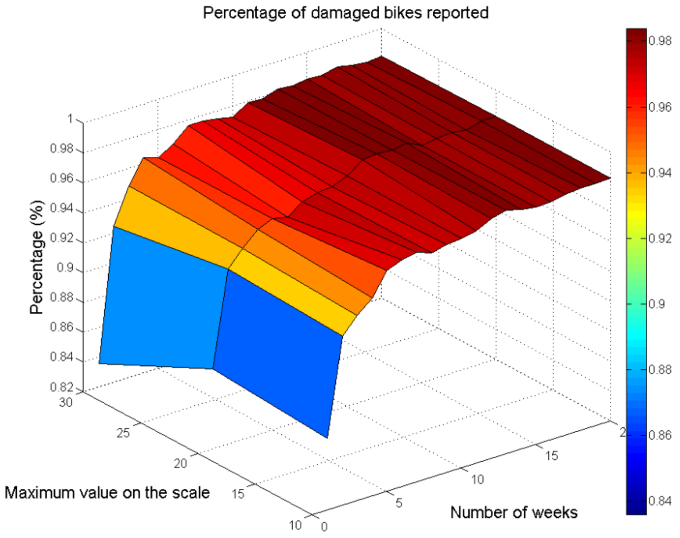


Figure 5.19: Influence across time of the reporting at stations on the coverage of damaged bikes

As expected, the variation of this parameter does not impact the results but since the value of 10 had been chosen 'empirically', we ought to check if varying its value had consequences.

Influence of number minimum of reports by user

As explained in the Implementation chapter, the number minimum of reports by user for the user to be taken into account in the ML algorithm had been set to 4. Empirically, we observed that this value produced good results in the sense that the error rate was low and the coverage of damaged bikes was among the highest.
Setting all previous parameters to the default value, we make this number vary from 1 to 10 to have a look at its influence on the results.

As previously, the first figure (Figure 5.20) shows the coverage of damaged bikes thanks to the whole scheme (how many are reported) while the two following (Figures 5.21 and 5.22) respectively represent the percentage of damaged bikes repaired and not damaged bikes identified.

This number does not seem to have a significant influence on the percentage of bikes damaged that are reported (coverage), however it does have one on both other variations.
The percentages of damaged bikes repaired and not damaged bikes identified both gradually decrease (at an equivalent moment in the run: for the same week) as the minimum number of reports by user increase: this is an expected behaviour because this number influences on the number of users that are finally taken into consideration in the ML run, thus on the number of bikes since

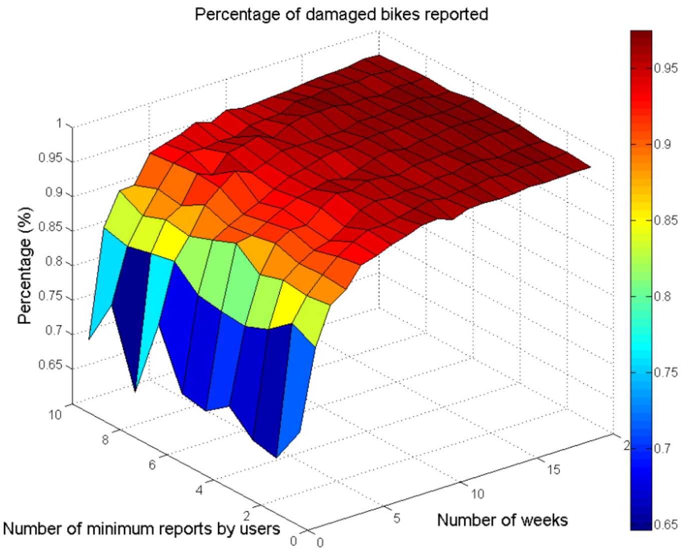
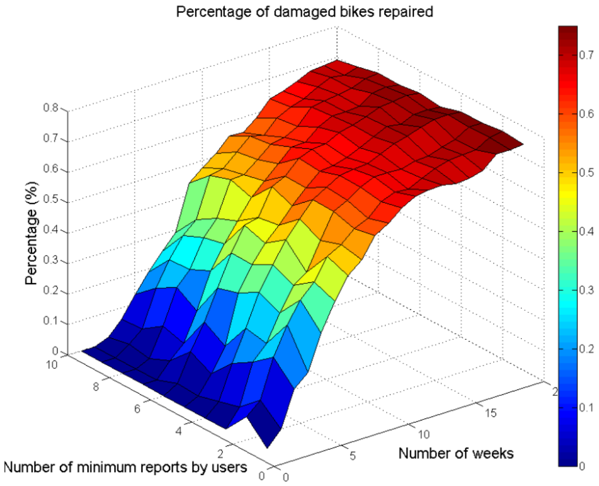


Figure 5.20: Influence across time of the minimum number of reports by user the coverage of damaged bikes



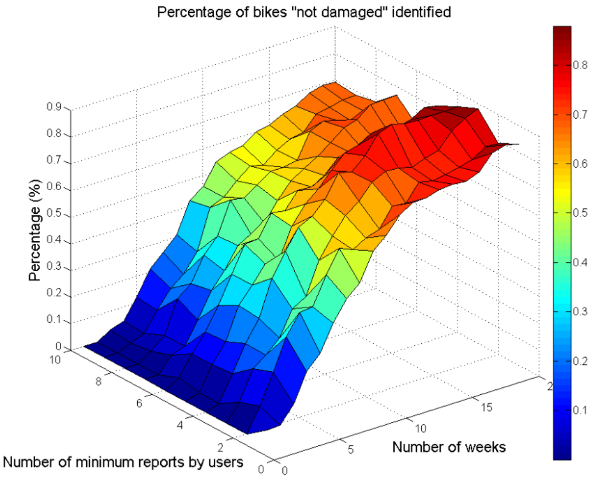


Figure 5.21: Influence across time of the minimum number of reports by user the percentage of repaired bikes

Figure 5.22: Influence across time of the minimum number of reports by user the percentage of 'not damaged' bikes identified

only the bikes associated with these users are considered.

A priori keeping this number as low as possible (the minimum being 1) would then be a good option, however we need to allow for the variation of the error rate of the classification in the category with the highest certainty. This is showed on Figure 5.23.

We can observe on this figure that the error rate is higher for the lowest values of this number. This is due to the fact that by lowering the number of reports by user, we allow the labels matrix to be proportionally more empty (the number of 0 by column increases) and this results in poorer results at the end of the run.

Considering the trend of all the previously examined figures, the value of 4 or 5 seems a good compromise: it enables us to keep one of the highest percentage of efficiency in the repairs and identification of not damaged bikes and also permits a low error rate.

Nevertheless, the operator of the server is left the freedom of adapting the value of this parameter to his needs, hence if TFL teams do not have enough bikes to collect a day, by decreasing this

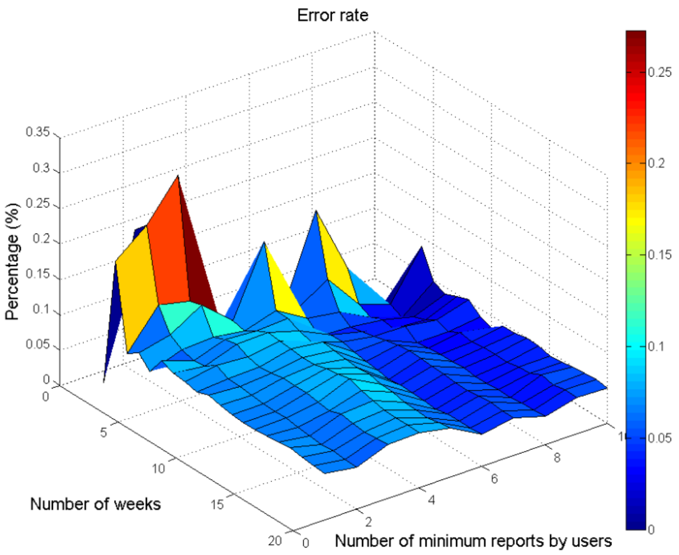


Figure 5.23: Influence across time of the minimum number of reports by user on the error rate

number they take the risk to collect more ”mistakenly classified bikes but also increase the number of picked bikes. On the contrary when there are too many to collect, they can decide to increase the number to output only the bikes classified with the highest certainty of prediction, and thus save time by not collecting misclassified bikes.

5.2.2 Which parameter prioritize

If we come to think again about the problematic of motivating the users to do the task more seriously, and considering the analysis we just made on the influence of some parameters on the correctness of the results, one of the purpose of TFL would be to play with these parameters, or rather decide which parameter improve in priority for having a greater impact on the accuracy.

The following figures (Figures 5.25 and **??**) respectively represent the percentage of damaged bikes repaired and not damaged bikes identified for different scenarios when varying both the percentage of reporting users and the percentage of reports at stations. The purpose is to determine whether having a better percentage of reporting users with a low percentage of reports at stations produces more accurate results than the opposite situation.

The figure representing the coverage of damaged bikes is not in this report because the curves are approximately the same for the diverse scenarios.
What we can notice from the other figures is that a scenario with 12.5 % of reporting users and 95 % of reports at stations performs better than a scenario where we have 35 % of reporting users and 50 % of reports at stations - damaged bikes repaired follow a similar variation but the first scenario is slightly better as for the identification of not damaged bikes.
As we can imagine, it is far more complicated to reach 35 % of reporting users (what is more 35 % of reporting users serious enough so that the final percentage of reports at stations is of 50 % as it is expected that the more reporting users there are the smaller is the proportion of them inclined to answer back correctly to the push notifications) than having 12.5 % of them. The difficulty in the last case is getting the percentage of reports at stations up to 95 %. This can be done by recompensing the users with something that motivates them to continue reporting (as explained in the background chapter, it can be money or privileged tariffs on the hire of bikes).

Reaching a reasonable proportion of reporting users and working at increasing the proportion

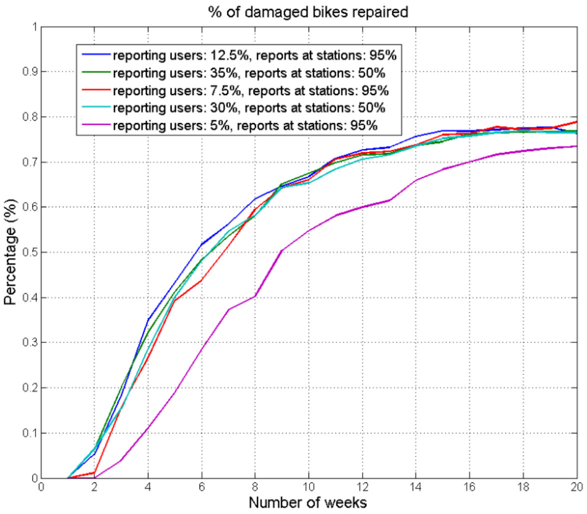


Figure 5.24: Percentage of repaired bikes for different scenarios

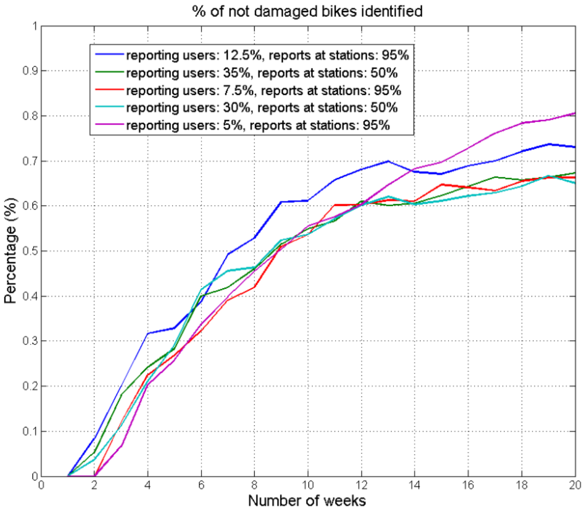


Figure 5.25: Percentage of 'not damaged' bikes identified for different scenarios

of reports at stations seems more profitable for TFL than working at increasing the number of reporting users and taking the risk that they do not participate to the checking tasks at stations, all the more so as we saw with Figure 5.4 the reports at stations are of more importance as time goes by.

5.3 Summary

In this chapter, we subjected the system to huge amount of data generated by our modelling tool and analysed the results. We varied the key parameters of the model to study how influent they are on the results.

Chapter 6

Evaluation

This chapter comments upon all the work that has been undertaken in this project in order to provide an objective evaluation of the final product. The evaluation is based on the project initial goals that were set out to address.
It at debating on the good points that altogether form the advantages of the system, but also listing the points that could be improved.
The first section focuses on the work done on the client side: the mobile application while the second part deals with what has been produced on the server side.
All along the chapter, when the opportunity presents, some ideas for future improvement are given.

6.1 Client side

On the client side, all the functionalities initially forecast are present in the final product, concerning the plugin:

- The user can easily have an overview on his location and select the bike station where he is standing by thanks to the Google Map view. The advantage of this feature is that in the case when the geolocalisation process does not provide an accurate enough location of the user and the location on the map is not quite next to the station in question, the user can still search very intuitively on the map the given station.
  The accuracy of the geolocalisation notably depends on the degree of precision set into the settings of the device, thus is dependent on the user's way of parameterizing his device.
- Once the station is selected, the user is redirected to an activity where he needs to fill minimum information in an intuitive way about the bike(s) he wants to report. The same kind of interface is available for when he receives a push notification with IDs of the bikes to check.
  In addition to that, the user is warned clearly when he makes a mistake or intends to valid a form with incomplete information. With the same spirit, if the user forgets to allow the plugin to save the output file onto his hard drive, a clear message is displayed.

The main application that has to be launched on the device is the one implemented into the Hive framework [12]: the one that lists all the tasks available on the server, and then when clicking on one particular task, all the stages comprised in it.
The advantages of using this main application to launch the different stages are those that are emphasized in the report of D. Pavlov [12]: the registration of users on the server is made easily thanks to the GMAIL account of each user, the ID of each of them being the part before the @; the framework has already an implementation that takes care of asking the user each time the permission to access or save a file or a resource on the device of the user; besides there is also a plugin that provides the functionality of uploading the data generated by our plugin and saved on the hard drive, the upload is linked to the user and the file stored on the server is named after the

user and the ID of the task, hence it is very easy to collect and reorganise the files on the server side.
The advantage of the task being divided into several stages is that if the user loses his data connection for whatever reason, he can still stores the output of our new plugin on his phone and wait before launching the next stage that will try and upload the data on the server.

On the other hand, since there is only one task in this project that has four stages, the fact of using the framework might seem a heavy procedure in the end for the user. Each time he wants to report a bike, he will have to launch the application listing all the tasks (in theory there is only one here) and click on the one of interest before accessing to the list of tasks and being able either to launch the next stage (if he had paused) or starting again the task.
What is more, in order to be able to start again the task from the beginning the user has to press the reset button as many times as there are stages completed in the task. Therefore if he has completed the task until the end (both reported a bike and checked some more bikes following the reception of a push notification and sent the two generated files to the server) he will have to press four times the reset button to reset completely the task and be able to report a bike after using it again.

Consequently, the use of the Hive framework brings some very attractive features implemented with great skills and that would take a lot of time to re-implement only for the need of this application but in the same time, it imposes some heavy procedures that can eventually bore the user. Making all this more flexible would propose a better experience to the user.

6.2 Server side

On the server side, the main functionalities targeted in the design have been implemented:

- the files uploaded to the server and stored into the blobstore with IDs made of the IDs of the user, the task and the stage of the task are read and the data is distributed into beans (ReportingUser, ReportedBike, DataStation) adequate to the processing step taking place later.
- the ML algorithm taking as inputs shortlisted bikes and users and producing as outputs the new performance value for each user and the true damage of each bike.
  However as emphasized into the Implementation chapter, due to the high number of 0 in the labels matrix - where 0 represent the absence of annotation by the user for the corresponding bike - entails results with different certainty from the algorithm. Only the results that are assessed to be certain are taken into account.
- A web user interface enables the developer - or the person in charge of managing the server - to visualise the reported bikes and users associated.
  This interface also provides the use of different datasets mainly for tests purposes. The data in each dataset can be stored into the cloud (service provided by the GAE) so that the data can persist in time.
- Two ways of testing the system: one is purely a way to simulate the input of data from the web interface instead of using the mobile application: it enables the developer to test the functioning of the server mechanisms overall: if the data is stored and processed accordingly to the expectations and affords him to detect more easily the presence of bugs.
  The second way is more a way of testing the functioning of the whole system under a huge input of data closer to what is expected to happen in reality. This way uses the history of hires released on a monthly basis by TFL and the model designed thanks to statistics that can be set as desired - and explained in the design chapter - to simulate the rhythm with which bikes get damaged and reported, and simulate the succession of data inputs and ML

runs.
For practical reasons, the use of distinct datasets has been set up to enable the storage of data (users and bikes) from one run or scenario while working on another dataset. The dataset called "reality stores the data uploaded from mobile devices.

While the web user interface is an advantage to monitor what is going on into the server mechanisms, it is also a time consuming task to build the pages and thus an obvious future work, though relatively easy, would be to improve the presentation of the display as well as the quantity of information displayed back to the developer.
We have already discussed it a few times along the report, however it seems necessary to recall that the use of Google App Engine is subjected to some annoying limitations in the case of our project. On the one hand, it gives us a free server to deploy Java Web Applications and easily accessible from the internet, and thus from the mobile devices of our future users. What is more, it provides useful services which reveal very helpful for our needs: the Google Cloud Storage enables to store the data from our users in the cloud with the NoSQL Objectify tool which is particularly easy to apprehend; the Google Cloud Messaging for Android is an inescapable tool for sending our push notifications from the server back to the mobile devices and the Google Maps Android API v2, though optional, provides a better experience on the mobile application for the client. In addition to that, we cannot deny that the administration console developed to manage the options and settings of our GAE Application are clear enough so that a quick start is possible: the management of different versions of the application, the data stored either into the blobstore as files or in the Datastore as Java objects instances, or the generation of keys for the APIs are things greatly facilitated via the console.
However, on the other hand, we came to test the limits of the free service: the number of accesses to the Datastore is limited so that when we model the dynamics of the reality we cannot store all the data processed. In the same way, the duration of the request - that we initiate when asking to the ML algorithm to execute a new run for example - are limited to 1 min and after that an exception is raised due to the overrun of the deadline.

An interesting feature of GAE which could be a solution to this last issue would be to use the back end instances that enable a long process to run in the background without worrying about any duration deadline. Nevertheless, this solution is not assured to be working on the long run as there are a limited number of back end instance hours in the quotas fixed for each GAE application. This number amounts to 9 instance hours and the speed with which they are consumed depends on the type of performance chosen for the computation on the server: default performances of the server are 600MHz and 128Mb RAM which is very low compared to performances of DoC lab computers, and upgrading the performance up to 1200MHz and 256Mb RAM consumes instance hours twice as faster as required for the first type.
It is actually the same issue for the front end instance hours, except that the number these hours is limited to 28 instead of only 9, which provides the developer a greater freedom. This is also the reason why we run most of our tests on an offline Java application in the lab of DoC: it went faster and did not pose any quotas issues.

If we had to evaluate the degree of accuracy of the model put in place along the project to simulate reality dynamics, it would be complicated though it seems obvious that this model can be improved.
As we observed in the Tests & Results chapter, the number of users considered is not so important, however the question is worth asking whether introducing some categories of users might be of significant importance: maybe the construction of a model comprising members and casual users of the bikes and the frequency with which they respectively hire bikes (especially for members who are the most prone to become regular users of the mobile application) might change the results, the difficulty is as always about finding the figures that are not publicly released by TFL.
Along the same idea, the modelling of how bikes get damaged or get reported damaged although

they are not might be to improve.

All the projected implementations have not been finished on the server side: it was planned to build a CBR structure to help determining the time of final breakdown of a bike by comparing his sequence of ”reports - non reports to the sequences of previous cases. Although some code has been written to initiate the idea, time constraint has not allowed the final elaboration of this feature. However, in the reflexion leading to the conception of this idea, it appeared relevant to introduce a ”degree of gravity of the damage, for example from 1 to 5, that would be filled by the user when reporting or checking a bike. This might enhance the results of the similarity measure in the end and help to identify the closest cases better, but on the other hand it would also complicate the implementation of the ML algorithm and the model statistics.

Finally, a word has to be expressed about the efficiency of the ML algorithm. Though it seems to produce attractive results on the long run, this algorithm might not be perfectly suited to the huge absence of annotations as it happens in the reality. This makes it produce very large quantity of uncertain results that we cannot take as certain if we want to minimize and keep low the error rate.

Chapter 7

Conclusion

7.1 Project outcome

This project had for main goals to provide a Mobile Crowdsourcing application that would enable the detection and tracking of damaged Barclays bikes across the city of London in order to facilitate the intervention of the TFL maintenance teams. The main motivation behind the project was to afford users an easy way to report bikes to TFL when a slight damage starts to be an issue for the rider but that would not worth to lock the docking point at the end station, preventing other users from using the bike afterwards. This should save time and money to TFL teams and get a better experience from the users' point of view.

As the Design and Implementation chapters show, the main functionalities and structures that were planned have been implemented: a new plugin has been developed for the purpose of the project, it extends the Hive framework which is a useful tool for building new Mobile Crowdsourcing applications without having to start from scratch. However, as seen in the Evaluation, despite the many advantages that this tool possesses, it also comes with a few downsides. Unfortunately there was already not enough time to implement all the functionalities thought of, thus it was out of question to "waste" time re-implementing the useful features needed from the framework.

In addition to the work done on the mobile application which enables users to report bikes in an intuitive way, the server structure has also been developed to treat the received data, reorganise it, process it and display the most meaningful results to the operator via a web interface.

Altogether it results in a physical architecture that manages the tracking of bikes reported as damaged and the users that participate to the scheme. Once there are enough reports so that the ML algorithm can determine for sure the state of a bike, it can be fetched by TFL teams or ignored if it turns out that the bike was mistakenly reported as damaged. On the other hand, as time goes by, users are categorised in terms of their performance on labelling the bikes so that the labels of the most reliable annotators matter more than ones labelling - deliberately or unwillingly - poorly.

On top of that, an attempt of modelling the dynamics of the reality was developed with two main goals: testing the good functioning of the whole scheme in terms of storage and processing - this highlighted the limits of the free service that permits Google App Engine - and evaluating the interest and the expectations that one could have from such a scheme and if it would be worth deploying it for real.

As the Tests & Results chapter has proved, this modelling part brought conclusive results as for the good functioning of the structure and the potential and interest that a company as TFL could have for such a solution.

The most annoying issues taking time to resolve or to find a relevant alternative encountered along the project have been discussed in the report when the opportunity seemed right. However one needs to consider all other issues that took less time to resolve or were obvious to solve once the adequate notions learnt which, summed up, make a considerable amount of time spent on overcoming difficulties.

7.2 Future work

However, all functionalities have not been developed and among those developed, some would need improvements.

The model is subject to enhancement in terms of modelling the users, their behaviour and the rhythm with which bikes get damaged. Official figures from TFL would greatly help. An interesting feature that was not taken into account is the possibility that a reporting user decide not to report a bike slightly damaged after using it even though he notices the presence of the damage: in this case he evaluates that the damage is not significantly annoying enough to be worth a report.

On a more technical point of view, the use of Google App Engine, though extremely useful and providing services such as the Google Cloud Messaging For Android - that would be difficult to reproduce on a private server, has proved its limits: the number of accesses to the cloud storage and the number of hours a day the server can run in the background are the main obstacles to the deployment of the solution at a large scale.

The functionalities that are worth developing are the ability to predict the final breakdown of a bike - in terms of duration of use, not in terms of days, as it depends on how much the bike is used in the incoming days - by comparing the way it was reported: the succession of reports and non-reports and how many times it has not been reported for when he is reported. Logically, if the bike is actually damaged, as the damage gets more and more annoying, the bike gets more frequently reported. This has to be coupled to the introduction of a degree of gravity for the damages (1 to 5 for example).

List of Figures

Bibliography

[1] P. Ipeirotis, ”Managing Crowdsourced Human Computation”, *Slides from the WWW2011 tutorial*, March 2011.

[2] Matthew Lease, ”On Quality Control and Machine Learning in Crowdsourcing”.

[3] L. Dickens and E. Lupu, ”On Efficient Meta-Data Collection for Crowdsensing”.

[4] V. C. Raykar, S. Yu, L. H. Zhao, A. Jerebko, C. Florin, G. H. Valadez, L. Bogoni, and L. Moy, ”Learning From Crowds”, *Journal of Machine Learning Research 11 (2010) 1297-1322*, 2010.

[5] V. C. Raykar, S. Yu, L. H. Zhao, A. Jerebko, C. Florin, G. H. Valadez, L. Bogoni, and L. Moy, ”Supervised Learning from Multiple Experts: Whom to Trust when Everyone Lies a Bit”, *Proc. of 26th Intl. Conf. on Machine Learning*, 2009.

[6] Jeffrey M. Rzeszotarski and Aniket Kittur, ”Instrumenting the Crowd: Using Implicit Behavioral Measures to Predict Task Performance”.

[7] Burr Settles, ”Active Learning Literature Survey”, *Computer Sciences Technical Report 1648*, 2009.

[8] Peter Welinder and Pietro Perona, ”Online crowdsourcing: rating annotators and obtaining cost-effective labels”.

[9] Peter Welinder, Steve Branson, Serge Belongie and Pietro Perona, ”The MultidimensionalWisdom of Crowds”.

[10] Alexander J. Quinn and Benjamin B. Bederson, ”Human Computation: A Survey and Taxonomy of a Growing Field”.

[11] Jacob Whitehill, Paul Ruvolo, Tingfan Wu, Jacob Bergsma, and Javier Movellan, ”Whose Vote Should Count More: Optimal Integration of Labels from Labelers of Unknown Expertise”.

[12] D. Valentinov, ”Hive: An Extensible and Scalable Framework For Mobile Crowdsourcing”, 2013.

[13] von Ahn, L. and Dabbish, L., ”Labeling images with a computer game”, *Proc CHI*, 2004.

[14] von Ahn, L., Maurer B., McMillen, C., Abraham, D. and Blum, M. ”ReCAPTACHA: human-based character recognition via web security measures”, *Science*, 2008.

[15] M. Bernstein, R. C. Miller, G. Little, M. Ackerman, B. Hartmann, D.R. Karger, and K.S. Panovich, ”A Word Processor with a Crowd Inside”, *Proc. UIST 2010.*

[16] A. Dawid and A. Skene, ”Maximum likelihood estimation of observer error-rates using the em algorithm”, *Applied Statistics*, 1979.

[17] C. Liu and Y.M. Wang, ”TrueLabel + Confusions: A Spectrum of Probabilistic Models in Analyzing Multiple Ratings”.

[18] Wikipedia website, ”Barclays Cycle Hire Scheme”, URL: http://en.wikipedia.org/wiki/Barclays_Cycle_Hire, accessed: August 2014.

[19] Wikipedia website, ”Exponential distribution”, URL: http://en.wikipedia.org/wiki/Exponential_distribution, accessed: August 2014.

[20] Sutton, Mark, (2011), ”London bike hire faring better than Paris scheme for write offs”, Bike Biz Magazine, 22 February 2011.

[21] Transport For London website, ”Barclays Cycle Hire Scheme — Transport For London — Interactive Map”, URL: https://web.barclayscyclehire.tfl.gov.uk/maps, accessed: August 2014.

[22] Transport For London website, ”Barclays Cycle Hire performance”, URL: https://www.tfl.gov.uk/corporate/publications-and-reports/barclays-cycle-hire-performance, accessed: August 2014.

[23] IBug website, departement of computing, ICL, ”Machine Learning Courses : Evaluating Hypotheses”, URL: http://ibug.doc.ic.ac.uk/media/uploads/documents/courses/, accessed: August 2014.

[24] Objectify documentation, ”objectify-appengine”, URL: https://code.google.com/p/objectify-appengine/wiki/Concepts?wl=en, accessed: August 2014.

[25] Google App Engine website, ”Google App Engine: Platform as a Service”, URL: https://developers.google.com/appengine/, accessed: August 2014.

[26] Google Maps Android API v2 website and documentation, URL: https://developers.google.com/maps/documentation/android/?hl=fr, accessed: August 2014.

[27] Bootstrap documentation, URL: http://getbootstrap.com/, accessed: August 2014.