Software Engineering — University of Amsterdam

Master Software Engineering

University of Amsterdam

Master thesis Software Engineering

Noise detection in software engineering datasets using Gaussian Processes

Adrian - Vlad Lep
adrian-vlad.lep@student.uva.nl
Student ID 6485839

December 5, 2013

Supervisors:

Jurgen Vinju
CWI: SWAT
Jurgen.Vinju@cwi.nl

Magiel Bruntink
UvA
M.Bruntink@uva.nl

Contents

CHAPTER 1

Introduction

It is widely known and accepted that most of the software costs represent maintenance and evolution activities [7]. Therefore significant efforts are being made in developing techniques for prediction, risk management and quality evaluation of software projects. Software metrics are "quantitative approaches to understand, manage and improve software engineering" [15]. They are the foundation for techniques and research done in the field of software engineering [13, 14, 8, 21, 18].

Measurements of software project properties can be easily obtained these days due to the the growth of open source. Large datasets of software metrics can be generated by mining existing software repositories and foundations such as GitHub, SourceForge, Google Code, Apache or directly downloaded from platforms such as Ohloh.

Surprisingly, the quality of these datasets is not a common concern and a systematic review of the literature performed by Liebchen [9] retrieved 161 papers that address data quality in the empirical Software Engineering field since 1993. This is a small number compared to the number of published papers, which is in the order of thousands. Secondly, the review pointed out that it was a "minority practice to even explicitly discuss data quality" [15] although "a substantial majority of papers, 138 out of 161, considered data quality to be a threat to analysis of empirical data" [9].

Data quality is clearly an important factor to consider in the empirical Software Engineering and in the recent two to five years more efforts are made to improve it. According to De Vaux and Hand [22], between 60 to 95% of the time needed for the analysis is spent on cleaning the data.

Our research focuses on how to assess and improve the quality of large software engineering datasets. We found three challenges to this:

1. ***Data quality in general***

   The problem of data quality is challenging by itself as it is extremely difficult to know, in general, the "true" value of a data item [10]. Furthermore, similar to software where we can not guarantee the absence of bugs even after testing, we can not guarantee the correctness of datasets even after using cleaning techniques.

2. ***Large datasets***

   For large datasets, manual inspection is not a feasible solution. Therefore in our research we aim to use automated approaches that can improve the quality of these datasets. Using techniques from the Machine Learning field we aim to automatically find abnormalities in big datasets of software metrics obtained from Ohloh.

3. ***Modeling real world datasets***

   To test an approach, it is common practice to artificially induce noise and try to detect it. This has the shortcoming that the noise induced might not represent real noise and that the data could already have noise in it.

   We used, for our validation, real world data mined from Ohloh [6] that contains information for different types of projects, from smaller ones that started only a few months ago, to bigger ones that started years ago (e.g. Mozila Firefox). The task of assessing the quality has added complexity because of the dataset itself which is unknown to us and very heterogeneous. Projects have a diverse evolution in time and are created using different programming languages. More about the Ohloh datasets can be found in Chapter 5.

Different automated approaches were tried in research for noise or outlier detection. To address our challenges, we experimented with the Gaussian Processes technique, a method known in Machine Learning that creates flexible models. Gaussian Processes define a probability distribution over non-linear functions and the model they create could be viewed as a non-linear mean function that has an adaptable Gaussian variance around it. The main contributions of this thesis are:

- **Evaluate the Gaussian Processes technique for noise detection.** From our knowledge, Gaussian processes were not used until now for noise detection in the field of empirical Software Engineering, according to the literature review of Liebchen [9] or in the Artificial Intelligence field according to the survey of Hodge and Austin [46]. In this thesis, we show a method to use these techniques for noise detection and evaluate our approach on a data with artificially induced noise and on two large, real world datasets obtained from Ohloh

- **Asses the quality of the Ohloh datasets.** The data obtained from Ohloh was unknown to us and we had no prior labels about the quality of the measurements. We used Gaussian Processes to assess the quality of the data. Our approach highlighted different interesting points and several errors in the dataset were found and mentioned in this paper.

In the following chapter we outline the literature found on data quality and how Machine Learning techniques were used to assess and improve it. Chapter 3 describes approaches to noise detection and especially Gaussian Processes. Chapter 4 presents an experiment done in controlled settings with simulated noise, to assess the capabilities of our approach, while Chapter 5 presents the tests performed on the Ohloh data. The conclusions are in the last chapter, while the Appendix shortly describes the source code.

CHAPTER 2

Background

In this chapter we present related literature found on data quality and how Machine Learning was used to improve it. The concepts explained are necessary for appreciating the rest of the thesis.

"The decisions made are only as good as the data upon upon which they are based" [26]

There are many definitions of data quality but the most widely accepted one describes it as "fitness for purpose" [10]. This implies that in order to consider the quality of a dataset, we need to understand what will it be used for. Many datasets are used for research in empirical Software Engineering, for diverse purpose as deciding the quality, predicting the evolution, effort, costs, bugs etc. We believe that the research done in this field stands on the data it uses and "great care is needed to ensure sufficient attention is paid to the data as well as the algorithms" [17].

Data quality is seen in literature as a multidimensional concept on which researchers do not have a common agreement [32]. Wand and Wang enumerated four dimensions that are repeatedly mentioned in literature [32] :

- accuracy
- completeness
- consistency
- timeliness

The timeliness dimension raises the issue that historical data might not represent current situations. For example productivity changes over time according to Shepperd [30]. Therefore, using old data to make predictions for current productivity could create bad results. Completeness, is easy to evaluate even on large

datasets. By creating a script missing measurement can be fast identified. A consistent dataset presumes that a "data value can only be expected to be the same for the same situation"[32].

Accuracy, has no clear definition, incorporating precision of measurements in some cases. We are not interested in precision, as for a given dataset, we can not increase the precision now; this being an early technique to increase data quality. We will follow the definition of Wand and Wang [32] which propose that "inaccuracy implies that information systems represent a real-world state different from the one that should have been represented". This is much harder to assure as distortions or noise are not easy to discover in large datasets.

Noise, is defined as bad data, an incorrect instance in our dataset. It could have different reasons for appearing: bad data collection, errors in manipulating it or errors in interpreting it. It is certain, that we want to remove noisy instances from our dataset as they can impact our research. This can be achieved by preventive techniques or correcting techniques. In our research, we will focus on the latter.

Outliers are highly atypical values, that appear in rare circumstances. They can also be affected by noise. E.g. If we keep track of activities done each day by people in the USA, "landing on the moon" might be correct for Louis Amstrong but it is certainly not a common activity, yet. Therefore, Louis Amstrong's "landing on the moon" activity would be an outlier in our dataset. On the other hand, if "landing on the moon" is reported for us, as our activity for today, it will be most likely noise.

In Machine Learning some researchers considered outliers to be noise [10] since they can create difficulties in research. We consider that depending on the question we want to answer we should exclude outliers from our data or not. This relates to the definition of quality as "fitness for purpose".

2.1 Noise detection using Machine Learning

Because of its great impact, data quality is also a concern in other fields and automatic ways to identify noise were tried out using Machine Learning techniques. One broad definition of Machine Learning is the study of algorithms that improves automatically through experience [3]. The typical approach for noise detection is to learn some classifier based on the data, predict the labels of the points and consider wrongly classified points as suspect. In this section, we will present some automated techniques, used by researchers in different fields for noise identification.

In medicine, many diagnostics are made based on data, making the quality an important factor. Different techniques have been investigated to detect noise in data and correct it. Gamberger uses compression based induction to handle noise [11]. Their idea is founded on the MDL (Minimum Description Length), which proposes that the best hypothesis for a given dataset is the one that lead to the best compression of the data. By cleaning the dataset in this way, they

managed to improve the classification for early diagnosis of rheumatic diseases. Similar techniques have been tried with success by Gamberger and colleagues for detecting coronary artery disease in [12].

MLD is a formalization of Occam's Razor, which states that among all models that correctly fit the training data, we should select the simplest. In the same category of algorithms decision trees were built in [19] which were simplified by pruning techniques. PCA (Principle component analysis), a dimensionality reduction algorithm could also be used.

Brodley and Friedl [23] focus on datasets used for automated land-cover mapping from satellite data, credit approval, scene segmentation, road segmentation and fire danger prediction. They used a different approach, by splitting their dataset into n parts. They created n models, with $n-1$ segments of data, and used the remaining piece for testing. Contrary to the previously described algorithms, in this case, the training and testing data are kept separately; we learn the model on one part and we make our prediction on the remaining. Moreover, in [23], they introduced the idea of multiple base classifiers. Basically they used decision trees, nearest neighbor and a linear machine algorithm as base classifiers to separately model the data and make classification. Afterwards, each classifier votes for each point, if it is noisy or not.

Finally, another class of techniques to deal with noise, is data polishing [9]. These methods detect noise but instead of removing it, they correct the erroneous values. For small collections, removing noise from the dataset decreases the number of instances, making polishing an interesting technique. We do not consider this is of interest in the case of large datasets. In this case it is better to remove noise, not to create artificial points in our data.

In this thesis, in order to detect noise we use Gaussian Processes, a probabilistic approach. They are based on different mathematical concepts than the previously mentioned techniques. Also, the problem that we approach on the Ohloh dataset, is different from the aforementioned ones since we do not have labeled data or a secondary framework that we could test our resultson; as Gamberger had a tool that predicted coronary artery disease. They improved its performance by cleaning the data. Nevertheless, we can notice some similarities to the other techniques. From some perspective, the way we use Gaussian processes, could be seen similar to a data compression technique since we learn a model that approximates the data. Second, we can keep training and test data separately and we can generate more models on different parts of the data. Last, since Gaussian Processes make predictions, we can polish our data by replacing the measured values with our prediction for the instances we consider noise.

2.2 Data quality addresses in Software Engineering

In empirical Software Engineering the main efforts are done by Khoshgoftaar and colleagues [24, 25, 26], Liebchen and Shepperd [10, 9, 20]. Liebchen in his thesis [9] researched how empirical analysts address the problem of data quality. He found that out of 161 studies:

1. **50 studies** (31%) Avoid poor data by improving the *data collection* procedures. This is a noise prevention technique, which is not always possible since many times we analyse data sets that were collected somewhere in the past.

2. **35 studies** (22%) *Manually check* the quality of the data. This is done by inspecting the measurements, measuring the same attributes in different ways and comparing the results, visual inspection.

3. **18 studies** (11%) *Use meta-data*, for example to describe the perceived level of quality for each attribute or instance. This is commonly used by in the ISBSG benchmark data sets, which assign grades from A(highest quality) to D(lowest qulity) to their measurements. [27]. Their grades describe the completeness of the data, having no missing values, which is not equal to ensuring quality.

4. **16 studies** (10%) Use *automated noise detection techniques* to improve their data quality

5. **21 papers** (13%) Carried out an *empirical analysis of noise.*

A comprehensive list with descriptions of used algorithms is found in Chapter 2 and 3 of Liebchen's PHD thesis [9]. We will briefly summarize some of the used techniques in this section.

Khoshgoftaar propose noise detection techniques based on boolean rules generated from the data [24]. They test their research by artificially injecting noise in NASA datasets, obtaining good results.

Secondly, they proposes the Pairwise Attribute Noise Detection Algorithm or PANDA. PANDA analyses each two attribute pairs from the dataset and calculates the mean and variance base on all the records for these two attributes. Then, for each instance, a score is assigned that describes its probability to be noise. Large deviations from normal, have higher contributions to the score. The final score for each instance is computed based on all the scores obtained for each combination of two attributes. The authors showed that PANDA found more noisy instances, fewer outliers and fewer clean instances than the nearest neighbor outlier detection technique.

Liebchen used in his thesis [9] three types of decision tree methods to handle noise: Robust Filtering, Predictive Filtering and Filter and Polish. He validates his approach on real data sets, using domain experts to decide on the correctness

of their classification. Their results are good, but when testing the algorithm on simulated data the results are less promising. This shows how hard it is to create a solution that can handle different types of noise, in different domains.

More efforts to maintain high quality datasets in the empirical software engineering field have been made by the PROMISE Group which currently has 20 data sets.

Most proposed techniques are evaluated by injecting artificial noise in datasets. The first limitation of this is that we do not know if the data that we use and inject noise into, does not have noise already. Second, the noise is usually added randomly with a distribution, but this might not accurately simulate real world noise.

This thesis examines the use of Gaussian Processes for noise detection on software engineering datasets. We test our approach on simulated noisy datasets but also on real world data with noise. We believe Gaussian Processes could perform well in detecting noise, since they create flexible models and are non-parametric approaches. Therefore, we do not need to set ourselves parameters that describe our data, they are learned in the training phase. This is a big advantage for datasets of which we have no prior knowledge. From what we know, Gaussian Processes have not been used until now for noise detection, this implies a beginning risk for us. Also their computational complexity is $O(^3)$, which could create problems for our big datasets from Ohloh.

CHAPTER 3

Approaches to noise detection

We can see the task of detecting noise in two different ways, depending on the data we have.

1. If we have pre-labeled data, which describes in advance whether each instance is noise or not, we can treat our problem as a *classification task*. We train our classifier and predict the class for each instance. Different options can be used for the classifiers, e.g. Support Vector Machines(SVM), Neural Networks, Decision Trees, K Nearest Neighbor, etc.

2. If we do not have prior knowledge of the data, no labels, we can not use a classifier. In this situation the approaches are similar to *unsupervised clustering* and the solutions is to model somehow the "normal" state, according to the training data, and consider the instances that are furthest from normal as suspect of having noise. Such methods are K-Means, Gaussian Model Mixtures, Linear Regression, etc.

The second problem is harder than the first, since we do not know in advance what are the possible problems that can appear. We can see the first problem as a sub-problem of the second, as all the methods used to find noise in non-labeled data can be applied on labeled collections.

In our research, using the Ohloh datasets, we have no labels for our instances, we do not even know what is the distribution of our data and what are the possible errors that could be in our dataset. Therefore, we are facing the second class of the problem and our aim is to learn what is normal for our dataset and then highlight the points that are furthest from normal, as suspect of noise.

In this chapter, we will shortly explain how can we learn what is normal in our data, how do Gaussian Processes work, how we used them in our research to detect noise in our dataset and why did we chose them. We will limit our examples to two dimensional data, as visual representations can be easily generated in this case.

3.1 Examples of how to define what is normal in data

K-means

To learn what is normal in a dataset, different models can be used. A very simple, but actually powerful technique for noise detection is K-means, used in [43, 44, 45]. K-means is a clustering algorithm that splits the data into K clusters, each instance belonging to the nearest mean. To calculate the means that best fit our model is NP-hard, but efficient approximation methods are used that converge fast to a local minimum. In Figure 3.1, we see on the left how that specific data would be split in two clusters based on the two centroids. The total space is split by a line, what is on one side is green and what is on the other side is brown. This is how K-means would be used for clustering.

For noise detection, we should calculate the means and afterwards select a variance around the means that we consider normal. The points that are close to one of the means, are normal points while the other ones are noise. This is represented in the picture, where the brown points inside the circles are normal, while the green ones are noise. By using mixtures of Gaussian, we can change the circles in eclipses and create a more flexible model. Nevertheless, the principles and the form of the model is the same.

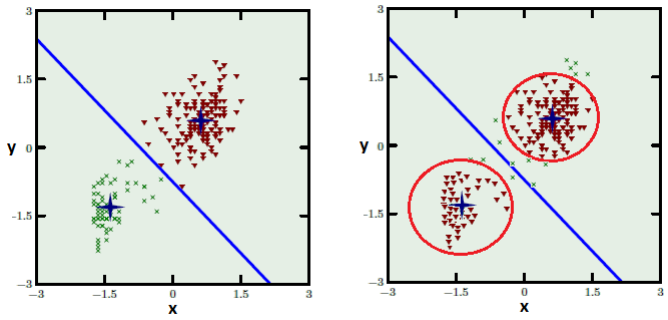

Figure 3.1: (left) Clustering with K-means. (right) Noise detection using K-means

Linear Regression

Another approach, is to learn a correlation between the two features, which is actually learning to predict one feature from the other. Linear regression was used in literature and some examples are decribed in [46]. We will use it also to exemplify.

We do not have a predefined class for our instances, therefore we choose one of the attributes to be the class that we want to predict. In Figure 3.2, after we learn the linear function we can make predictions for new points, e.g. the red point in the graph on the left. It is probably not an exact prediction but it is the estimation we can make, based on what we learned. This is how linear regression

would normally be used.

In our case we are not interested in the predictions, we need to detect noise. To judge if a point is noisy or not, we should calculate the Euclidean distance between its real value and its prediction and if it is higher than a prior selected tolerance we should report it as possible noise. In this approach, we are generating a model that varies around a linear function with a certain variance.

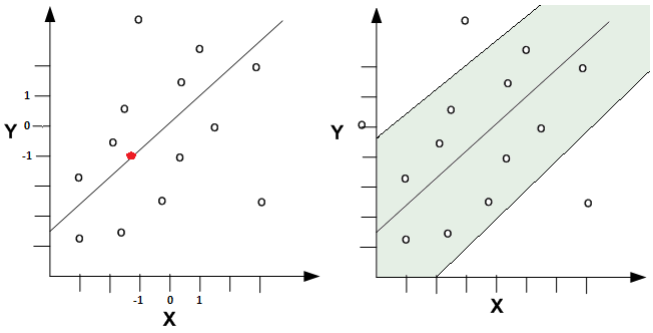

Figure 3.2: (left) Linear regression. (right) Noise detection using linear regression

The two models are different, each being better suited for particular domains. K-means, has the advantage that it can handle data that is grouped around some points, anywhere in space but it would have difficulties to model linear functions for example. While linear regression can have difficulties to fit non-linear data, e.g. data distributed on a circle.

Gaussian Processes

Because we do not have prior information about the Ohloh dataset and we want to be as flexible as possible, we chose Gaussian Processes(GPs). GPs define a probability distribution over non-linear functions and the model they create could be viewed as a non-linear mean function that has an adaptable Gaussian variance around it. The way our model looks can be changed by the choices we make for the covariance function and mean. Models like the ones presented in Figure 3.3 can be obtained. Some are more smooth or more continuous than others.

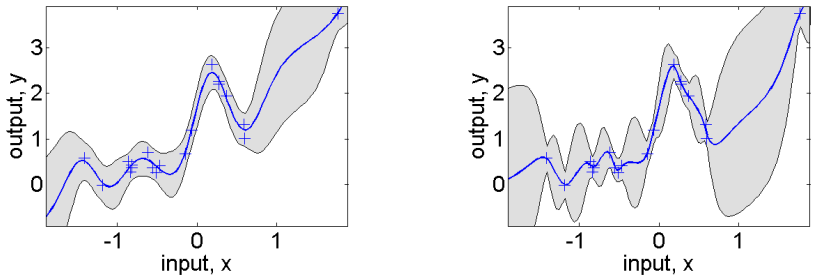

Figure 3.3: GP model with different covariance functions [28]

3.2 Gaussian Processes basics

The study of Gaussian Processes and their use for prediction is far from new [34]. The astronomer Thiele used Gaussian Processes since 1880 [33] for time series analysis, while in 1948 Wiener-Kolmogorov proposed predictions theories using Gaussian Processes for trajectories of military targets [34]. In this chapter we will explain concisely the mathematical background of the Gaussian Processes, using the introduction written by Boyle in [2] as outline. The deeper mathematical concepts are explained in Mackay [35] and Rasmussen [38]. For more details please read these papers, follow the online lectures [37] or read some of the other materials [29, 28]

In machine learning, the use of Gaussian Processes has gained more interest with the introduction of back-propagation for learning in neural networks. Neural networks are composed of connected layers of non-linear functions (neurons), each connection having a weight that establishes its importance. If we view this from a Bayesian perspective, these weights are basically defining a prior probability over the non-liniear functions. The back-propagation training methods from neural networks adapts these weights, which basically can be seen as defining a posterior probability for the non-linear function. The obtained model is a composition of non-linear function with certain probabilities learned while training. Neal [36] showed that Gaussian Processes can substitute the parameterized normal neural networks, facilitating simpler computations using matrices instead of more complicated parameter optimization for neural networks. He showed that the prior distribution of the non-linear functions are a subclass of probability distributions of Gaussian Processes, while the hyperparameters of the neural network determine the lengthscale of the Gaussian process.

In our thesis we use Gaussian Processes for regression. We describe the principles behind GPs starting from the notion of Parametric Regression, then explaining Bayesian Regression and last Gaussian Processes.

Parametric Regression

We will use X to define all the N input vectors $x_{(i)}$, each $x_{(i)}$ of dimension I, and real numbers Y for targets y_i corresponding to the inputs. A regression problems presumes to learn a mapping between the inputs X and the targets Y .

In parametric regression, our mapping is a function $f(x; w)$ defined in terms of the parameters w. The objective is to find the parameters that "best" describe the data.

One way to compare what is a better description of our data, would be to minimize a cost function. A typical cost function is the least squares, which basically measures the square distance between the predictions and targets.

$$L(w) = \sum_{i=1}^{N} (y_i - f(x_i; w))^2 \tag{3.1}$$

In this approach, the parameters that generate the lower value for the cost function, created the "best" model. As examples of parametric approaches we can think of polynomial regression, where the parameters are the coefficients of the polynomial, or fed-forwards neural networks, where the parameters are the weights.

There are three shortcomings of using cost functions. First, is the lack of error bars for predictions. This means that we obtain just a predicted value, but no measurement of how likely is it that this prediction is correct. Secondly, we need to initialize the parameters w, which can have a big impact on the resulting model. Third, we need to deal with problems of overfitting. Using the least squares cost function, we are are minimizing the function errors on the training data. We can obtain very low values for $L(w)$, close to zero for some domains, if we use more complex models (e.g. higher order polynomials) but this will most likely give bad performance on the test data because we overfit, learn details of the training data instead of general properties. If we choose simpler models (e.g. lower order polynomials) we might not be able to learn enough about our data and therefore also have bad prediction performance. The solutions is somewhere in between.

A second approach to compare what model is better, is to add a noise model besides the function:

$$y_i = f(x_i; w) + \epsilon_i \tag{3.2}$$

where ϵ_i is independent noise, usually of Gaussian distribution $N(0, \sigma)$. Now we can use a likelihood function $p(y|X, w, \sigma^2)$ as in [2], to obtain error bars for our predictions.

Bayesian Regression

We can use, for these parametric approaches, Bayesian theory to minimize the overfitting problem. With Bayes rule we can calculate the *posterior* distribution [2]:

$$p(w|y, X, \sigma^2) = \frac{p(y|X, w, \sigma^2)p(w)}{p(y|X, \sigma^2)} \tag{3.3}$$

where p(w) is the prior density function and it is set prior, based on what we think the mode should look like for the specific data. $p(y|X, w, \sigma^2)$ is the likelihood function and $p(y|X, \sigma^2)$ is the marginal likelihood that we calculate by integrating over the parameters w.

To make the prediction y_* for x_*, we calculate the probability (error bars) using the formula from [2]:

$$p(y_*|x_*, y, X, \sigma^2) = \int p(y_*|x_*, W, \sigma^2)p(w|y, X, \sigma^2)dw \tag{3.4}$$

The formula shows one important thing: that not only a single set of parameters contributes to the predictions, but all parameters do; "the predictive contribution from a particular set of parameters is weighted by its posterior probability" [2]. Combining models with different parameters to obtain the final one makes the generated model less likely to *overfit* our training data.

Gaussian processes

Gaussian Processes can be considered replacements for many parametric models. The major advantage is that the functions are not explicitly parametrized, making them more useful in situations were we do not know beforehand what the data looks like and how to choose the parameters.

We define a probability density function $p(f)$ over a function space F. We then sample function f, $f : X \to \mathbb{R}$ from F, according to the distribution $p(f)$. If we calculated $f(x)$, where $x \in X$ for multiple sampled functions f we obtain a random variable $f(x)$ with a certain distribution. An example is shown in Figure 3.4 from [2].

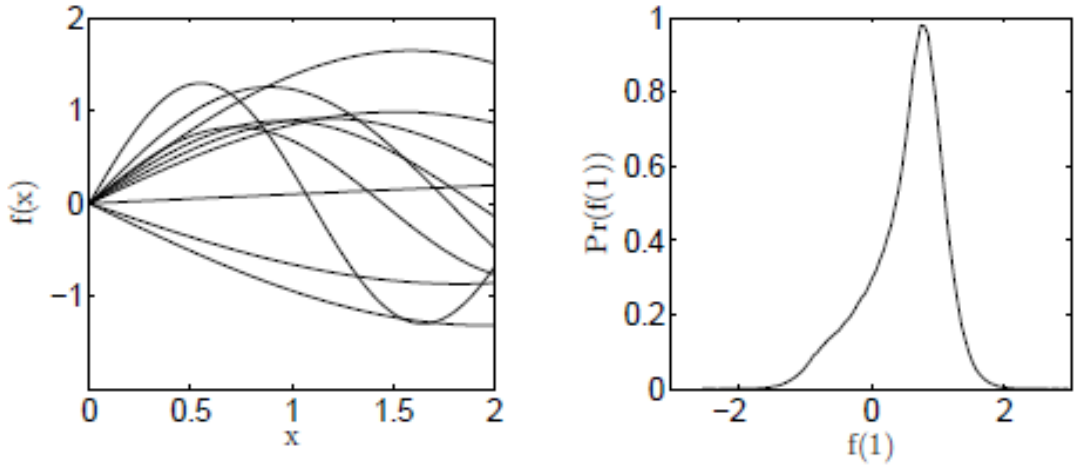

Figure 3.4: (Left)Multiple sampled functions f. (Right) Distribution of $f(1)$ after normalizing the histogram of 1000 sampled functions at $f(1)$ [2]

If $x \in X$ and the distribution of $f \in \mathbb{R}$ for any finite N is a multivariate Gaussian, we call this stochastic process a *Gaussian process*.

Gaussian processes are non-parametric methods and are fully defined by a *mean function* μ and a *covariance function* $C(x, x')$. The prior mean function μ is usually unknown, therefore it is set to zero and learned from the training data. The covariance function must be chosen by us and it defines how smooth the models looks like.

Having these two set, we have a prior for our Gaussian Processes. Nevertheless, the prior specifies only the properties of the functions and does not depend on the training data. For this hyperparameters θ are introduced to describe the mean

and the covariance functions, making them more adaptable. By inferring θ from the training data we are afterwards able to make predictions.

The mathematical theory of how the hyperparameters are learned from the training data and how the predictions are made using matrix multiplications, will not be presented in this thesis. Please read the papers suggested at the beginning of this Chapter for more details. We will not present some graphical representations of Gaussian Processes produced with different covariance functions.

Covariance functions Choosing the covariance function seems to have the greatest impact on the generated mode. To be a valid covariance function it must be positive semidefined which means it has to respect the condition:

$$\int C(x, x')f(x)f(x')du(x)dy(x') \geq 0 \tag{3.5}$$

Some of the used covariance functions are presented in Table 3.1.

covariance function	expression
constant	σ_0^2
linear	$\sum_{d=1}^{D} \sigma_d^2 x_d x'_d$
polynomial	$(xx' + \sigma_0^2)^p$
squared exponential	$exp(-\frac{r^2}{2l^2})$
Matern	$\frac{1}{2^{v-1}\Gamma(v)}(\frac{\sqrt{2v}}{l}r)^v K_v(\frac{\sqrt{2v}}{l}r)$
exponential	$exp(-\frac{r}{l})$
γ-exponential	$exp(-\frac{r}{l}^\gamma)$
rational quadratic	$(1 + \frac{r^2}{2\alpha l^2})^{-\alpha}$
neural network	$sin^{-1}(\frac{2x^T \sum x'}{\sqrt{(1+2x^T \sum x)(1+2x'^T \sum x')t}})$

Table 3.1: Several covariance functions [29]

We present in the Figures 3.5, 3.6, 3.7 how changing the covariance function can affect the generated model.

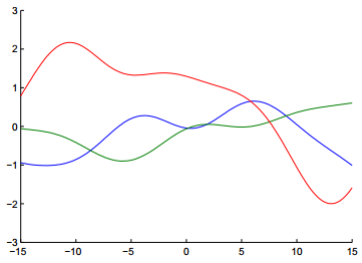

Figure 3.5: Three sample GP functions with squared exponential covariance, lenth-scale $\lambda = 5$ and $a = 1$ from [40]

The squared exponential covariance from Figure 3.5, has unrealisticly strong smoothness assumptions. Nevertheless it is the most used in Machine learning because it is infinitely differentiable.

The Matern covariance from Figure 3.6, is rougher than the squared exponential being considered by some to better map real work circumstances. If we would set the hyperparameter $v \to \infty$, Matern becomes actually the squared exponential covariance function.

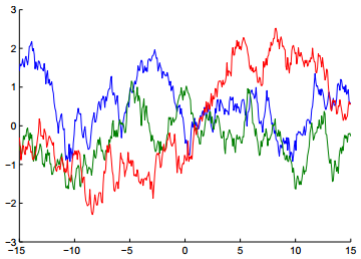

Figure 3.6: Three sample GP functions with Matern covariance, $v = 1/2$, lenthscale $\lambda = 5$ and $a = 1$ from [40]

The neural network covariance from Figure 3.7, is a nonstationary function. The prior functions were stationary covariances. Nonstationary covariances allow the model to adapt to functions whose smoothness varies with the inputs. E.g. If the "noise variance is different in different parts of the input space, or if the function has a discontinuity, a stationary covariance function will not be adequate" [42].

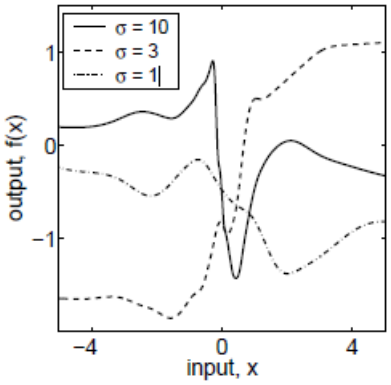

Figure 3.7: Samples GP functions with neural network covariance, $\sigma_0 = 2$ and σ shown on the image [29]

Advantages

Using Gaussian Processes has the following advantages:

- **non-parametric**: Successful methods in machine learning are essentially non-parametric. Moreover, being able to adapt the hyperparameters from the training data and not setting them prior permits a lot of flexibility in unknown datasets.
- **flexible**: Many popular non-parametric methods used in machine learning can be substituted by particular Gaussian Processes. Flexible models can be generated which have been shown to consistently outperform the more conventional methods.[4]
- **unlikely to overfit**: Overfitting, is adapting our model too much on our training data. We end up having a great performance for the training data, but not describing the underlying relationships, therefore obtaining bad predictions on newly seen data. GPs are known to be less prone to overfitting [2] which should help us create a generalized model from the training data, that could describe well the test data.
- **can measure the probability of our predictions**. We do not only obtain a prediction based on the learned model, but we also estimate the probability of our prediction to be true, based on the model we learned.

All these propose GPs to be a valid technique for noise detection, that could achieve good results. A first known disadvantage of GP is the computational complexity. This is $O(n^3)$, n=number of datapoints. For our datasets of more than 800 000 instances, it was a problem to use all instances for training. We solved this by training only on a portion of the data(1%), which made our script execute only two hours. Secondly, we are expecting GPs not to be able to detect systematic errors, because they would be learned by our model. For removing these, rule based filtering might be more appropriate.

3.3 Matlab framework

To build our tool, we used the framework supplied by Rasmussen and Williams [28]. It includes several predefined functions to choose from for the mean or covariance, and methods to compute our model and predictions. To define our GP, four choices need to be made:

- **Mean function**: We set this to 0 and learn it from the training data. This is the normally used approach in case no prior information is known about the mean.
- **Covariance function**: There are many possibilities of covariances functions that are predefined in the matlab framework, more can be generated by composing (adding or multiplying) these covariances or by manually specifying new ones.
- **Likelihood functions**: "The likelihood function specifies the probability of the observations given the latent function, i.e. the GP (and the hyperparameters)" [28]
- **The inference methods** "specify how to compute with the model, i.e. how to infer the (approximate) posterior process, how to find hyperparameters, evaluate the log marginal likelihood and how to make predictions"[28].

CHAPTER 4

Simulated noise experiment

Before running the GP algorithm on the unknown Ohloh data, we measure its potential in identifying noise in controlled settings. By this we test:

- if Gaussian processes are capable to detect noise
- if our methodology, our way of setting up the experiment, is correct and
- experiment with different GP settings and see their influence on the results

As a ground truth experiment we chose a known dataset, added artificial noise to it and measured the noise detection performance of our approach. The following sections describe the chosen dataset, the experimental setup, the results obtained and states our conclusions.

4.1 Dataset description

In order to determine if Gaussian Processes is a viable algorithm for our problem, we first selected a known database for experimenting. We chose from the UCI repository the Housing dataset. It contains 506 instances, each of them describes 14 characteristics of housing in the suburbs of Boston. The description of all the attributes can be seen in Table 4.1.

We considered this dataset an appropriate one for our experiment since it is highly used in the Machine Learning community, increasing our trust in the validity of the data. Moreover it has similarities with Ohloh :

- it is positively defined and has no missing values for attributes
- contains 13 continuous attributes (including "class" attribute "MEDV") and 1 binary-valued attribute [5]. This is ideal for regression and similar to Ohloh where all attributes are continuous.

Abbreviation	Description
CRIM	per capita crime rate by town
ZN	proportion of residential land zoned for lots over 25,000 sq.ft.
INDUS	proportion of non-retail business acres per town
CHAS	Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
NOX	nitric oxides concentration (parts per 10 million)
RM	average number of rooms per dwelling
AGE	proportion of owner-occupied units built prior to 1940
DIS	weighted distances to five Boston employment centres
RAD	index of accessibility to radial highways
TAX	full-value property-tax rate per $10,000
PTRATIO	pupil-teacher ratio by town
B	$1000(Bk - 0.63)^2$ where Bk is the proportion of blacks by town
LSTAT	% lower status of the population
MEDV	Median value of owner-occupied homes in $1000's

Table 4.1: Housing attributes

- it is complex enough not to represent a trivial test for GP. The number of features is higher than for our Ohloh dataset and their correlation is diverse as shown in the Figures 4.3 4.1, 4.4, 4.2.

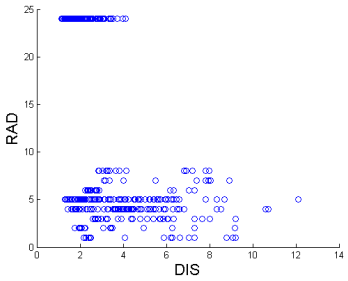

Figure 4.1: Correlation between DIS and RAD

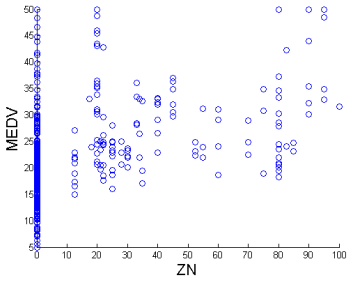

Figure 4.2: Correlation between ZN and MEDV

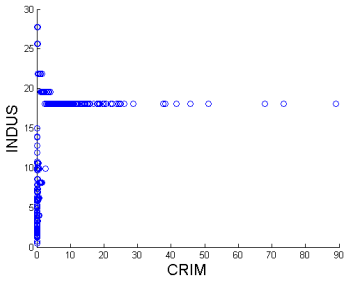

Figure 4.3: Correlation between CRIM and INDUS

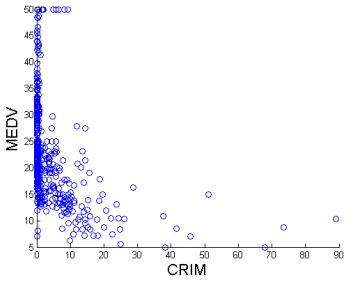

Figure 4.4: Correlation between MEDV and CRIM

4.2 Experimental setup

For our experiment, we considered the first 13 dimensions as input features, while MEDV is the attribute that we want to predict. Our objective is to answer the question:

"If we modify the predicted MEDV value of the house, for some instances, between -100% and +100%, can we detect the noisy points in our data using Gaussian processes? "

New points with noise were created by randomly sampling 50 instances from the existing housing dataset and afterwards adding random noise to them. Adding noise presumes, in our case, modifying the MEDV values by randomly multiplying them with a coefficient between -1 and 1, while keeping the other features intact. This is equivalent to saying, that in case all the other properties of an instance remain the same, the predicted median value of the home will increase or decrease with max 100%.

The distribution of the used noise is presented below. This distribution has impact on our experiment, since adding 2% to MEDV is much harder to detect as noise (maybe it should not even be considered noise) than adding or subtracting 90% of the original MEDV, which should be definitely highlighted.

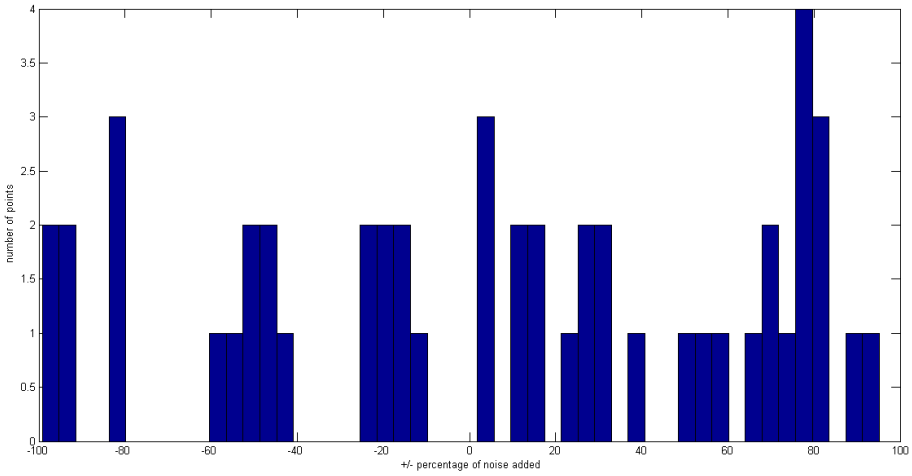

Figure 4.5: Histogram of percentage of noise added to MEDV

After creating the noisy points, the dataset was randomly split in two parts. We obtained 259 data points for training and 247 for testing. To the testing dataset we added the 50 new noisy points, ending up to a total of 297.

Finally, we used the GPs regression algorithm on our data. We aimed to test three characteristics:

1. The success of GP in the default settings
2. What is the impact on the results when varying the hyperparameters?
3. How does changing the covariance function impact the results?

Our results are discussed in the following section.

4.3 Results

4.3.1 Gaussian processes in the default settings

To show the capabilities of the GP algorithm, several experiments were run on the housing dataset. In the default GP process, we chose the following configuration :

- **mean function**: it is set to zero and learned from the training data.
- **covariance function**: We used the neural network covariance function with the posterior hyperparameters learned from the training data.
- **likelihood function**: The Gaussian likelihood with posterior standard deviation learned from the training data.
- **the inference method** is exact, without approximations.
- **maximum number of function evaluations** is set to 100.

Using this trained model, we calculated the log-probabilities of each of the points from the test dataset(including the new noisy instances) and computed the Receiver operating characteristic(ROC)curve. This captures both the true positive rate (TPR), the noisy points correctly classified, and the false positive rate(FPR), the percentage of normal points classified as noisy of the model when the discriminating threshold is varied.

Figure 4.6 shows how selecting different thresholds (minimum accepted probabilities) we can detect the noise in the test dataset and the effect it has on the FPR(normal instances wrongly classified as noise). We notice that we can find 54% of the added noise with less than 9% FPR.

By sorting the log probabilities of points to be noise, in the first 50 most probable we had 27 actual noise points. These results show a very good performance of the algorithm for the current dataset. Our approach does not make the difference between outlier and noise. Therefore we should consider that in the first 50 most probable points, besides the 27 noisy ones, we could have also highlighted outliers in fact, which could be of interest in some cases.

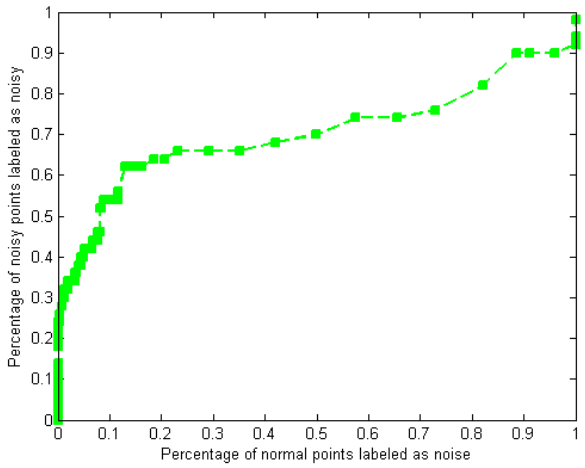

Figure 4.6: ROC curve for GP algorithm on all the test dataset.

By manually inspecting the log-probabilities assigned to each point, we noticed that the first 12 most probable points are labeled correctly as noise. On the other hand, the last 4 points, that should have the lowesst probability to be noise, are actually noisy points induced by us. We can explain this by recalling the noise distribution from Figure 4.6. From the histogram we can see that the added noise can be between -100% to 100%. It is clear that bigger changes make it easier for us to identify noisy points. In our case the last 4 points were changed with approximately 2%, 12%, 4.7% and 4%.

Nevertheless, this should not make them more probable than points that have not been changed at all. The complete explanation lies in the way we generated noise. We randomly selected 50 points from all our data and created new instances with noise from them. Thus some of the original instances were split in training others in test data. The original measurements for these 4 data points ended up in the training dataset and the modeled managed to fit them very well. This, correlated with the small amount of noise added, made the disturbed instances from the the test data hard to distinguish.

Our manual analysis highlighted two concerns. First, small changes are impossible to detect and might not be noise, as our inputs are continuous and slight variation in value estimation is normal. Second, if the noise is similar to the training data (or part of the training data, as it will be in the Ohloh dataset) its probability to be classified as noise will decrease.

We ran more tests on the dataset, to measure how would our performance change if we remove the noisy instances, from the test data, that changed the initial measurement with less than 20% respectively 40%.

The ROC obtained after removing less clear noise shows a visible increase in performance. Also in Table 4.2 we compare the difference in performance based on the given noise. The decrease in FPR is significant when the noise is more significant.

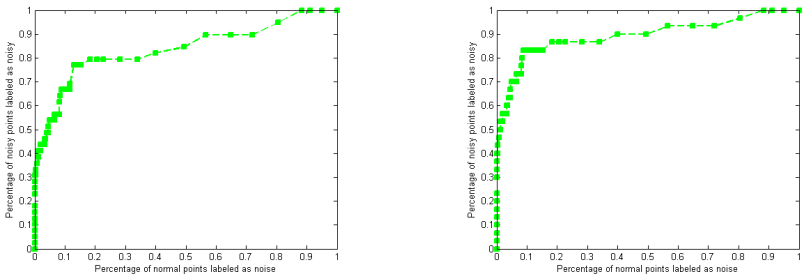

Figure 4.7: ROC after removing noise points that affect MEDV with less than 20% (left) and 40% (right)

Test data with	**Remaining noise**	**FPR for 50% recall**	**Number of instances found in the first 50**
All the noise	50	9%	27
Noise with more than 20% deviation from original	39	4.4 %	26 (22 out of first 30)
Noise with more than 40% deviation from original	30	1.2 %	25 (19 out of first 30)

Table 4.2: Comparison between the 3 test data containing different amount of noise

4.3.2 Impact of the hyperparameters

The posterior hyperparameters of the Gaussian Process are learned from the training data. Nevertheless, the number of function evaluations is manually set by us. This parameter sets the maximum number of evaluations preformed, if convergence is not reached before. We tested how varying it can influence our performance.

We realized that the performance of the algorithm can be significantly changed as noticed by the results in Figure 4.8. In these figures we tested the performance of each of the models created on the three types of testing data (all instances, or instances with noise variance higher than 20% and 40% from normal) by varying the number of function evaluations from 1 to 100. We measured the FPR to detect A minimum 50% of the noisy points. We are aiming to obtain the smallest values for the FPR.

The best results are obtained when the parameter is set to 12. In this case the FPR becomes 4.8% for the test data with all the noisy points, 0.8% and 0% for test data with more than 20% respectively 40% variance from normal.

The performance can be greatly changed when setting the parameter to 13. On the test data with all the points, the FPT reaches 100%, meaning that we can not detect 50% of the noise without looking at all the other normal inputs from the testing dataset. This is highly undesirable.

The negative log probability likelihood obtained for the different number of

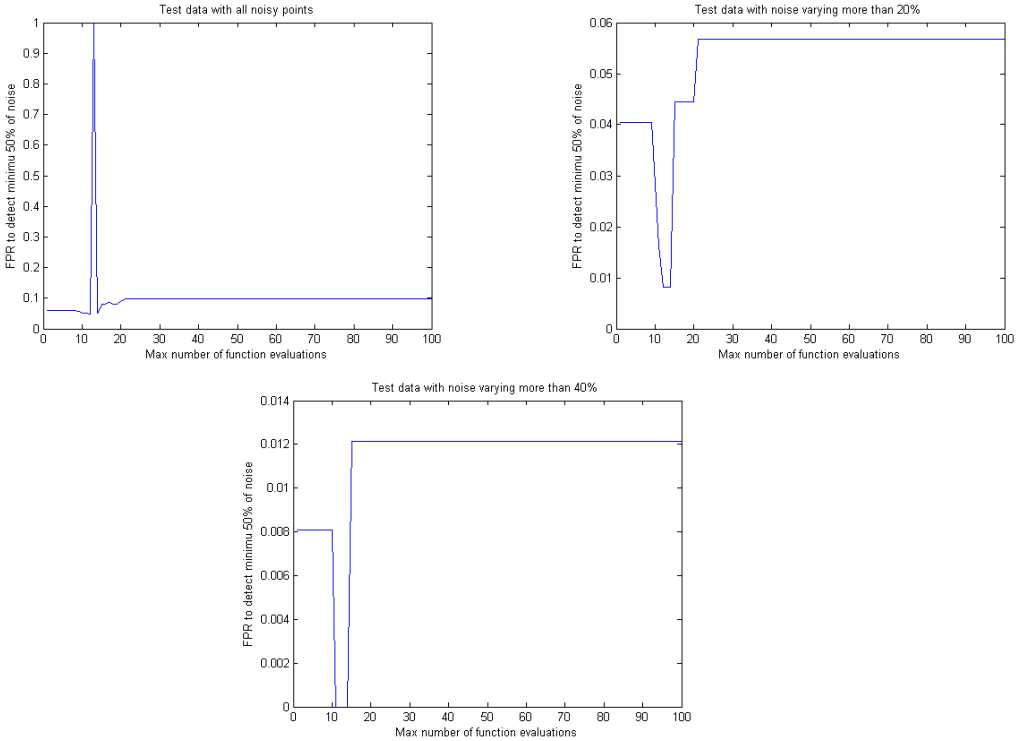

Figure 4.8: FPR to detect minimum 50% of the noise on the 3 test datasets when varying the maximum number of function points

function evaluations is plotted in Figure 4.9. We want to minimise this. A smaller value, states that the obtained model is more likely to correctly approximate the training data. We notice that 15 function evaluations is a cutting point, after which our approximation of the training data becomes better. Therefore convergence is reached on this dataset by setting the maximum number of function evaluations higher than 15. It is safer, in general, to choose a substantially higher number.

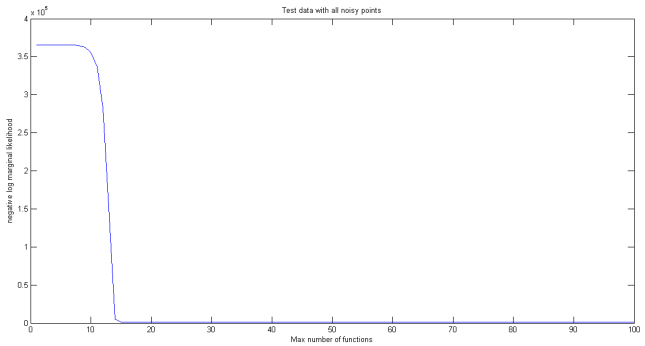

Figure 4.9: Variation of the negative log probability likelihood with the maximum number of function evaluations

We conclude that to minimize the problem, it is safer to choose a higher number of basis functions and have a stable prediction, instead of working in the risky area.

4.3.3 Impact of the covariance function

The Matern covariance

We tested how changing the covariance function would affect our performance. Based on our data, we considered that the Matern covariance function would be another suitable option. In Figure 4.10 we plotted the ROC curve.

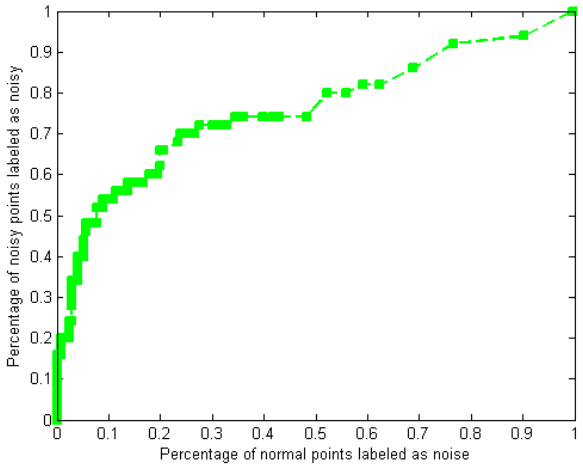

Figure 4.10: ROC for Matern covariance function

The differences between the ROCs of these functions are not that high, nevertheless the area below the curve is higher for the Matern covariance than of the model obtained with the neural network function. On the other hand, looking at the beginning of the curve we notice that the FPR has a increasing hop around 20% recall, while for neural networks it is around 30%. This would mean that the first 20% of noise instances can be noticed with a small FPR for the Matern covariance function while for Neural Network we can detect 30% before increasing the FPR, making it more convenient for big datasets. If we would aim to detect a higher percentage of noisy points the Matern covariance function would be more appropriate.

4.3.4 Varying other settings

Using the Inverse Gaussian likelihood function for strictly positive data and the Laplace approximation

To force our prediction to be in the positive domain we can change the likelihood function and the inference method. This combination generated the ROC from Figure 4.11 on all the dataset.

The result is noticeably better than the previous inference method, creating an increased steep at the beginning of the curve, meaning that more noise can be detected with less false positives. This is confirmed by our second measurement;

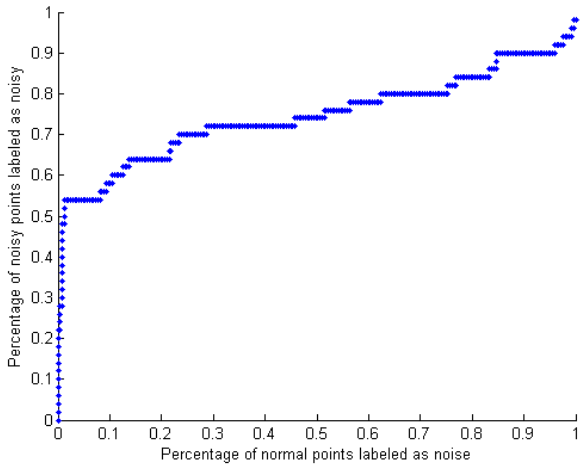

Figure 4.11: ROC for the Inverse Gaussian likelihood function

we detected 50% of the noise with 1.21% FPR instead of 9%. On the other hand, when usingh the Inverse Gaussian likelihood, the domain of the output is forced to be strictly higher than zero, which is not the case for our Ohloh dataset where we have zero values.

Using approximation methods for big data sets

The computational complexity of the normal GP is $O(n^3)$ where n is the number of data points. Approximation methods have been developed that lower the computational complexity to $O(n^2 * m)$ where m is a smaller parameter set by the approximation method. We tried these methods on our data and obtained the ROC in Figure 4.12. As expected the accuracy of our noise detection decreases

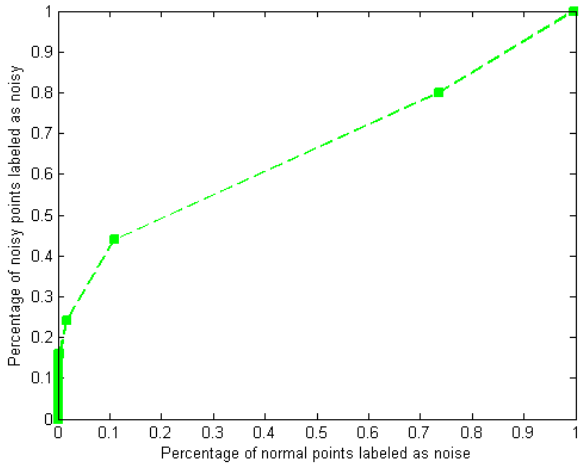

Figure 4.12: ROC for approximation methods

significantly. This has also been shown in other research [41], where a full GP has

better performance than approximation methods. Based on these, we will use the full GP for Ohloh, which can support up to 10 k points.

4.4 Discussion

Our ground truth experiment **confirmed that GP can be successfully used to detect artificially induced noise** in the data with good accuracy and that our experimental setup is valid, but also highlighted several challenges that we will have to confront on the Ohloh dataset:

- **False positive rate (FPR)** becomes more important in big datasets. Putting it in perspective, for the Ohloh dataset we have more than 800 000 points. A 9% FPR would mean that around 82 000 points are wrongly classified, which is a big amount of wrongly labeled data.
- **If we train our model on noisy points, these points might become harder to detect afterwards**. In the Ohloh dataset we do not have the luxury of knowing what points are noise and exclude them from the training dataset.
- **Outliers and noise can not be differentiated**, since we do not have labeled data. The instances that we highlight need to be manually inspected in order to conclude if they are noise, outliers or false positives.
- **The performance is influenced by the noise type**. Our approach is influence by the noise amount (deviation from normal). Small noise is very hard to detect, while big variations from the normal case will be easier to detect.
- **The maximum number of functions evaluated influences the performance of the GP algorithm.** A high number should be chosen; this would let the inference converge by itself and not stop it prematurely.
- **Using likelihoods function for strictly positive data can increase our performance** also in Ohloh. On the other hand our outputs can have zero as a value, are not strictly positive, which means it would not be able to calculate the probabilities for these points. We could adapt all zeros to be a bit higher than zero but this would presume altering much of the data.

CHAPTER 5

Noise detection on real world data

One of our research goals was to evaluate the quality of the Ohlhoh dataset. Since the dataset is large, an automatic approach was needed, which led us to using Gaussian Processes. In the previous chapter we shown that Gaussian Processes can obtain good results on artificial noise. In this chapter we will test how Gaussian processes will perform on a real world dataset.

5.1 Description of the dataset

For our experiments, we mined our data from Ohloh "an open source directory that anyone can edit. It features comprehensive metrics and analysis on thousands of open source projects" [6]. As presented even in its description, Ohloh offers editing rights to anybody; the name, description can be changed, but also more interesting things: history, code location, etc. Modifying these, can have a significant impact on the metrics obtained.

The Ohloh repository contains in total 65,712 projects of which we downloaded the metrics for 12,360. The data was mined by M. Bruntink and can be found publicly on Git[1]. He selected the first projects that Ohloh offered from their API, based on activity (most active projects).

For each project we obtain three files ActivityFacts, SizeFacts and MetaData. The MetaData.xml file contains general information about the project e.g.: main programming language, other languages, licence, average rating etc. However, we did not use it in our research. In Table 5.1 we show what each of the other two files contains, the name of metrics being self-explanatory. The measurements are taken monthly, therefore each month we create an activity fact and a size fact.

In Figure 5.1 we capture project ages from our dataset. We can see that most of the projects live for about 60 months, but we have also much older ones. Also by inspecting the size facts we found that projects can have different sizes, varying

[1] `https://github.com/MagielBruntink/OhlohAnalytics/tree/searhus`

ActivityFacts file	**SizeFacts file**
month	month
code added	code (total)
code removed	comments (total)
comments added	blanks (total)
comments removed	comment ratio
blanks added	commits (total)
blanks removed	man months (total)
contributors	
commits	

Table 5.1: Metrics saved for each month in each of the xml files mined from Ohloh

from 12 LOC to 158 milion LOC (project DD-WRT).

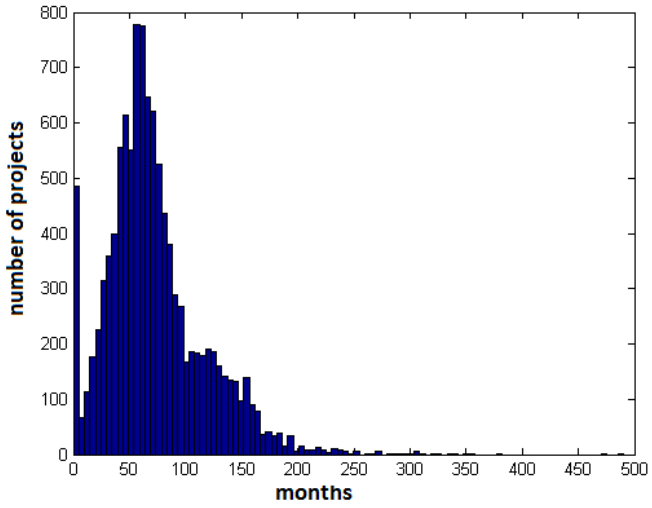

Figure 5.1: Histogram of the project length in months

We selected the first 10 000 instances from our dataset and generated images that show the correlation between some attributes. Figures 5.2, 5.3, 5.4, 5.5 exemplifies the correlations between attributes in the activity fats file, while Figures 5.6, 5.7, 5.8, 5.9 exemplifies the correlations between attributes in the size facts file. We can see the correlations are diverse some similar to the Housing dataset, while others not.

From Figures 5.7, 5.8 and 5.9 we can already notice that in the size facts file, some instances have negative errors since the plot does not start in the coordinated (0,0). This is an early detection of noise in our data, obtained by simple visual inspections.

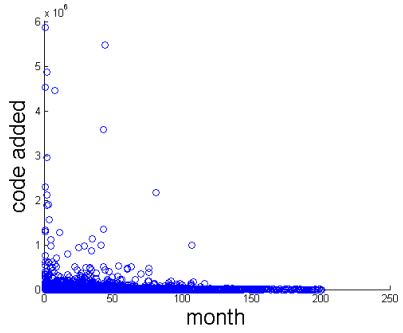

Figure 5.2: Correlation between month of life and code added

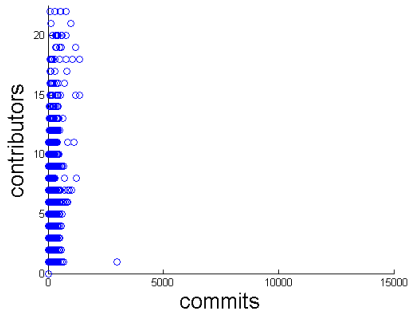

Figure 5.3: Correlation between commits and contributors

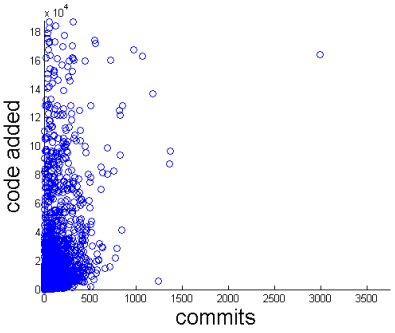

Figure 5.4: Correlation between commits and code added

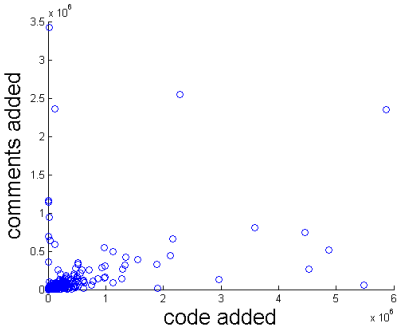

Figure 5.5: Correlation between code added and comments added

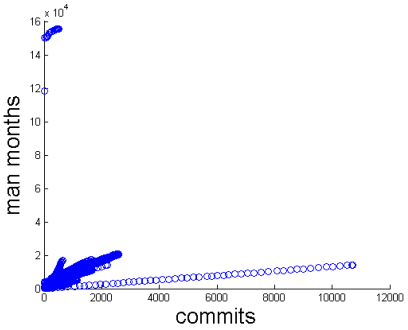

Figure 5.6: Correlation between total commits and man months

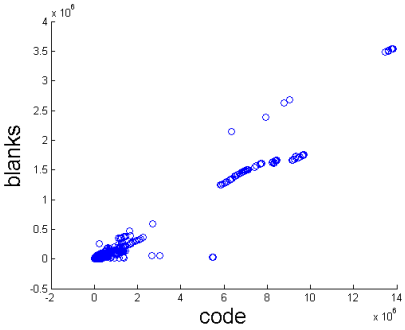

Figure 5.7: Correlation between total code and blanks

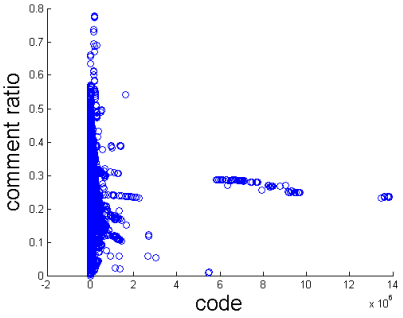

Figure 5.8: Correlation between total code and comment ratio

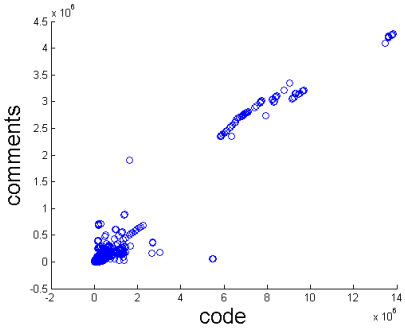

Figure 5.9: Correlation between total code and comments

5.2 Experimental setup

The goal of these experiments is to test the capacity of Gaussian Processes to detect noise on large real world data and to assess the quality of the Ohloh datasets. We have two big experiments on two datasets generated from the Ohloh activity and size facts and a smaller experiment that we ran only on the Rascal project. Our hypothesis is that GPs will suggest to us interesting instances in our data, abnormal peaks in the evolution of the projects, that could be considered noise for some purposes. We will describe on this section the steps we took to set up the two big experiments.

First, the Ohloh dataset is split in two, activity facts which are gathered from the ActivityFacts files of all the projects and size facts from the SizeFacts files. We did not merge them because for some projects (e.g. which have no activity) the activity facts is logged for each month, while only one size fact is written in the size file. For activity facts we obtained an array of 889 117 instances with 10 columns. Besides the columns from the xml files, we added an extra column that is our project id. This column is not used to learn the model, it is used afterwards by us to manually inspect the data. For size facts we obtained 838 158 instances with 8 columns, of which one is the project id. For both datasets, the month was transformed from a date format to numbers representing the month of life for a project (e.g. first months of activity). More explanations of the process can be found in the Appendix.

Second, we learn our model for each of the dataset. We used the *commits* as attributes that we want to predict (class attribute) for activity and size facts. Ideally, we would use all the data to learn and test the model, but because of the size of the data it was computationally too expensive. We chose to use as training only 1% of the data, that was randomly selected. Alternatively, approximation methods are available that can reduce the time, but they affect the performance. By training only on a subset and testing on the entire data, the cleaning process takes around two to three hours to execute.

Afterwards, we tested our models on the entire dataset. Our learner was configured the same way as in the default Housing experiment, having :

- **mean function**: set to zero and learned from the training data.
- **covariance function**: neural network covariance function.
- **likelihood function**: Gaussian likelihood.
- **the inference method** is exact, without approximations.
- **maximum number of function evaluations** is set to 100.

As a result, we obtained the prediction based on our model of each of the testing point and the log probabilities of the predictions to be true, compared to

the actual measurement written by Ohloh. We sorted all the data, so the less probable prediction would be first, these being most likely noise or outliers.

During the process of transforming the data from xml to matlab structures, or before running our tool, we found several errors in our data. From the total of 12,360 projects:

- 426 projects have missing ActivityFacts files
- 433 projects have missing SizeFacts files
- 161 projects have empty SizeFacts files; the file contains no entries or only one entry that is 0.
- 3 projects have $-Infinity$ for comments ratio in the size file. This is because the $code + comments$ is 0 (zero) and Ohloh uses this to make divisions.
- 70 projects have a total of 1432 entries with negative values in the size facts.

The projects that encountered one of the first 3 types of errors were removed from the dataset. Only the monthly instances that encountered $-Infinity$ values were removed from the size data. The negative numbers were left, as we wanted to test if our automated approach would also be able to detect them.

5.3 Results

In this section, we will present the results obtained on data from the activity files and from the size files from Ohloh. But first, we ran a small experiment on the Rascal project to test if Gaussian Processes can model the type of data measured by Ohloh.

5.3.1 Rascal experiment

The first experiment we ran, was on a sub-piece of the Ohloh data. We selected a project well know to us, Rascal, and tried to see what our method would suggest to be noise. By doing this we wanted to confirm that our method can model attributes measured by Ohloh.

We know one atypical event that happened when generating the documentation, which is highlighted in Figure 5.10 with a red dot above. A good outcome for this experiment would be if this point is ranked in the top most probable noisy points.

We trained and tested our GPs algorithm on the 58 months of activity facts from Rascal. We sorted the rows, which represent one month of activity facts, by the predictive log probability. Table 5.2 shows the months that are most probably noisy and one of the least probable noisy months (the last 17 months have the same probability).

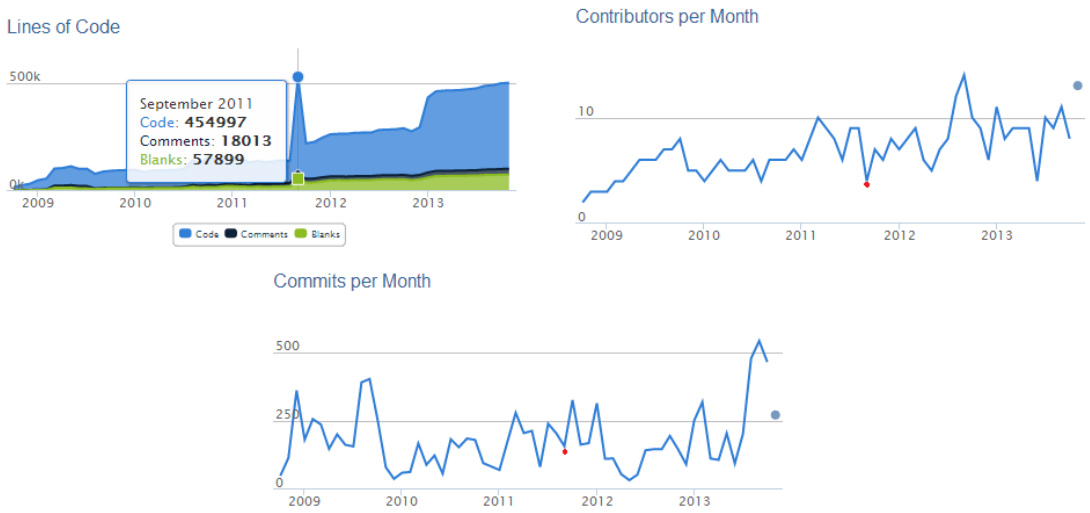

Figure 5.10: Rascal graphs from Ohloh

The most probable noisy instance is declared September 2011. This is the known month where the documentation was generated (several times) adding a big number of HTML files that were afterwards removed. Next probable noisy months were August 2010 and October 2009. Looking at the table 5.2 we can see that in Aug. 2010 there was a lot of activity for only 4 contributors. In Oct. 2009, comparing to previous months where the contributors were less and the commits were much higher (403), these two metrics varied inversely, therefore this month the model probably had to adapt. Nevertheless, after looking in the commit logs and checking the history we could not find any reason why these two months of measurements should be noise.

Rascal Metrics	1^{st}	2^{nd}	3^{rd}	*most normal month*
month	36 (Sept 2011)	23(Aug.2010)	13(Oct. 2009)	55(April 2013)
code added	870 709	195 155	69 646	4 200
code removed	518 752	166 422	66 685	2 798
comments added	5 621	16 577	1 573	238
comments removed	5 100	12 511	1 300	242
blanks added	38 104	15 272	5 372	640
blanks removed	1 063	10 417	4 728	210
contributors	4	4	8	9
commits	165	131	222	87

Table 5.2: Most probable noisy months and one of the most normal months.

Conclusion The experiment on Rascal, showed that GPs can model the features measured by Ohloh and correctly identified the noisy point from our data. The next suggested instances are normal from our perspective, but we knew no other abnormal month for this project.

5.3.2 Activity facts results

In this section we discuss the results we obtained using Gaussian Processes on the activity data. According to literature [41], GPs can train on up to 10,000 instances without using approximation methods, but the time increases fast with the increasing size; e.g. it took around one hour to train on 7,500 instance and it takes around two to train on 8,500 inputs. We chose to use 1% of the data, around 8,500 instances randomly sampled, to learn our model and then make predictions for the entire dataset.

As we mentioned before, we predicted the *commits* of each month for all the projects and calculated the log probability of our predictions to be true. The other features were considered inputs. At the end, we sorted our data based on the log probabilities, having the predictions that are ranked as less probable on top. These are the points suspected to be noise. The histograms of the commit values from all the activity data and only the training set are plotted in Figure 5.21. We zoomed in, to be able to visualize the results, since about half of our data has zero commits and the first bin of the histogram is actually much larger that all the rest.

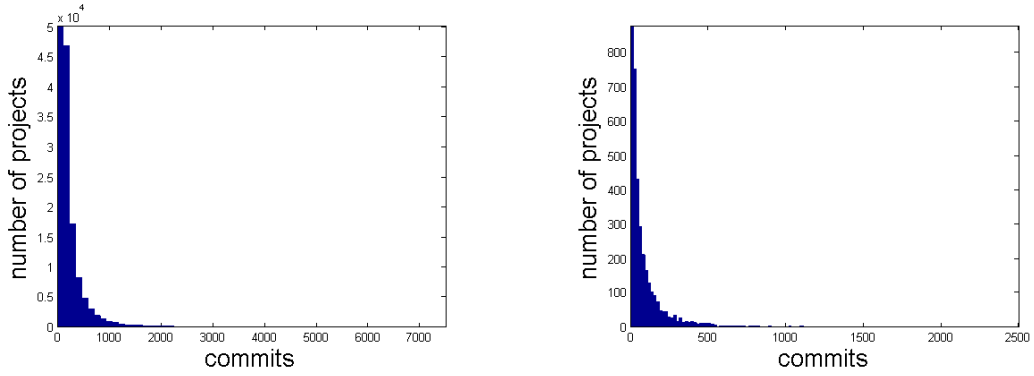

Figure 5.11: (Left) Zoomed in histogram of all commits form the activity array. (Right) Zoomed in histogram of all the commits form the training set of the activity data

To interpret our results, we looked at the first points and judged if they are noise, outliers or normal points. As expected it was difficult to decide on the quality of an entry, as limited information could be obtained for these projects. We analyzed the quality of data having in mind two possible purposes:

- predicting the project's death
- estimating the general productivity per person, in a project

In the first 10 points that we inspected we found measurements from 3 projects which will be further discussed .

ApacheFlex project

The first and fourth most probable noisy points were from ApacheFlex. Their values are presented in the Table 5.3 and the graph of the projects evolution are in Figure 5.12. We marked the points that we are discussing about on the graph with a small red dot.

ApacheFlex metrics	1^{st} **point**	4^{th} **point**
month of life(month)	1 (2012-01-01)	2 (2012-02-01)
code added	2 291 792	4 872 344
code removed	669 122	131 966
comments added	2 542 260	518 488
comments removed	633 480	81 117
blanks added	559 541	1 692 530
blanks removed	94 326	7 260
contributors	4	4
commits	118 176	31 868
predicted commits	23.56	30

Table 5.3: 1^{st} and 4^{th} most probable noisy points from our dataset.

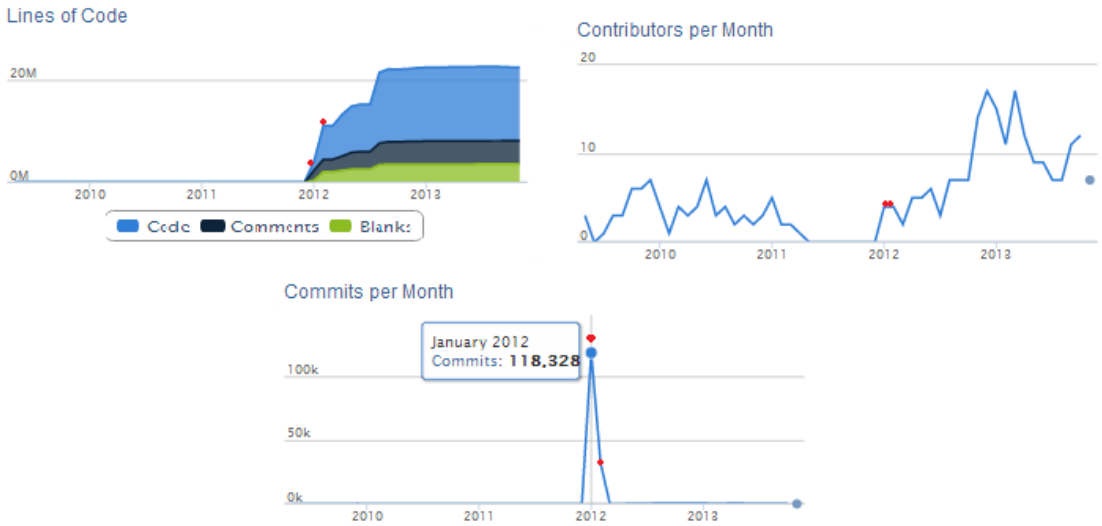

Figure 5.12: ApacheFlex graphs from Ohloh

Discussion

These two instances are very unlikely points because **4 contributors** add 2.2 million lines of code in the first month and 4.8 million LOC in the second month with 118 176 respectively 31 868 commits. These two months are, most likely, initial imports thus explaining the bad correlation between measurements. Relating to our two possible purposes:

- if our objective is to *predict a project's death*, these points should not be considered noise. They might actually be of interest to detect. Maybe repeated big variations like these could be a cause for the death of projects (this is just an untested hypothesis).

- if our objective is to *estimate the general productivity per person, in a project*, the quality of these entries is questionable. Having 4 contributors add that much activity to a project could affect our conclusions, therefore we should consider these points noise and remove them from the data.

  A rule could be generated that could highlight similar points. e.g.: if the contributors are less than 10, the LOC added should not be above 1 million and the commits not more than 10,000 in that month.

While inspecting these points we noticed two other flaws:

- in the activity facts, January 2012 is the first month of the project while on the graph we have activity since May 2009.

- the exported measurements contradict what is shown on the graph. According to the activity facts we should have 1 622 670 LOC in the first month, but on the graphs the total code was 1 770 327 LOC. From 2 separate measurements we have 2 different values, which is a clear sign of error in the Ohloh system.

OpenVZ project

Four points from the top ten that are highlighted as having a high probability of noise, are from the OpenVZ project. Their values are presented in Table 5.4 and a visual representation of the project's evolution is presented in Figure 5.13.

OpenVZ metrics	2^{nd} **point**	3^{rd} **point**	5^{th} **point**	9^{th} **point**
month of life(month)	18 (2006-09-01)	31(2007-10-01)	26 (2007-05-01)	28(2007-07-01)
code added	5 277 281	6 812 835	4 749 627	3 395 710
code removed	2 530 006	4 203 349	2 576 007	2 897 968
comments added	1 003 888	1 412 474	843 530	611 152
comments removed	410 235	810 728	400 821	600 354
blanks added	740 500	1 142 649	758 198	508 052
blanks removed	228 259	581 321	340 077	378 180
contributors	500	642	637	649
commits	12 786	23 146	17 359	19 042
predicted commits	3 059	1 006	759	1 132

Table 5.4: Most probable noisy points from the OpenVZ project

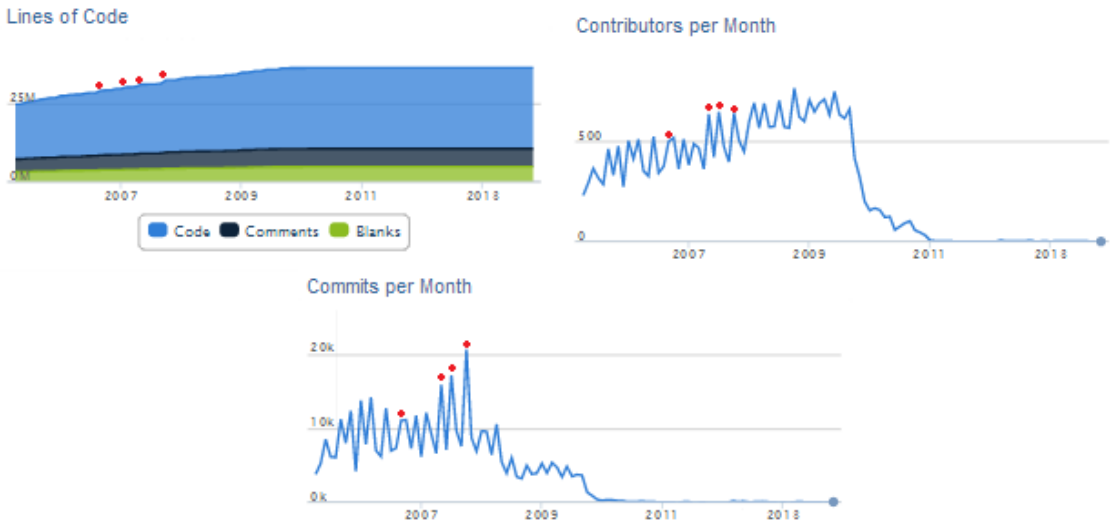

Figure 5.13: OpenVz graphs from Ohloh

Discussion

For the OpenVZ project, the instances that are highlighted are peaks in the number of contributors and commits, while having a high number of code added and removed. The biggest variance for commits is for the month of October 2007 when we have 20,680 commits while in the next and previous months there are only 7,590 and 8,760.

We further inspected the release history [2] near the reported months (marked as bold in the table), also shown in Table 5.5, and noticed some correlations to

[2] `http://freecode.com/projects/vzkernel/`

month	date of release	release version	branch	comments
April 2008	11 Apr 2008	2.6.18-53.1.13.el5-028stab053.10	RHEL5 2.6.18	Minor bugfixes
March 2008	26 Mar 2008	2.6.24-ovz004.1	2.6.24	Minor bugfixes
	01 Mar 2008	2.6.24-ovz002.2	2.6.24	Initial freshmeat announcement
February 2008	15 Feb 2008	2.6.18-028stab053.6	2.6.18	Major security fixes
January 2008	18 Jan 2008	2.6.18-53.1.4.el5-028stab053.4	RHEL5 2.6.18	Major security fixes
	18 Jan 2008	2.6.18 028stab053.4	2.6.18	Major security fixes
December 2007	04 Dec 2007	2.6.18-53.el5 028stab051.1	RHEL5 2.6.18	Minor security fixes
	04 Dec 2007	2.6.18-028stab051.1	2.6.18	Minor security fixes
November 2007	17 Nov 2007	2.6.22-ovz005	2.6.22	Major feature enhancements
	17 Nov 2007	2.6.18-8.1.15.el5 028stab049.1	RHEL5 2.6.18	Minor bugfixes
	17 Nov 2007	2.6.18-028stab049.1	2.6.18	Minor bugfixes
	01 Nov 2007	2.6.18-8.1.15.el5 028stab047.1	RHEL5 2.6.18	Minor security fixes
	01 Nov 2007	2.6.18-028stab047.1	2.6.18	
October 2007	01 Oct 2007	2.6.18-8.1.14.el5 028stab045.1	RHEL5 2.6.18	Major security fixes
	01 Oct 2007	2.6.18-028stab045.1	2.6.18	Major security fixes
	01 Oct 2007	023stab044.11	RHEL4-2.6.9	Major security fixes
September 2007	11 Sep 2007	2.6.22-ovz003.1	2.6.22	Initial freshmeat announcement
July 2007	30 Jul 2007	2.6.18-028stab039.1	2.6.18	Major bugfixes
	30 Jul 2007	2.6.18-8.1.8.el5 028stab039.1	RHEL5 2.6.18	Major bugfixes
June 2007	14 Jun 2007	2.6.20-ovz007.1	2.6.20	Minor bugfixes
	13 Jun 2007	2.6.18-8.1.4.el5 028stab035.1	RHEL5 2.6.18	Major security fixes
	13 Jun 2007	2.6.18-8.1.4.el5 028stab035.1	RHEL5 2.6.18	Major security fixes
	13 Jun 2007	2.6.18-028stab035.1	2.6.18	Major security fixes
	04 Jun 2007	2.6.18-028stab033.1	2.6.18	Minor bugfixes
May 2007	29 May 2007	2.6.9-023stab044.4	RHEL4-2.6.9	Minor bugfixes
	02 May 2007	2.6.9 023stab043.2	RHEL4-2.6.9	Minor bugfixes
	02 May 2007	2.6.18-8.el5 028stab031.1	RHEL5 2.6.18	Minor security fixes
	02 May 2007	2.6.18-028stab031.1	2.6.18	Minor security fixes
...				
September 2006	28 Sep 2006	2.6.16-026test018.1	2.6.18	Minor feature enhancements
	28 Sep 2006	2.6.8-028stab078.21	RHEL4-2.6.9	Minor bugfixes

Table 5.5: Release history of the OpenVZ project

our activity graphs. First, we will discuss the three months that are close to each other, from 2007: *May, July, October*. This period is with many peaks and we suppose one of the reasons could be the "Major feature enhancements" release from November 17. Only four releases were tagged as major feature releases, on the dates: 17 Nov 2007, 02 Nov 2006, 19 May 2006 and 20 Apr 2006. We can presume that for such a release, the development team did not work only 17 days in November, but most likely worked the previous months and reserved some time for testing in the last period. This would explain the highest peak from October, when besides the 3 major security releases, many of the new features were probably developed. November, the month of the big release, was actually a quite month; less development was probably done, more testing and maybe the developers took a short period to plan or relax after finishing the big sprint. We notice that after this release the project started dying also, the commits decreased rapidly and the number of releases decreased also.

August was a month of holidays, most likely, since no releases were planned. This would explain the high activity from July when many people finished up things before taking their vacation. We could not really explain May, even with the release history available.

September 2006, is the most probable noisy month of this project. We can see that at the end of the month, 28 September, a minor feature release was launched.

Besides normal updates, fixes as other releases have, this one mentioned in its notes two developments of interest: " updates from the stable branch" and "a mainstream update up to 2.6.16.29. Code cleanups from sparse". These merges and cleanups could explain the increased code added and removed, for the small number of commits and contributors.

We could not find more information from the OpenVZ repositories but from these findings we can see some interesting activities in the reported months. If our purpose would be to predict a project's death, they should be kept in the dataset, maybe they should even be highlighted, since the reported month of October 2007 could be consider the month after which this project started dying. Otherwise, if we would want to measure productivity, we should think more if the projects we want to measure productivity for are similar to this one? These points could be considered outliers, in some cases.

KDE project

The KDE project has 4 points in the top 10 most probable noisy points. The values of these points are in Table 5.6 and the evolution is presented in Figure 5.14.

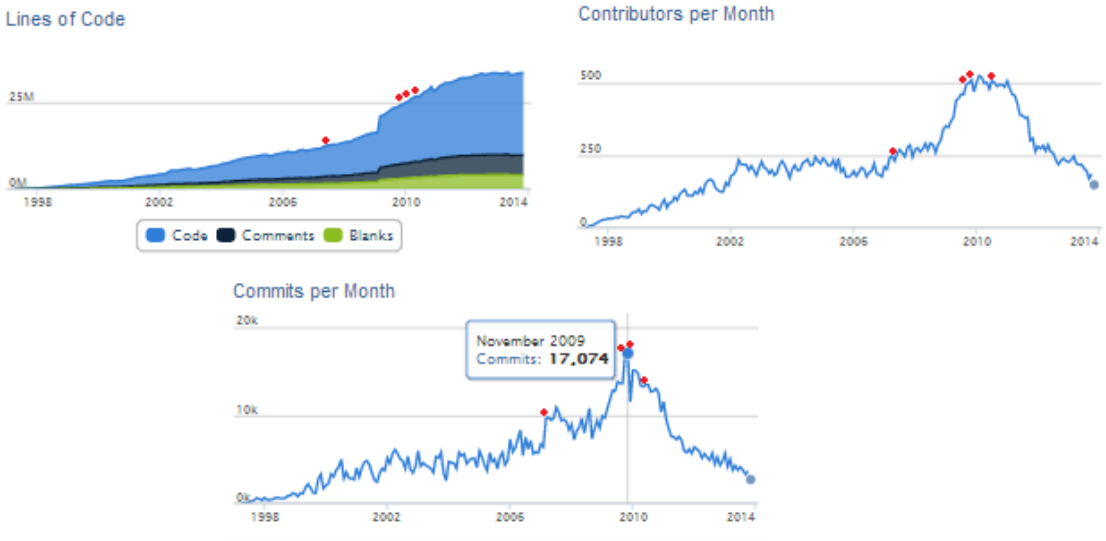

Figure 5.14: Kde graphs from Ohloh

KDE metrics	6^{th} **point**	7^{th} **point**	8^{th} **point**	10^{th} **point**
month of life(month)	151 (2009-10-01)	152(2009-11-01)	155 (2007-05-01)	157(2010-04-01)
code added	3 389 734	5 997 357	11 771 004	6 561 374
code removed	2 372 109	3 107 546	4 541 292	4 287 693
comments added	1 017 856	1 574 175	2 364 451	1 415 746
comments removed	666 669	1 075 331	1 045 022	954 298
blanks added	490 359	897 508	1 651 042	886 058
blanks removed	313 872	507 238	650 380	496 462
contributors	501	509	523	498
commits	18 797	19 205	18 827	16 312
predicted commits	288	127	1 254	499

Table 5.6: Most probable noisy points from the KDE project

We further inspected the release history [3]. The table bellow presents the latest releases, covering the dates we are interested in. Starting with release 4.1 they have a similar pattern. Each new major release starts with by freezing different parts of the system, tagging versions and releasing versions starting with Beta releases and ending at the X.X.5 release. The older versions are not the same, being longer and less structured.

[3]http://techbase.kde.org/Schedules

release	start month	end month
4.12	October 2013	April 2014
4.11	May 2013	Jan 2014
4.10	October 2012	July 2013
4.9	May 2012	January 2013
4.8	October 2011	July 2012
4.7	April 2011	December 2011
4.6	October 2010	July 2011
4.5	April 2010	January 2011
4.4	October 2009	June 2010
4.3	April 2009	January 2010
4.2	October 2008	June 2009
4.1	April 2008	January 2009
4.0	April 2007	January 2008
3.5	August 2005	August 2008

Table 5.7: Release dates for KDE project

Discussion

For the KDE project, the first 3 suggested noisy points are as for the OpenVZ project peaks in the measurement. The variations for these months are high having up to 11 million LOC added per month. The fourth point does not stand out on the graph but looking at the table 5.6, we can see that 6M LOC were added that month and 4M were removed.

For the 4.0 release of the KDE, *May 2007* started by freezing the "kdelibs Soft API" on the 1st. On the 3rd an alpha release was tagged, on the 8th we had the milestone "New Application Freeze and Usability and Accessibility Review" and on the 1st of June the milestone "Module Freeze". For the 3.5 release May 2007 had the following milestones: "Tagging KDE 3.5.7", "Expected release date of KDE 3.5.7". Thus May 2007 is a month where two release have overlapping activity.

October and *November 2009*, the highest peaks in the commits graph, had also activity for two overlapping major releases. For 4.3 the following milestones were set: "Tag KDE 4.3.2", "Release KDE 4.3.2", "Tag KDE 4.3.3", "Release KDE 4.3.3", "Tag KDE 4.3.4". For the release 4.4 we had the milestones: "Trunk depends on Qt 4.6, Soft Feature Freeze", "Hard Feature Freeze", "Message Freeze", "Tag KDE SC", "4.4 Beta 1".

In *April 2010* the 4.5 release started and the 4.4 release was finishing up.

Overall, we can see that there are some months every year where two release of the KDE are both active in development. All the highlighted points, are months like these. As for the OpenVZ points, we can not firmly classify these points as noise but we notice they are of interest. For the task of estimating the productivity, these points could be considered outliers.

Performance on small projects

By looking at the first 10 most probable points, we noticed that large projects are favored by our approach. Of course, if we put all the projects together from one person projects to big-renown projects, it is hard to create a model perfectly adaptable for all types.

After our first inspections, we now searched what are the smaller entries (that have less than 0.5 M LOC added) that rank higher as possibly having noise. The first entries were at positions 41, 57 and 60. The values are shown in Table 5.8 and the graphs for these projects are plotted in Figures 5.15, 5.16 and 5.17.

metrics	41^{th} **point, project html5lib**	57^{th} **point, project mifos**	60^{th} **point, project gaia-ajax**
month of life(month)	1(2006-12-01)	57(2010-12-01)	8(2009-06-01)
code added	10 508	210 234	32 193
code removed	7 470	154 380	25 687
comments added	1 907	36 705	5 249
comments removed	1 366	25 918	4 544
blanks added	1 365	26 375	5 514
blanks removed	802	15 382	4 284
contributors	5	19	4
commits	320	582	103
predicted commits	10 065	9 495	7 837

Table 5.8: Most probable noisy points from the KDE project

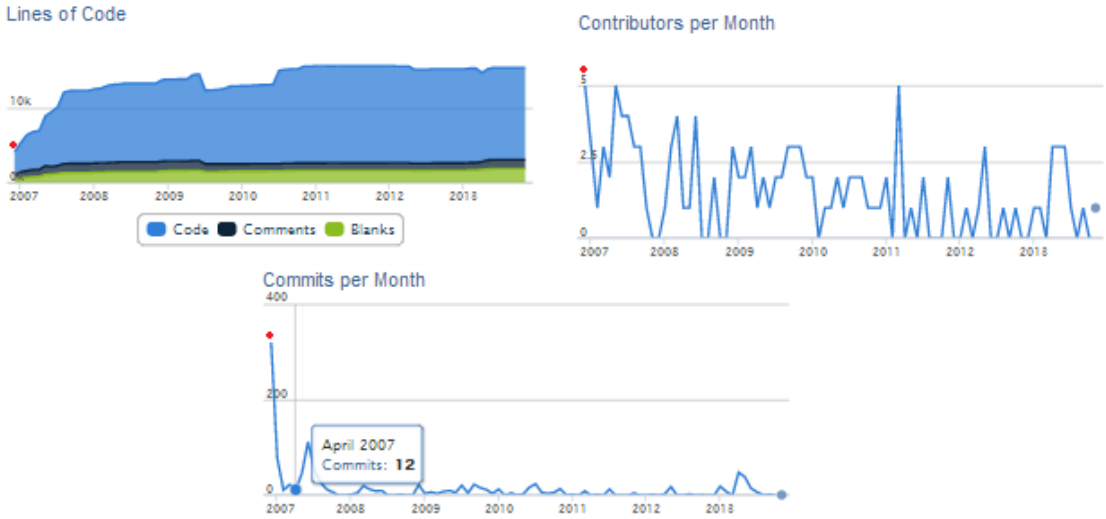

Figure 5.15: Html5lib graphs from Ohloh

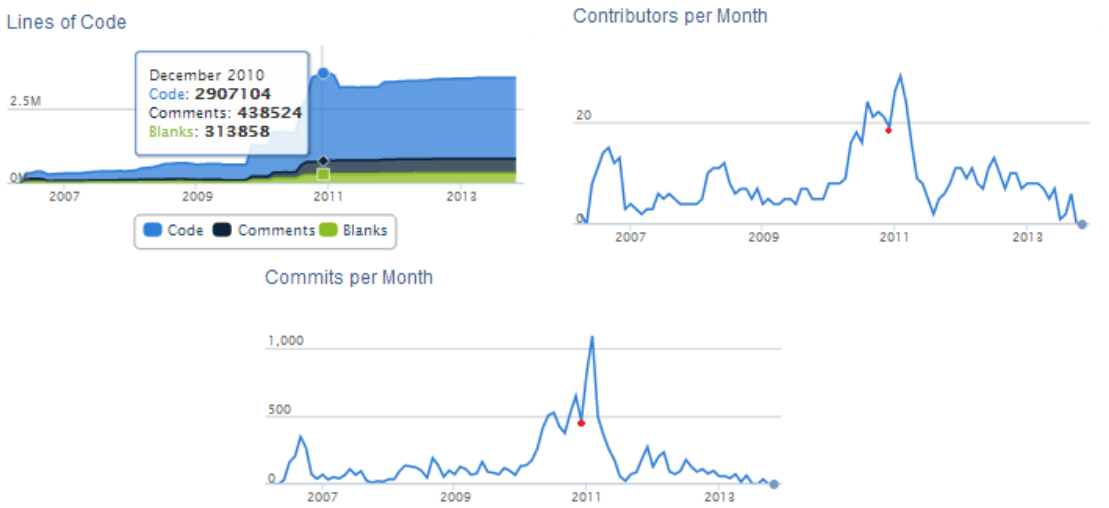

Figure 5.16: MIFOS graphs from Ohloh

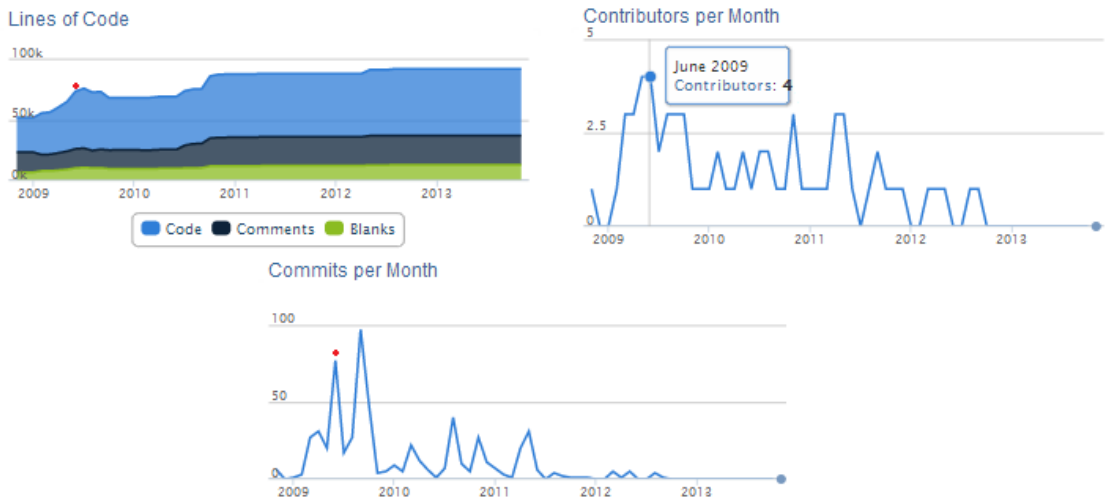

Figure 5.17: Gaia-Ajax graphs from Ohloh

Discussion

For these 3 projects, the suggested noise are peaks in the data. First point is an initial commit. The point from MIFOS is a peaks in the code added correlated with a decrease in commits and contributors. The instance from Gaia Ajax is a peak on all the three graphs.

In this cases, because the projects are also smaller, the excesses or uncorrelation are not that exaggerated as the point from ApacheFlex but are noticeable on the graphs. This shows that although the methods favours big project it also highlights interesting instances on smaller/medium projects.

How do lower ranked points look like?

Further, we investigated how points that are not highly ranked look like? Until now, we noticed that the GPs detect peaks in the activity data, which could be considered outliers. We inspected the first 10 points plus three other points for smaller projects and peaks in the data have been detected. Nevertheless, since we have little information regarding the dataset itself, we verified if our ranking is consistent. In the extreme and unreal case, the part of the data that we have not seen could be only peaks. In this case we did not detect any abnormalities, we just sampled normal instances, very likely to exist in our data. Therefore, we wanted to approximate when would the peaks in data stop and would the next points be less outstanding if we look at the graphs. This would suggest that we have a consistent ranking for our algorithm, the peaks are on top of the ranking while the other points are at the bottom.

Figure 5.18 shows the log probability of the first 2,500 instances from the activity results. We can see from it that the first points have a much lower log probability and it increases in big steeps, while after 500 points until 1,000, the log probability increases more slowly. Therefore we decided to inspect points starting with 1,000. If we would find that after 1,000 points the instances are still peaks, we could suspect that the data might actually contain many peaks.

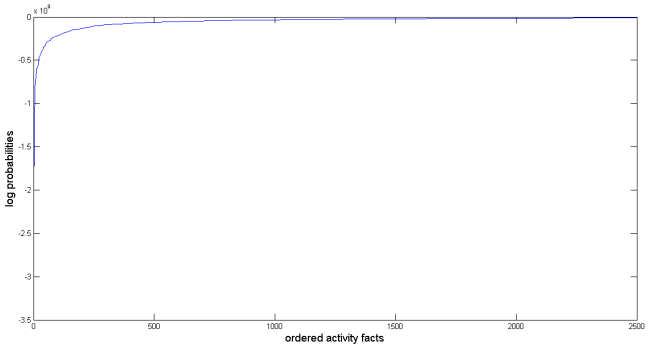

Figure 5.18: Gnome graphs from Ohloh

Project	**month**	**commits**	**predicted commits**	**after inspection**
1000. Gnome	2010-11-01	4377	990	not a peak
1001. LinuxACPI	2008-04-01	5460	2395	not a peak
1002. linux-davinci	2009-12-01	5727	2753	not a peak
1003. linux-davinci	2006-01-01	4000	386	not a peak
1004. openinkpot	2009-05-01	3744	219	not a peak

Table 5.9: Instances 1000 to 1004 from the activity results

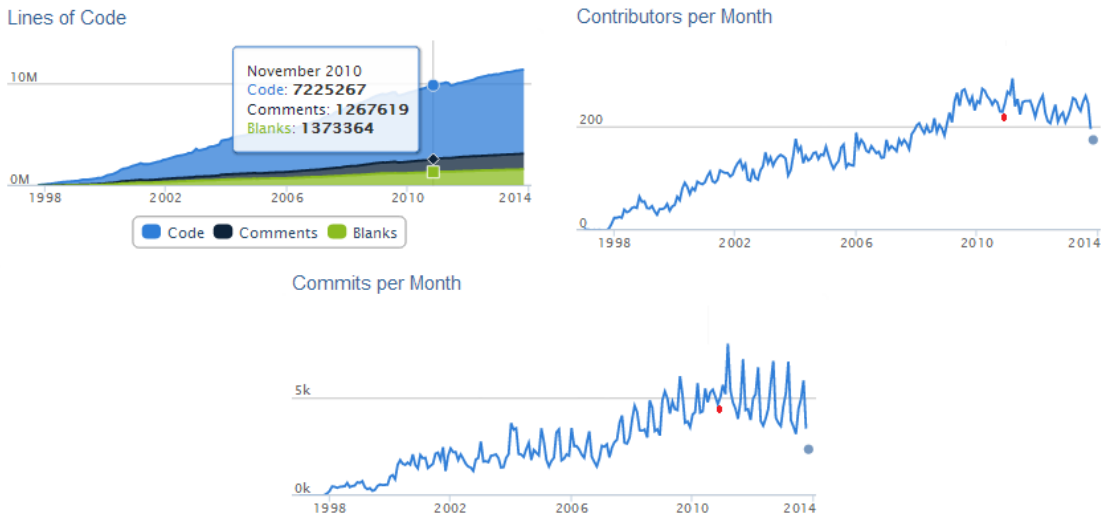

Figure 5.19: Gnome graphs from Ohloh

Discussion

Table 5.9 shows what project are found after the first 1000 most highly noisy points. Figure 5.19 shows the point that was highlighted from Gnome(the other images are similar). As we can see, these points are not peaks and according to the information we have, should not be considered noise. After this inspection we can have more trust in our ranking.

Prediction accuracy

Our method is based on making predictions about the *commits* per month according to the other attributes. We evaluate in this section our prediction capabilities by calculating the *absolute difference between our prediction and the real value*. In Figure 5.20, on the X axis we vary the difference between prediction and real value from 1 to 1,000 and on the Y axis we plot the percentage of points that have a smaller difference.

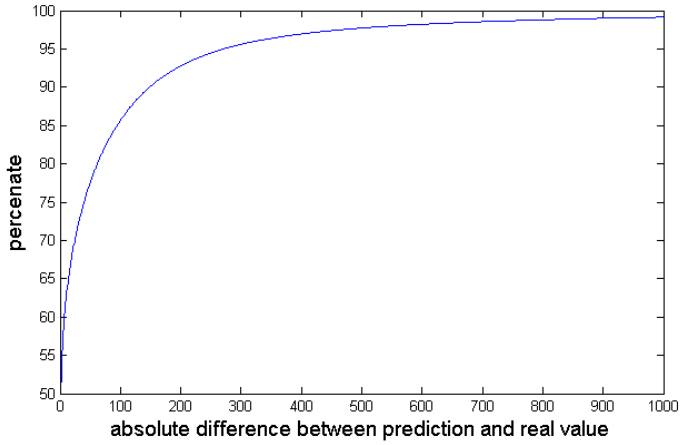

Figure 5.20: Prediction accuracy for activity facts

Discussion

On the figure we see that our of 889 117 instances we can predict around 50% of them with less than 1 commit difference. This high starting percentage is given mainly by the 403 760 measurements that have zero commits and zero for the other attributes. Nevertheless, our predictions have a difference from the real value between $(-1, 1)$ for 446 676 and between $(-10, 10)$ for 555 938 points. 75% of the points are predicted with a absolute difference smaller than 40 commits. Based on these we could say that we have leaned a good model for the activity data using Gaussian Processes.

Conclusions for the activity dataset

In this section we tested how Gaussian Processes perform on data formed of activity facts from 12,360 projects from Ohloh. This generated a set of 889,117 monthly measurements for projects.

Since we do not know in advance if the instances are noise or not, we manually judge the results. We consider two possible purposes to decide on the quality of points: predicting a project's death or estimating the general productivity per person, in a project.

From the results on the activity dataset we can draw the following conclusions

- **It is hard to assess the quality of reported instances** Even when setting up a purpose, it was hard to estimated the quality of instances without knowing what actually happened in that months of development. This was much easier for Rascal, were we had access to history logs and developers of the projects. Interpreting the quality of the points was driven by judging their visual evolution. Peaks in projects suggested a higher change of having a noisy point.
- **We were able to model the activity data using Gaussian Processes**: By manually sampling points from the results, we observed that they were sorted in a meaningful way (peaks at the beginning and not at the end) and the predictions that were make are good considering the heterogeneity of the data.
- **We were able to find noise in the activity data** For the purpose of estimating productivity we found points that are noise in the first 10 entries that we inspected.
- **Our model favors big projects** By leaving all the projects together in the dataset, we created a model that favors the bigger ones. If we desire to have a more refined analysis of projects, we could either penalize big projects or we could split the projects in categories based on size (or other attributes) and create a model for each of the groups. Now, most of the first instances were from big projects, smaller ones started appearing only at position 41. Nevertheless, the highlighted months for the small instances appeared as peaks in the graph.

Overall, we consider that Gaussian Processes performed well on the activity data, reporting interesting points that for some purposes should be considered noise and excluded from the dataset. Having an additional classifier that would, for example, predict a project's death could help us to better evaluate the performance of our approach in cleaning the Ohloh data for this specific purpose. We can compare the performance of the classifier before and after removing the most highly ranked noise from the data. A classification improvement would reflect a good cleaning procedure.

5.3.3 Size facts results

Running our method on the size facts data, started with diverse challenges. We were obtaining very inaccurate predictions or errors, which led us to finding several errors in the data:

- 110 projects have *NaN* values for *comments ratio*.
- 70 projects have negative measurements. The projects that have *-Infinity* for comments also showed that negative values could be in our data. When checking, we found that 70 projects have a total of 1432 instances with negative measurements.
- 2 entries from our training data created numerical problems. These are from the *ddwrt* project, the biggest in our dataset with 129 million LOC. The entries are for the *code added* metric for the months of March 2009 and June 2010 were 163,976,569 LOC respectively 258,756,226 LOC were added. Although we can not say these are noise, we had to remove them from the training data in order for our method to be able to make calculations. If a different samples of training data are to be randomly selected, metrics that are as high as 100 million should be excluded from the the set.

The instances that had *-Infinity* or *NaN* for *comment ratio* were removed from the array.

We first plotted the histograms for the training set and all the data. We see that around 1100 instances have zero total commits. This is similar to activity facts (but there we zoomed in). But, in this case we notice the max number of commits is very high. This is because in our training data we have a size fact from with 341 402 commits, from the project *django sorting*. This number of commits is much higher than the rest.

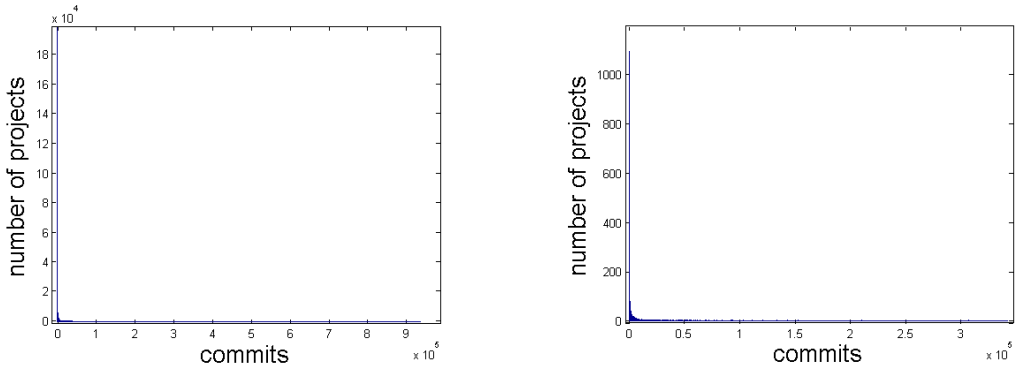

Figure 5.21: (Left) Histogram of all commits form the size array. (Right) Histogram of all the commits form the training set of the size data

The results, for the size activity, showed the limitations of modeling this data using Gaussian Processes. In the first 300 point all except two had more than

5 million LOC and those two had 4.8 million LOC. The first 59 highest ranked instances were measurements from KDE. They are the months from *September 2009* until the last one measured, July 2013. If we recollect the activity results, we remember that in the same period we had two suggested outliers in the months of October and November 2009. These point and the next one in April 2004 are still marked on the graphs from Figure 5.22. We see therefore that in that short period we have two outliers detected in the activity facts and we also start reporting oultiers in the size data.

These results are understandable. Size facts measure totals of the metrics, from when the project started. Activity reports only the current month metrics. For activity facts, we could have one abnormal month and the next ones normal. But for size facts, once a month disrupts the normal correlation between inputs and outputs this is maintained as long as there is no correction. To correct, probably a peak in the different direction is needed. E.g. if we add a lot of code with few commits; to correct, next month we should add few code with many commits. This is not the case for KDE, since instead of having corrections it actually has more outlies next months, according to the GPs.

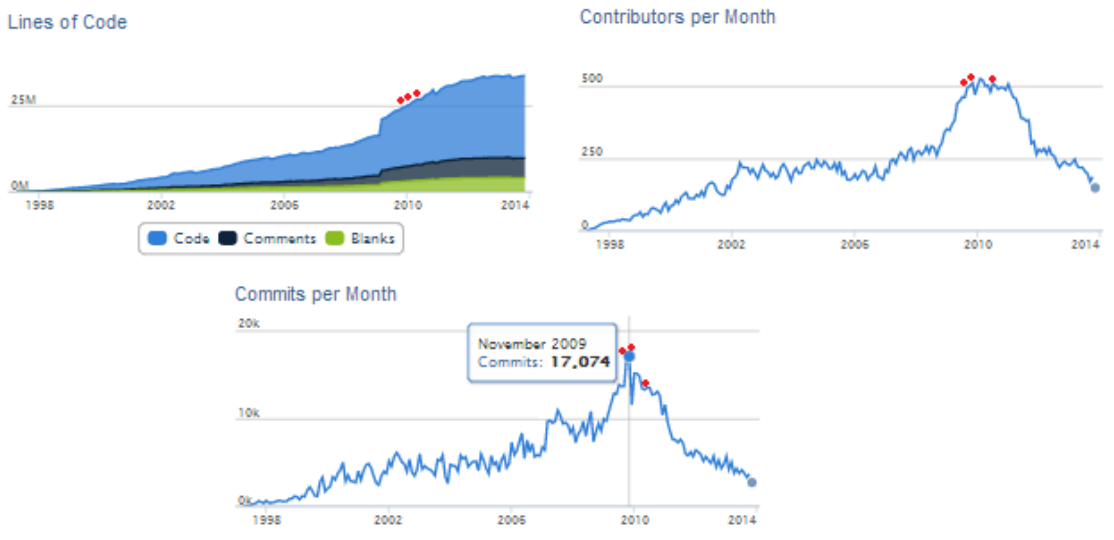

Figure 5.22: KDE graphs from Ohloh

As seen from Figure 5.23 our prediction accuracy also dropped significantly, comparing with the activity facts. This could also be partly explained by the bigger range of possible values.

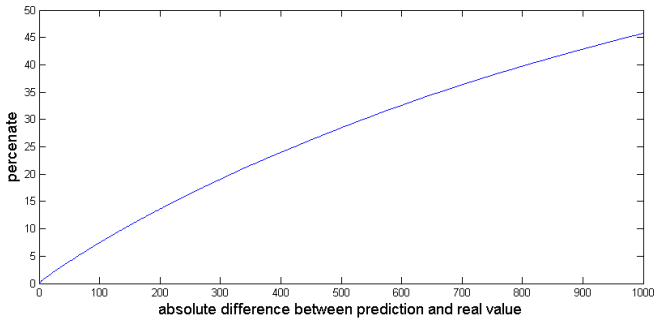

Figure 5.23: Prediction accuracy for size facts

Conclusion

Modeling the size facts does not give the same good results as the model for activity facts. We believe that the most likely reason, is the data itself. As before, we have projects of totally different sizes, but in this case we measure the totals for each project not the activity for each month. Some projects end up having very high values for their attributes and once they do that it will remain constantly like that unless a peak in the opposite direction appears in the next months.

There could be other explanations for our results on the size facts, but they are less likely. One issue that might cause problems is us working with big numbers. We already had numerical issues when training. After removing the two instances, our algorithms executed but the difference between metrics is still very high therefore our precision might become low. Also, as we noticed from the training data histograms, we have an big outlier in our training set. The point from *djanog sorting* has, the highest number of commits from all the projects in the train data and only few others have even close to him. This might impact negatively the learned model.

How to model the size facts correctly, remains an open question of our research. A possible approach could be to learn a model for individual projects and calculate the log probability of the points according to the model. To increase our numerical stability and prevent the errors that occurred in our settings, we could normalize the data.

CHAPTER 6

Conclusions

The literature review performed by Liebchen [9], concludes that not many publications in the empirical software engineering address the problem of data quality and a minority of papers even discuss it. Nevertheless, "a substantial majority of papers, 138 out of 161, considered data quality to be a threat to analysis of empirical data" [9].

In this thesis we researched how the quality of data can be asses using Machine Learning techniques. We chose Gaussian Processes, an approach that has not been used for noise or outlier detection until now, to our knowledge. We performed tests on two types of datasets and situations.

First, we used the Housing dataset and induced noise in it. Our method performed very well on this data, having high recall and good false positive rate. Moreover, we showed different GPs configurations that could improve the performance in particular situations.

Second, we tested Gaussian Processes on two real world datasets obtained from Ohloh, of which we had no prior knowledge or known types of noise. The results from the first data, the activity facts, showed that our method suggests interesting points that could be considered noise, depending on the purpose. The task of manually inspecting the suggested noise points and deciding on their true label was very hard. To make stronger statements about the success of our approach we could further test it in other settings as described in the future work section.

The size facts data raised difficulties from the beginning. Many noise values were found before we could even run the Gaussian Processes script, from negative numbers, NaN or $-Infinity$ values, to extremely large commits that were causing numerical errors. After removing these, our technique was not able to learn a good model. We believe that the different project sizes and the particularities of this data, made it hard for our method to learn a meaningful model. Future improvements are suggested in the previous chapter and in the future work.

6.1 Contributions

This section lists the contributions of our work:

- **Evaluate the Gaussian Processes technique for noise detection**.

  From we know, GPs were not used for noise or outlier detection. The experiments show GPs are capable to detect artificially induced noise but also highlight suspicious noisy points from real data. On the Rascal project experiment the point ranked most probable to be noise, was actually noise and on the activity data, several outliers were detected which could be considered noise, depending on the purpose for which we use the data. GPs showed their limitations on the size facts data.

- **Model real world dataset**.

  Many researchers validates their approach only on artificially induced noise. The advantage of this approach is that it can offer a clear benchmarking methodology. On the other hand, this has two limitations. First, we do not know if the data we inject noise into does not already have noise. Second, the noise is added according to a distribution which might not reflect accurately the real word environment.

  We also tested our approach on two real world datasets obtained from Ohloh, with the results described above.

- **Manage large datasets**

  The Ohloh datasets have more than 800,000 entries each, which imposes some restrictions on our methodology. To adapt, we trained our model by using only on a subset of the data.

- **Assess the quality of the Ohloh dataset**

  Data quality is hard to define for a certain instance. This was the case even on artificially induced noise, that created only a small variance from the original point. In the Ohloh dataset assessing the quality of a point was even harder, as we had limited ways of understanding what happened years ago on certain projects.

  Nevertheless, several errors were found on the Ohloh dataset and are mentioned in Chapter 5, which raise questions regarding their measurements. Some of the points highlighted by our approach could be considered noise, depending on the purpose.

6.2 Limitations and future work

- **Group projects by size or other characteristics**. By splitting the entire Ohloh data, we can create better models, more specific for the group of interest. This should improve our noise detection performance.

- **Filter only features of interest**. At first, we were running our experiments on an old version of the activity data. We did not used all the 8 features in order to predict commits but only the code added and removed, the comments added and removed. The results differed and a higher number of unique projects had instances in the highest ranked noisy points. We could further research, if we are only interest in the quality of a subset of features, would it be better to consider as inputs only those attributes or all.

- **Use meta-techniques from Machine Learning**. We can use bagging or voting techniques on top of our Gaussian Processes. These presume learning more models on different pieces of the data or using different configurations for GPs (or even other algorithm), and then deciding if a point is noise or not taking into account the judgment of all of them.

- **Better validation of the current approach**. The lack of pre-defined labels for our data made validation a hard task. Gamberger in [12] showed that by removing noise from the data he improved the accuracy for detecting coronary artery disease. To better judge the quality of a point and improve our validation it would be useful to have a system that predicts project failures or makes other estimations. Then, we could evaluate our performance on detecting noise by measuring its change in performance. Such a system is not available in our case would first need to be built.

- **Experimenting with more covariance functions or inference methods**. In our current approach we tested a few combinations on the Housing data, but many more combinations could be formed and tested on the Ohloh data.

- **Take recent history in consideration** Currently time is taken into consideration, but only by adding the month of life for each project measurement. This creates a weak coupling between the measured metrics and the evolution of a project. Stronger connections can be made by adding the previous or even future month to each current month's feature vector. E.g. For the activity facts we can extend the input vector for each point from 9 inputs to 27 and add for the 10^{th} month of life the measurements from the 11^{th} and 9^{th} month.

- **Vary the training set** In our experiments we used only one small piece of the data from Ohloh for training. Ideally, in our case, we would have trained on a bigger quantity of data or even all of it, but this was computationally

unfeasible with our methodology. More experiments can be done with approximation methods or more of the data could be used for training with meta-techniques.

- **Systematic errors**. We do not believe that a "silver bullet" technique is possible. If an error systematically appears in our data, the GPs model might learn it as "normal" and not highlight these instances as noise. Therefore, we are not expecting our approach to detect systematic errors, in the current methodology. Generating more models on smaller training datasets and applying voting procedures to decide on the points label could be a solution.

Bibliography

[1] Max Meier, *The Final theory*, Springer 1999

[2] Phillip Boyle, *Gaussian Processes for Regression and Optimisation*, PHD Thesis, Victoria University of Wellington, 2007

[3] T. Mitchell *Machine Learning*, McGraw Hill, 1997.

[4] C. E. Rasmussen. *Evaluation of Gaussian Processes and other Methods for Non-linear Regression*, PhD thesis, Department of Computer Science, University of Toronto, 1996.

[5] UCI Machine Learning Repository, *Housing dataset*, `http://archive.ics.uci.edu/ml/datasets/Housing`

[6] Open source network,*Ohloh*, `http://www.ohloh.net/`

[7] B.W. Boehm, *Software engineering economics* ,Prentice Hall, 1981

[8] T.M. Khoshgoftaar, J.C. Munson, B.B. Bhattacharya, G.D. Richardson, *Predictive modeling techniques of software quality from software measures*, Software Engineering, IEEE Transactions on , vol.18, no.11, pp.979,987, Nov 1992

[9] G.A. Liebchen, *Data Cleaning Techniques for Software Engineering Data Sets*, PHD thesis, Brunel University, 2010

[10] G.A. Liebchen, M. Shepperd, *Data sets and Data Quality in Software Engineering*, International Conference on Software Engineering, ACM 2008

[11] Dragan Gamberger, Nada Lavrac, Sao Deroski, *Noise elimination in inductive concept learning: A case study in medical diagnosis*, Springer Berlin Heidelberg, 1996

[12] Dragan Gamberger, Nada Lavrac, and Ciril Groselj, *Experiments with noise detection algorithms in the diagnosis of coronary artery disease*, In IDAMAP-98, Third Workshop on Intelligent Data Analysis in Medicine and Pharmacology, 1998

[13] S. Gala, G. Robles, and J. M. Gonzalez-Barahona, *Intensive Metrics for the Study of the Evolution of Open Source Projects: Case Studies from Apache Software Foundation Projects*, 2013.

[14] L.C. Briand, V.R. Basili and W.M. Thomas, *A Pattern Recognition Approach for Software Engineering Data Analysis*, IEEE Trans. Software Eng., vol. 18, no. 11, pp. 931-942, Nov. 1992.

[15] Martin Shepperd, *Data Quality: Cinderella at the Software Metrics Ball?*, Proceeding of the 2nd international workshop on Emerging trends in software metrics, May 24-24, 2011, Waikiki, Honolulu, HI, USA

[16] Dragan Gamberger, Nada Lavrac,Sao Deroski, *Noise elimination in inductive concept learning: A case study in medical diagnosis*,Book Section: Algorithmic Learning Theory, 1996

[17] Martin Shepperd, Qinbao Song, Zhongbin Sun, and Carolyn Mair, *Data Quality: Some Comments on the NASA Software Defect Datasets*, IEEE Transactions on Software Engineering, vol. 39, no. 9, pp. 1208-1215, Sept., 2013

[18] Kim Shunghun, *Adaptive bug prediction by analyzing project history*, PHD Thesis, University of California, Sept. 2006

[19] George H. John. *Robust decision trees: Removing outliers from databases*, 1995.

[20] G. Liebchen, B. Twala, M. Shepperd, *Filtering, Robust Filtering, Polishing: Techniques for Addressing Quality in Software Data*, Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on , vol., no., pp.99,106, 20-21 Sept. 2007

[21] T.M. Khoshgoftaar, J.C. Munson, B.B. Bhattacharya, G.D. Richardson, *Predictive modeling techniques of software quality from software measures*, Software Engineering, IEEE Transactions on , vol.18, no.11, pp.979,987, Nov 1992

[22] R. De Veaux and D. Hand, *How to Lie with Bad Data*, Statist. Sci., vol. 20, pp. 231-238, 2005.

[23] Carla E. Brodley, Mark A. Friedl, *Identifying mislabeled training data*, Journal of Artificial Intelligence Research (JAIR), 11:131-167, 1999.

[24] T. M. Khoshgoftaar, N. Seliya, and K. Gao, *Rule-based noise detection for software measurement data*,In IRI, 2004.

[25] T. M. Khoshgoftaar and J. D. Van Hulse, *Identifying noise in an attribute of interest*, In ICMLA '05: Proceedings of the Fourth International Conference on Machine Learning and Applications (ICMLA'05), pages 55-62, Washington, DC, USA, 2005. IEEE Computer Society

[26] J. D. Van Hulse, T. M. Khoshgoftaar, and H. Huang, *The pairwise attribute noise detection algorithm*, Knowledge and Information Systems, 11(2):171-190, 2007

[27] International Software Benchmarking Standards Group, Isbsg.

[28] C. E. Rasmussen, H. Nickisch, *Gaussian Processes Matlab package*, `gaussianprocess.org`

[29] C. E. Rasmussen, and C. Williams, *Gaussian Processes for Machine Learning*, The MIT Press, 2006

[30] R. Premraj, M. Shepperd, B. Kitchenham, P. Forselius, *An empirical analysis of software productivity over time*, Software Metrics, 2005. 11th IEEE International Symposium , vol., no., pp.10 pp.37, 1-1 Sept. 2005

[31] Thomas C. Redman,*The impact of poor data quality on the typical enterprise*, Commun. ACM 41, 2 February 1998, 79-82, doi: http://doi.acm.org/10.1145/269012.269025

[32] Yair Wand and Richard Y. Wang, *Anchoring data quality dimensions in ontological foundations*, Commun. ACM 39, 11 November 1996, 86-95, doi: http://doi.acm.org/10.1145/240455.240479

[33] S.L. Lauritzen, *Time series analysis in 1880, a discussion of contributions made by T.N. Thiele*, ISI Review 49, 319 333

[34] N. Wiener, *Cybernetics*, Wiley, 1948

[35] D. J. C. MacKay, *Introduction to Gaussian Processes*, In C. M. Bishop, editor, Neural Networks and Machine Learning, pages 133-165. Springer, 1998.

[36] R.M. Neal, *Bayesian Learning for Neural Networks*, Lecture notes in Statistics, Springer, New York, 1996

[37] D. J. C. MacKay, *Gaussian Process Basics*, University of Cambridge, June 2006, `http://videolectures.net/gpip06_mackay_gpb`

[38] C.E. Rasmussen, *Gaussian Processes in Machine Learning*, Advanced Lectures on Machine Learning, Lecture Notes in Computer Science 3176. pp 6371. doi:10.1007/978-3-540-28650-9

[39] Gwenn Englebienne, *Kernel Methods*, Machine learning pattern recognition lecture at University of Amsterdam, 2011

[40] Edward Lloyd Snelson, *Flexible and efficient Gaussian process models for machine learning*, PHD Thesis, University of London, 2007

[41] Miguel Lzaro-Gredilla, Joaquin Quionero-Candela, Carl Edward Rasmussen, Anbal R. Figueiras-Vidal; *Sparse Spectrum Gaussian Process Regression*,Journal of Machine Learning Research, 11(Jun):18651881, 2010.

[42] Carl Edward Rasmussen, Zoubin Ghahramani, *Innite Mixtures of Gaussian Process Experts*, In Advances in Neural Information Processing Systems 14, Cambridge, MA, 2002. MIT Press. 7.1

[43] W. Tang, T.M. Khoshgoftaar, "*Noise identification with the k-means algorithm*, Tools with Artificial Intelligence, 2004. ICTAI 2004. 16th IEEE International Conference on , pp.373,378, 15-17 Nov. 2004, doi: 10.1109/ICTAI.2004.93

[44] J.C. Barca, G. Rumantir, *A Modified K-means Algorithm for Noise Reduction in optical Motion Capture Data, 6th IEEE International Conference on Computer and Information Science, 2007, Melbourne*

[45] H.S. Behera, A. Ghosh, S. Mishra, *A New Hybridized K-Means Clustering Based Outlier Detection Technique For Effective Data Mining*, International Journal of Advanced Research in Computer Science and Software Engineering, vol. 2, Issue 4, April 2012

[46] V. Hodge, J. Austin, *A Survey of Outlier Detection Methodologies*,Journal Artificial Intelligence Review, Vol.22, Issue 2, pp 85-126, 2004

APPENDIX A

Source code explanations

Our source code was written in Matlab and versioned on bitbucket on a private repository. To request access to it please write an email to:
adrianv-vlad.lep@student.uva.nl

Importing the Ohloh data in Matlab

The xml files for the projects were mined from Ohloh by M. Bruntik and uploaded on the Git public repository:
`https://github.com/MagielBruntink/OhlohAnalytics/`,
on the *searhus* branch. The first 12 360 most active projects are in the collection. We have downloaded the latest version to date(December 2013), with the hash 7be4672128.

To import the data in Matlab the *main.m* should be run from the *datasets* folder. This script will pass through all the 12360 projects from the folder *datasets/projects* and read from *ActivityFacts.xml* and *SizeFacts.xml*. The output of this script will be an *activity_data* and *size_data* vector of structures. Besides these, it will also generate a vector *errors_data* which can contain three types of errors:

1. 426 projects have missing ActivityFacts file, e.g. AgiloforScrum, Avira, BootstrapTwitter, BranchingGuidance.
2. 433 projects have missing SizeFacts file, e.g.: darcs, darcsweb, darwinports.
3. 161 projects have empty SizeFacts file; the file contains no entries or only one entry that is 0. E.g.: asdsa, c-minc, bergamot.

These projects were removed from the dataset. Each of the two structures are then transformed to input arrays using the script *struct_to_array_activity.m*. While running this script we found 3 projects that have *-Infinity* for comments ratio in

the activity file. This is because the *code + comments* is 0 (zero) and Ohloh got a division error. These projects were removed from the size facts dataset: acegisecurity, appcelerator, jflex.

The projects that have $-Infinity$ for comments also showed that *negative values* could be in our dataset. We checked using the *find_negatives.m* script and found 70 projects having a total of 1432 entries with negative measurements e.g. FreeMat, LinuxACPI, UniOffice. These entrances

While training out GP on the size facts, we encountered several other errors that were stopping our algorithm. The cause of that turned out to be the existence of *NaN values* in our data, for *comments ratio*. There were 1827 instances with this problems from 110 projects, e.g.: bash-completion, FarseerPhysics, FreeCRM.

Generating the results

The main scripts for each of the two datasets are listed below. Besides these more functions were created to manipulate data, generate graphics or fulfill other tasks.

Housing data

For the housing data, several script were developed. The main scripts are:

- *housing_main.m* source code for GPs in default settings
- *housing_main_matern.m* source code for GPs with Matern covariance function
- *housing_main_positive.m* source code for GPs with Inverse Gaussian likelihood, for strictly positive numbers
- *housing_main_aproximation.m* source code for GPs with approximation methods for large datasets.
- *create_random_noise.m* adds random noise to the housing instances

Ohloh data

For the Ohloh data the main scripts are:

- *ohloh_activity_main.m* used to generate the results for the activity facts.
- *ohloh_size_main.m* used to generate the results for the size facts.
- *rascal_activity_main.m* used to generate the results for Rascal.

- *ohloh_activity_selection_main.m* used to generate results for a selection of the activity facts features. We selected only commits, code and comments added or removed.

To run any of the experiments, one of these files needs to be run. They are dependent on the functions in the folders *mean, lik, inf, util, cov* and *doc*, therefore we need to add them to the path before starting. This can be done from Matlab, by clicking right on the folder and selecting "Add to path".

Results for the Ohloh data

Two *.mat* files are saved in the repository. *ohloh_old.mat* is an older version of the data, from June which had around 10 000 projects. The newer version with more recent facts and more projects, 12 360, is saved in *ohloh_data.mat*. This *.mat* file contains all the data; inputs, results and variables used.

The complete sorted results are stored in the *results_activity* and *results_size* arrays from the*ohloh_data.mat* file.