# Purpose:

*To deliver a report which considers the ways in which the software engineering process can be measured and assessed in terms of measurable data, an overview of the computational platforms available to perform this work, the algorithmic approaches available and the ethics concerns surrounding this kind of analytics.*

# Introduction:

Most software engineers can sit down and write a small program with little planning or forethought. As programs become larger and more complex the need for a well-defined plan and process increases. A software engineering process is a set of related actions for managing the creation of software from initial customer conception up to the completion of the finished project. A software process usually includes the following activities:

* Software specification
* Software design and implementation
* Software verification and validation
* Software evolution/maintenance

In this report I will analyse the software engineering process. I will consider four main areas:

* The ways in which a software engineering process can be measured and assessed and the measurable data which can be used for this.
* The computational platforms available to perform this work.
* The algorithmic approaches used to collect and analyse said data.
* The ethical concerns surrounding these approaches.

# Measurable Data:

In today’s modern world, there is an infinite amount of data readily available about almost every topic imaginable. Data analytics is the process of examining data sets in order to draw conclusions about the information they contain. Business’s now realise that data science plays a fundamental role in solving problems and optimising their production. More and more data scientists are being employed for this reason. Their job is essentially to convert data into tangible information that can be understood and used to make decisions.

The very nature of software engineering makes measurement a necessity. Rigorous methods for production planning, monitoring, and control are needed, otherwise the level of risk associated with certain software projects may become excessive, and software production may easily get out of industrial control. Software engineering differs from other engineering disciplines for two main reasons. Firstly, software engineering is a young discipline, so its theories, methods, models, and techniques still need to be fully developed and assessed. Other engineering branches rely on older, well-consolidated scientific disciplines. Second, Software Engineering is a very human-intensive discipline, while other engineering branches are based on the so-called hard sciences. Therefore, some aspects of software measurement are more similar to measurement in the social sciences than measurement in the hard sciences.

This human intensive nature means that there are two components to consider when assessing a software engineer’s performance. The traditional or code-related data which relates to the properties of the engineer’s code and process. Then there is non-traditional or non-code related data such as how interactive and engineer is with other workers or how well an engineer takes on criticism.

**Traditional Data**

One difficulty associated with measuring the software engineering process is the huge amount of data that can be measured. It is impossible to analyse all of this data so we must identify and measure the characteristics of software processes and products that are believed to be relevant and should be studied. Some of the most important metrics used to assess a software engineer’s performance are:

1. Lead time
2. Code churn
3. Impact
4. Active Days
5. Efficiency
6. **Lead Time**

This refers to the time elapsed between identification of a requirement and its fulfillment. Lead time is essentially the amount of time between beginning a project and it being delivered to customers. Lead time data is extremely useful. Lead time history can be used to predict the completion time of future projects.

1. **Code Churn**

[Code Churn](https://blog.gitprime.com/why-code-churn-matters) is the percentage of a developer’s own code representing an edit to their own recent work. It’s typically measured as lines of code (LOC) that were modified, added and deleted over a short period of time such as a few weeks. The primary purpose of measuring churn is to allow software managers and other project stakeholders to control the software development process, especially its quality. When churn starts to spike, this can be an indicator that something is off with the development process.



In situations where a developer is given a very vague project outline such as “create settings for this app” this can turn into weeks of iteration on the same feature without a lot of forward progress. This will show up as code churn. Churn rate can also help to identify problems with individual developers. For example, a sudden increase in churn rate may indicate that a developer is experiencing difficulty in solving a particular problem or is repeatedly polishing a feature that’s ready for release. A high churn rate may also mean that a developer is under-engaged. Other causes of high churn include an indecisive product team that has the developer running in circles.

1. **Impact**

Impact measures the effect that changes made by an individual software engineer have on the project. Change sets that are more difficult to implement will therefore result in a higher impact score. The impact of a change set depends on a variety of factors such as the amount of code in the changes, the severity of those changes and the number of files that the changes affected.

1. **Active Days**

An Active Day is a day in which an engineer contributed code to the project, which includes specific tasks such as writing and reviewing code. Engineers are uniquely skilled at building and solving difficult conceptual problems, so contributing code is one of the most important things that an engineer can do. This data essentially tells us which engineers are the most active. The most active software engineer is not necessarily the best software engineer however, they are the hardest working software engineer. This type of data can be important to managers when making decisions about promotions in the workplace.

1. **Efficiency**

Efficiency is the percentage of an engineer’s contributed code that’s productive, which generally involves balancing coding output against the code’s longevity. Efficiency is independent of the amount of code written. The higher the efficiency rate, the longer that code is providing business value. A high churn rate reduces it. Different types of engineers will have different efficiency rates. A engineer that is trailblazing a new solution may try a lot of paths in the discovery phase, and a low efficiency rate may be expected. [The most prolific engineers](https://blog.gitprime.com/check-in-frequency-and-codebase-impact-the-surprising-correlation/) contribute lots of small commits, with a modest churn rate, resulting in a high efficiency rate. Understanding an engineer’s typical efficiency rate can help you understand their character and where they will fit in best.

These metrics can be used to assess a software engineer’s performance however they must be used in conjunction with non-traditional metrics in order to provide a more well-rounded assessment of said engineer.

###### **Non-Traditional Data**

The non-traditional data refers to how the human intensive nature of software engineering is measured. There are many non-code related factors which influence the performance of software developers.

In the paper “Network Effects on Worker Productivity” Matthew J. Lindquist and Jan Sauermann discuss how co-workers can exert economically significant effects on their peers. The group analyzed data from a multi-national mobile network operator to see how co-worker productivity impacts on worker productivity via network effects. They found that a 10% increase in average co-worker productivity produces a 1.7% increase in a worker’s own productivity. They attribute this productivity spillover to conformist behavior in the workplace.

Another paper “Measuring Happiness Using Wearable Technology” from the Hitachi Review looks at how happiness influences productivity. It has been found that people who are happy have 37% higher work productivity and 300% higher creativity. It has also been reported that companies with a large number of happy people have higher earnings per share. This clearly indicates that the happier a software engineer is the better their work and production will be. The problem is happiness cannot be quantified. This group are looking at quantifying happiness using wearable technology. This may be possible in the not so distant future however for now we must still look at happiness as something not strictly quantifiable.

These are two examples of non-code related factors which could greatly influence the performance of a software engineer. These examples highlight that there are aspects of software engineering that cannot be assessed purely on the metrics mentioned in the previous section. To focus on these metrics alone would be to say that software engineers are machines with no emotions or human instincts. These papers focus on the human intensive nature of software engineering. Measuring this human side of the software engineering discipline is going to become more and more important as technology advances.

Other non-code factors which should be considered when assessing the performance of a software engineer are their soft skills. Soft skills refer to personal attributes that enable someone to interact efficiently and harmoniously with other people. Managers when considering employees for promotions generally look for candidates with both hard and soft skills. A software engineer’s hard skills are measurable by the traditional code-metrics I have discussed. A software engineer’s soft skills refer to their interpersonal skills and are much harder to define and measure. Some soft skills that should be measured when assessing a software engineer are:

1. Communications: How well they communicate with other developers and management.
2. Emails: How often and accurately the engineers respond to and composes emails.
3. Meetings attended: How many meetings the software engineer attends and how they engage in these meetings.

Generally, people may think all software engineers need to succeed is to be highly component coders. While this is an essential skill, that alone does not an excellent software engineer make. A software engineer needs both hard and soft skills. A software engineer who has abundant hard skills but is a terrible communicator and doesn’t work well in a team is not some a hiring manager will be interested in. It is clear that there must be a combination of hard and soft data used to measure a software engineer’s performance.



# Computational Platforms available:

Having laid out what data should be used to assess the performance of a software engineer we must now consider what to do with this data. Computational platforms are environments of integrated software that have user-friendly interfaces and that can be used by researchers to properly analyze data. I will discuss the computational platforms used in the past to assess a software engineer’s performance as well as the computational platforms used today.

**Personal Software Process**

In a paper titled “Searching under the Streetlight for Useful Software Analytics”, Philip M. Johnson, University of Hawaii at Manoa discusses his team’s research into some of the ways of gathering data from software engineers. Since 1996 Johnson and his team of researchers at the Collaborative Software Development laboratory(CSDL) at the University of Hawaii have been looking into the analytics that help developers understand how they can improve their processes and the products they produce.

The team in Hawaii started by evaluating the Personal Software Process developed by Watts Humphreys. Humphrey first began work on the PSP when he applied Capability Maturity Model (CMM) principles to small programs. The CMM for software was introduced in 1987 and focused on the management of the software engineers and the help and assistance they received. Humphrey then sought to apply these principles to larger more complex programs with more engineers. He began working full time on the PSP in 1989 while he was a fellow at the Software Engineering Institute(SEI). He published his findings in a manuscript which he then sent to several colleagues. The first PSP course was given by Howie Dow in 1993 at the University of Massachusetts. Based on feedback from this course and others Humphrey revised his original manuscript and the final version was published in 1994.

Under the PSP data was manually collected from developers. This involved developers filling out a number of lengthy forms. This data was compiled in spreadsheets and then manually analysed. Some of the forms required were: A project plan summary, a time-recording log, a defect-recording log and a code checklist. This involved a significant amount of time and effort. Despite this seemingly inefficient process Humphrey’s insisted that the process would never be fully automated. “It would be nice to have a tool to automatically gather the PSP data. Because judgement is involved in most personal process data, no such tool exists or is likely in the near future.” PSP provides engineers with a general framework they must comply with when developing software. The PSP consists of methods, forms and scripts engineers can follow to ensure they consistently use sound engineering practices. The manual nature of the PSP meant the analytics were fragile and created significant data quality problems.

**LEAP Toolkit**

The Leap toolkit was developed to solve the problems of data quality associated with the PSP by automating and normalizing data analysis. The developer still manually enters data but the toolkit automates the data analysis and provides analyses not available from manual analysis. Leap data enabled developers to maintain control of their own data through a repository specific to that developer. This means the leap data is portable as the developer can keep using this repository regardless of what project they are currently working on. There are also negatives associated with the leap toolkit. The automation of the analysis made some analytics extremely difficult to collect. The research team eventually came to agree with Humphreys that the PSP approach could never be fully automated.

**HACKYSTAT**

Hackystat is the third generation of PSP data collection tools developed at the University of Hawaii. The Hackystat project was completely different to the PSP or the Leap toolkit. The goal of the hackystat was to find a way to collect software process and product data with little to no manual input from developers – it was to be as unobtrusive as possible. Hackystat uses sensors attached to development tolls to gather process and product data and then sends this data to a server, which other services can query to conduct higher level analyses. There were four key design features: Both client and server-side data collection, unobtrusive data collection, fine-grained data collection and finally both personal and group-based development. Hackystat analysed much of the same data as the original PSP but it removed the manual nature of the data collection and analysis. Hackystat collects all the data and then sends this data to a server which performs high level analyses.

Hackystat has led to a variety of technical innovations, including

* The development of a toolkit for defining and visualizing software project telemetry.
* Support for high-performance-computing software development.
* A method for prioritizing which software development artifacts to inspect.
* An operational definition for test-driven development.
* An approach to software process discovery.
* The Software ICU (intensive care unit), which assesses a project’s health both alone and in relation to other project. See image below.



Software measurement is an emerging field of software engineering. The errors associated with manually collected data meant there was a gap in the market for computational platforms which would automatically collect and analyse software engineer’s data. Today this gap has been filled and there many computational platforms available which provide automated code review and allow companies to analyse data in a simple straightforward manner. Some of the main computational platforms available today are: Code Climate, Codebeat, Codacy and Scrutinizer.

**Code Climate:**



Code Climate is a computational platform that is used by over 100,000 projects. They analyse over 2 billion lines of code each day. Code Climate helps team create better code, faster by incorporating fully-configurable static analysis and test coverage data into the development workflow.

Some of the services provided by Code Climate are as follows:

* Automated Git Updates- Nothing to install. Code Climate runs every time a new commit is pushed.
* Activity Feeds- Up-to-the-minute information so a company can see when and how code changes.
* Instant Notifications- Major security and quality changes pushed to where employees work: email, Campfire, HipChat, and RSS feeds.
* Team Sharing- Instant access for a whole team to maximize code visibility across projects.
* Duplication Detection- Fuzzy matching algorithm finds DRY-violations that human reviewers might miss.
* Email Notification- Instant email notifications to let a company can know when new security and code issues arise
* Security Dashboard

**Codebeat:**



Codebeat is a dynamically growing tool that covers major technologies and supports many programming languages. Codebeat try to combine hard computer science with modern day data science to improve the users code and help them grow as a developer. Codebeat start by using static analysis to understand the structure of the code. Then from looking at the code structure they calculate standard code quality metrics. They use their own algorithms to maximize correctness, extensibility and performance. One the algorithms have finished and generated a report for the project codebeat process current and historical data to ensure users get the most out of their codebeat experience.

* They find Quick Wins by figuring out what are the most significant problems in the users codebase.
* They generate annotated differentials for edits that cause significant code quality changes.
* They try to relate the current state to historical data to help the user find that extra bit of inspiration to they need to solve quality problems.

**Codacy:**



Codacy is another alternative. Their goal is to help engineers write better code, faster. “We wanted to let engineers get on with the parts of their work that will change the world, while we make the process of creating high quality software easy.”

# Algorithmic Approaches

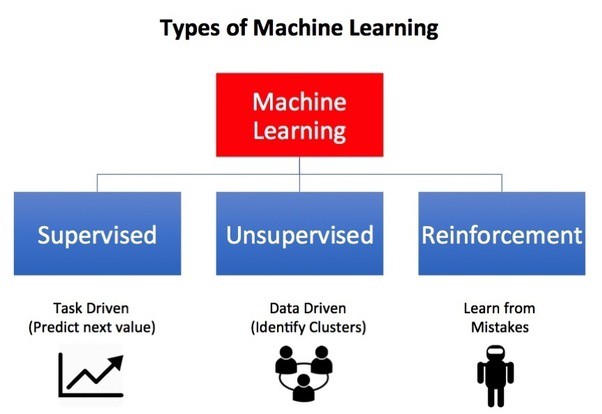
Artificial Intelligence (AI) is a rapidly advancing technology, made possible by the Internet, that may soon have significant impacts on our everyday lives. AI traditionally refers to an artificial creation of human-like intelligence that can learn, reason, plan, perceive, or process natural language.

**Machine Learning**

Machine Learning is one particular approach to Artificial Intelligence. The basic idea is that instead of programming every step the computer takes, machine learning makes use of learning algorithms so computer systems can make inferences from data to learn new tasks. These algorithms make data driven predictions or decisions. The result is that learning algorithms and not computer programmers dictate the course taken by any particular program. This means that computers can now be use for more complex problems than they could when everything was manually programmed. The field of machine learning has definitely gained popularity over the past couple of years. Today, machine learning algorithms are used in many aspects of everyday life. For example, Gmail successfully filter 99.9% of spam emails through the use of machine learning algorithms.

The basic process of machine learning is to give training data to a learning algorithm.

The majority of practical machine learning uses supervised learning. Based on inferences the learning algorithm makes from the data it generates a new algorithm. The same learning algorithm can be to generate many different models if the training data is altered. For instance, one learning algorithm could be used to teach the computer how to translate languages. There are three main categories of machine learning algorithms: Supervised learning, unsupervised learning and reinforcement learning.



**Supervised Learning:**

Supervised learning is where you have input variables (x) and an output variable (Y) and you use an algorithm to learn the mapping function from the input to the output.

*Y = f(X)*

The goal is to approximate the mapping function so well that when you have new input data (x) that you can predict the output variables (Y) for that data. Essentially we want to find specific relationships or structure in the input data that enable us to effectively produce

correct output data.  Common algorithms in supervised learning include logistic regression, naive bayes, support vector machines, artificial neural networks, and random forests.

There are two main points of concern to consider when conducting supervised learning. These are model complexity and the bias-variance trade off.

Model complexity refers to how complex the model you are attempting to learn is. If you have a small amount of input data or if the input data is not uniformly spread throughout different possible scenarios a simple or low-complexity model should be used. If a complex model is used on a small number of data points it will overfit. In simple terms this means the model will learn to produce the input data without actually learning the internal structures of the input data.

The bias-variance tradeoff refers to the balance between bias, the error of any given model, and variance, the amount by which bias can vary for different training sets. Typically, increased bias leads to decreased variance. Where a model falls on the bias-variance scale should be decided based on the specific problem and the nature of the data. It also depends on what the criteria for the model are. If a guaranteed baseline level of performance is critical then a higher level of bias should be implemented.

**Unsupervised Learning:**

Unsupervised learning is where you only have input data and no corresponding output variables. In other words, the data given to the learning algorithm is unlabelled. The most common tasks within unsupervised learning are clustering, representation learning, and density estimation. In each of these situations the challenge is to discover implicit relationships without using explicitly provided labels. Some common unsupervised algorithms include k-means clustering, principal components analysis and autoencoders.

Unsupervised learning is very useful in exploratory analysis because it can automatically identify structure in data. Brett Houlding an Assistant Professor within the discipline of Statistics, School of Computer Science and Statistics, Trinity College Dublin, discusses in his lecture notes for Multivariate Linear Analysis how unsupervised learning in the form of k-means clustering can be used to divide data into groups of similar observations, whilst observations between groups are different. An example of this would be when an online retailer uses an unsupervised learning algorithm to discover similar items that are often bought together. The retailer can then use this algorithm as a recommendation system for customers.

**Reinforcement Learning:**

Reinforcement learning falls between supervised and unsupervised learning algorithms. In this type of Machine Learning an agent learns how to behave in an environment byperforming actions and seeing the results. Reinforcement learning is based on feedback available for each predictive step or action. There are three basic concepts in reinforcement learning:

1. State: This describes the current situation
2. Action: This refers to what an agent can do in each state.
3. Reward: This is what the agent receives each time it performs an action in a state.

In its simplest terms, reinforcement is learning best actions based on reward or punishment.

Given its and the environments states the agent will choose the action which will maximise its reward. This action will change the environments and the state of the agent. The action will also be interpreted to give a reward to the agent. By repeating these steps many times the agent will learn the best behaviours.

# Ethics

In this report I have discussed in detail the collection of measurable data used to assess a software engineer’s performance along with how and where this data can be analysed. The final point of discussion is: How ethical is all of this?

Ethics refers to the moral principles that govern a person’s behaviour or the conducting of an activity. What are the moral principles that govern the collection and analysis of software engineering data?

Software is present in almost all aspects of everyday life today. By participating in a software development process, software engineers can influence the final product, namely the software itself, in different ways including those that may be contrary to public interest. In other words, they could engage in an unethical behavior, inadvertently or deliberately. To ensure, as much as possible, that their efforts will be used for good, software engineers must commit themselves to making software engineering a beneficial and respected profession.

Before discussing the ethics of data collection regarding software engineers specifically I want to discuss the ethics of data collection in general. Fundamentally, almost every aspect of our lives revolves around data.  From social media companies, to banks, retailers, and governments - almost every service we use involves the collection and analysis of our personal data and sometimes the ethics of these institutions is called into question. One prime example of this is the Cambridge Analytica scandal. Facebook exposed data on up to 87 million Facebook users to a researcher who worked at Cambridge Analytica the political consulting firm that did work for the Trump campaign. Through an app he developed this researcher was able to gather information about Facebook users and their friends without their knowledge. This scandal brought the topic of data protection to the attention of the general population and now more than ever people are questioning how much they can trust Facebook, and other online platforms, with their personal data.

One thing the Cambridge Analytica scandal highlighted was the conflict companies face between protecting user’s data and using this data to make money. Following the scandal Sandy Parakilas, who worked on the privacy side at Facebook, told the New York Times**:**

*“The people whose job is to protect the user always are fighting an uphill battle against the people whose job is to make money for the company”.*

This highlights the conflict between protection users’ personal data and generating revenue. Businesses and institutions must decide which of these aspects is more important to them and act accordingly. Unfortunately, businesses often choose wrongly and users personal data gets abused. General Data Protection Regulation (GDPR) is a new set of rules designed to give EU citizens more control over their personal data. Unfortunately, data breaches are inevitable. Information gets lost or stolen or released into the wrong hands as was the case with Cambridge Analytica. Under the terms of GDPR, not only will organizations have to ensure that personal data is gathered legally and under strict conditions, but those who collect and manage it will be obliged to protect it from misuse and exploitation, as well as to respect the rights of data owners - or face penalties for not doing so. GDPR represents a step in the right direction in terms of prioritising the protection of personal data.

There are similarities to be drawn between the protection of personal data and the protection of software engineer’s data. Computational platforms like the ones I discussed above take copious amounts of data from the software engineers without them even realising. Some software engineers see this as invasive and an abuse of power. Managers can use this data for whatever they like and the engineers have no control over the data.

In the paper “Searching under the Streetlight for Useful Software Analytics” which I have already discussed in this report, Philip M. Johnson of the University of Hawaii claims that the easier an analytic is to collect and the less controversial it is to use, the more limited its usefulness and generality. For example, collecting the data in a configuration management repository is easy, and the repository’s public nature means that developers generally don’t object to analysis of this data. However, the resulting analytics are constrained by the very narrow slice of development activity captured. Conversely, the hackystat technology can yield insightful, high-impact analytics but there are also certain social or political problems associated with it. The team of researchers at University of Hawaii discovered a number of these problems. Firstly, some developers viewed the hackystat technology as a bug. They would not install software which collected data regarding their work without telling them. Secondly, the fine-grained data collection was found to sometimes cause discord within a development group. The transparency of the hackystat data means that developers know exactly what everyone in the group is and is not doing. This can cause conflict. Finally, developers are not comfortable with the level of fine-grained data about their work which was being provided to management. Essentially, some developers view the hackystat technology as invasive.

Certain metrics are important and should be measured as they can provide managers with useful information which they can then use to get the most out of their employees. However, a clear line must be drawn between data that is work related and personal data. More and more often companies are crossing the line and analysing private personal information about their employees.

An article in the New York Times highlighted the degree to which companies today use their customer’s personal data to increase their sales. “Almost every major retailer, from grocery chains to investment banks to the U.S. Postal Service, has a ‘predictive analytics’ department devoted to understanding not just consumers’ shopping habits but also their personal habits, so as to more efficiently market to them.” It is unsettling to think about how much companies can decipher about us just based on what we buy, when we buy it and where we buy it. These practices are becoming more and more commonplace as the data analytics industry continues to grow.

# Conclusion:

The field of software engineering is complicated. Measuring the software engineering process is in turn very complicated. What is measurable? What should we measure? What should not be measured? Where should we analyse this data? What does this data tell us? Is all of this ethical? There is no hard and fast rule for how we should measure the software engineering process. The uncertainty surrounding the area is something I think will exist for years to come and is the reason why we must practice caution when measuring the performance of a software engineer.

The aim of this report was to analyse how the software engineering process is measured. I have done so under the headings of measurable data, computational platforms, algorithm approaches and ethical concerns. I provided a well-rounded view on each of these aspects. As I mentioned there is much uncertainty regarding measuring the software engineering process and it will continue to evolve as the field of software engineering itself continues to evolve.

When measuring the performance of a software engineer it is important to remember that this developer is a person not a machine. The nature of the work done by software engineers means it is sometimes easy to forget that there is a real person sitting behind the screen typing out the code. This person is influenced by all kinds of factors. Coding metrics are only half of the story. How comfortable is the engineer is in the workplace? How happy are they? How well do they get on with their co-workers? These factors although more challenging to measure and quantify are fundamental when assessing a software engineer’s performance. Managers must be responsible when deciding what to measure and try their best to perform well-rounded analyses.

Computational platforms are the vehicle between collecting the raw data and useful, insightful analyses. The automation available on these platforms today mean that there is no limit to the insight which we can gain from data collection. Machine learning and AI mean that machines today are capable of much more than they were in the past. As AI advances there is no limit to what computers will be able to derive from data.

The ethics surrounding the areas of data collection and data analysis are definitely concerning. I think the lines between business and personal data have been blurred but with laws such as GDPR society are taking steps to set clear boundaries and protect our personal data. As technology advances governments and law enforcement must take the necessary steps to keep up and protect people against the new threats that will come with new technology.

The field of software engineering is one that has undergone rapid expansion in the past decade. ‘Artificial Intelligence’ and ‘Machine Learning’ are terms that meant nothing to most people even five years ago. Today they influence most of us in our daily lives. However, it appears that few genuinely understand the true meaning of them and the potential they have to alter our world irreversibly. And, while the principal component of such change is for the good, there is inevitably an element of harm. It is alarming, the extent to which humans can be tracked and analyzed in the modern world today. At the end of the day, the responsible implementation of this power is up to us and it is our responsibility to discuss and examine all of the possible implications of these.

**Bibliography:**

1. <https://web.cs.dal.ca/~hawkey/3130/SEBackground4.pdf>
2. <https://medium.com/omarelgabrys-blog/software-engineering-software-process-and-software-process-models-part-2-4a9d06213fdc>
3. <https://www.uio.no/studier/emner/matnat/ifi/INF5181/h11/undervisningsmateriale/reading-materials/Lecture-06/morasca-handbook.pdf>
4. <https://blog.gitprime.com/5-developer-metrics-every-software-manager-should-care-about/>
5. <https://www.researchgate.net/profile/Jan_Sauermann2/publication/285356496_Network_Effects_on_Worker_Productivity/links/565d91c508ae4988a7bc7397.pdf>
6. <http://www.hitachi.com/rev/pdf/2015/r2015_08_116.pdf>
7. <https://www.thebalancecareers.com/what-are-soft-skills-2060852>
8. <http://www.citeulike.org/group/3370/article/12458067>
9. <https://www.codacy.com/about>
10. <https://codeclimate.com/>
11. <https://codebeat.co/>
12. <https://www.kdnuggets.com/2016/08/10-algorithms-machine-learning-engineers.html>
13. <https://www.techemergence.com/everyday-examples-of-ai/>
14. <https://machinelearningmastery.com/supervised-and-unsupervised-machine-learning-algorithms/>
15. <https://www.internetsociety.org/resources/doc/2017/artificial-intelligence-and-machine-learning-policy-paper/?gclid=EAIaIQobChMI59Tpj6vP3gIVBNKyCh2fZw48EAAYAyAAEgKwIvD_BwE>
16. <https://www.scss.tcd.ie/Brett.Houlding/ST3011.html>
17. <https://towardsdatascience.com/supervised-vs-unsupervised-learning-14f68e32ea8d>
18. <http://enhancedatascience.com/2017/07/19/machine-learning-explained-supervised-learning-unsupervised-learning-and-reinforcement-learning/>
19. <http://www.kaner.com/pdfs/metrics2004.pdf>
20. <https://www.nytimes.com/2017/11/19/opinion/facebook-regulation-incentive.html>
21. <https://www.vox.com/policy-and-politics/2018/3/23/17151916/facebook-cambridge-analytica-trump-diagram>
22. <https://www.computer.org/web/education/code-of-ethics>
23. <https://www.nytimes.com/2012/02/19/magazine/shopping-habits.html?pagewanted=1&_r=1&hp>