MEASURING THE PERFORMANCE OF A SOFTWARE ENGINEER

INTRODUCTION:

Measuring the work of a software engineer has long been debated. Some adamantly oppose the idea - claiming that tracking metrics works best in a factory, but software engineers don't work on an assembly line. They argue that this approach is burdensome, tends to actually reduce productivity and has a negative impact on motivation.

I respectfully disagree. If the appropriate data is firmly established, measured in an efficient manner and interpreted correctly, I believe that meaningful insights can be provided. Measurement can be used to assess situations, track progress, evaluate effectiveness, and aid management decisions. Those against the idea of measuring a software engineer's work tend to consider an individual programmer in isolation. Nowadays, however, it’s important to look at the bigger picture. Incredible coordination is required throughout the whole company. If the entire team is on the same page, all understanding the metrics for success, the company as a whole is better prepared to excel.

Compared to traditional industries, software engineering is a relatively new phenomenon, originating in the 1960s. In the early years, despite plenty of academic research and publications relating to the measurement of software engineers, the actual implementation of this by companies was patchy at best. It proved a lot more difficult than simply counting how many products a factory worker makes, or the volume of sales from brought in by a salesperson. The lack of a coordinated, comprehensive framework for understanding and using the data collected was a glaring issue. The recent rise of new technologies, methodologies and companies solely focused on data analysis has bridged the gap somewhat. An increasing number of companies are measuring the work of their software developers, but are they doing this correctly?

This report explores the ways in which the software engineering process can be assessed in terms of **measurable data**. It will also provide an overview of the **computational platforms** available to perform this work, as well as some of the **algorithmic approaches** available. Finally, I'll describe some of the ethics concerns surrounding this kind of analytics.

Before we delve into specific topics, let's first consider a **case study** of how a well-known company measures the performance of their software engineers: Google. Their practices can be kept in mind for the subsequent topics discussed in the report.

Google:

Google employees undergo an annual performance review, along with a mid-year checkpoint. Each employee has pre-defined OKR's (Objectives and Key Results) that they're expected to deliver on. For such a high-tech company, the manual nature of some of these processes is surprising. Surveys are filled in by the employee and their peers, including the 'Googleist Engagement Survey' and the 'Annual Upward Feedback Form'. A former employee, Edmond Lau, claims "it pains me to think of all the lost productivity that must take place during their twice-a-year performance reviews and promotion processes". But is there any way to avoid this?

Perhaps not. Google are widely considered to be ahead of the curve when it comes to performance management.

The primary focus of Google's performance management procedures are the "non-code" aspects, such as 'Googleyness' (the employee's adherence to Google's values) and 'Thought Leadership Ability' (how much an employee is seen as a reference for a given niche of expertise). Other considerations that are more specific to projects they've completed include problem solving, the ability to handle complex tasks and the execution of these tasks. An employee's impact on Google's bottom-line performance is also taken into account.

Interestingly, Google have a team of quants - the 'People & Innovation Lab' - whose sole aim is to study people data (performance, engagement, happiness etc.). Here, we see how data can be used to measure their software engineers' performance and how their findings can be used to aid management decisions. According to the lab manager, "when your employees build virtual tours of the Amazon and tools to translate between 60+ languages, you need creative ways to think about productivity, performance and employee development". Although they don't explicitly reveal the "code" metrics they track, one can assume that they use a combination of these and the "non-code" aspects mentioned previously to evaluate the performance of their software engineers.

MEASURABLE DATA:

"If we have data, let's look at data. If all we have are opinions, let's go with mine."

– Jim Barksdale

Is it enough to say 'you know a good software engineer when you see one'? It's easy to list their qualities - they're productive, make good architectural decisions, care about coverage tests and consistently produce quality code. The problem is that this is a purely subjective approach. No doubt, it's a combination of elements that determines a good software engineer, but are these measurable?

Data is an important part of quality improvement in any industry. A huge amount of data can be collected throughout the software engineering process. However, the challenge of gathering software engineering data is ensuring that it can provide useful information for process improvement. Also, the collection process shouldn't be burdensome for the software developers as this would damage their existing productivity. Although some software engineers may continue to claim that key attributes like quality, usability and maintainability are simply not quantifiable, we can try use measurement to advance our understanding of them. Perhaps we can identify links between these attributes and quantifiable data like lines of code, severity of bugs or code coverage.

Before deciding what data to gather, it's important to understand the Software Development Life Cycle. The aim of such a process is to divide projects into distinct, manageable phases. Generally, software engineers will adhere to a standard product development framework.

1. **Planning**: A vital part of software development. Requirements are gathered and the scope of the project is determined and documented. A plan is drafted.
2. **Development & Implementation**: The actual task of writing the code to develop the software. There may be a pilot stage to see if it's functioning properly.
3. **Testing**: Assess the software for defects or bugs.
4. **Deployment**: After it's approved for release, the software is deployed
5. **Maintenance**: Software improvements and new requirements can often take longer than creating the initial software. It is maintained and upgraded from time to time to adapt to changes in demands.

There are several specific models followed by organisations, such as the Waterfall Model and Iterative Process. Every step in the project should be documented for future reference.

So why should we gather data on software engineers? They're the most important asset of a software company, so it's essential to understand how they're working. If any noticeable flaws in their development process can be identified, management can maximise efficiency by encouraging the adoption of the most efficient processes. The primary goal of measuring a software engineer should be to drive improvements in performance, but the company should establish specific questions to be answered by the data gathered. This avoids the over-collection of data, which is a problem that usually arises when there's a lack of purpose. It’s important for the data to be focused, accurate and useful, rather than plentiful.

A review of the software engineer's coding activities can yield useful information. The following are examples of what can be gathered:

- **Readability** of their code
- **Source Lines of Code** is simply how many lines are in the software's source code, but can the number of lines written really be an accurate measure of skill? It's a poor measure if considered in isolation, because it can vary depending on the engineer's approach.
- **Number of Commits** refers to the amount of times a software engineer contributes to their code. A high number here doesn't necessarily distinguish a good software engineer from a poor one, but it gives us an insight into their activity.
- **Lead Time** is a great metric to track. It's defined as "the time elapsed between the identification of a requirement and its fulfilment. Put simply, it measures the total time it takes for work to move through the development process, from the moment it is requested to the time it's delivered. Lead time can help management predict how much work a software engineer is capable of completing in a particular time frame.
- **Cycle Time** is similar to lead time. The difference is that for cycle time the clock starts when the software engineer actually begins working on the project, rather than when the initial request is made.
- **Technical Debt** is a concept in software development that reflects the extra development work that arises when code that is easy to implement in the short run is used instead of applying the best overall solution. A good software engineer pays back technical debt promptly with a rewrite.
- **Code Churning** is the percentage of a developer's own code representing an edit to their recent work. It's typically measured as lines of code that were modified, added and deleted over a short period of time. A sudden increase in the churn rate can indicate

that something is off with the development process. Perhaps the software engineer is experiencing difficulty in solving a particular problem, or maybe they're just putting the final touches to their software before a release.

- **Test Coverage** is a measure used to determine whether the programmer's test cases are actually covering the source code and how much is executed when these test cases are run. Management can decide what coverage they deem adequate and see if their software engineers are meeting these standards.
- **Bug Fixing.** How much time is the engineer spending on bug fixes? Management should be interested not necessarily in how many bugs are arising in the developer's code, but rather the severity of these bugs and how they deal with these.
- It's also useful to track the **complexity of the task** and whether it was carried out by an individual or a team.

There's a major problem in using the above metrics in isolation. If a company only cared about code related data, there'd be an obvious flaw. Someone who commits regularly to their repositories and has excellent code coverage, but isn't responsive to emails, doesn't contribute much to meetings and isn't receptive to constructive criticism will still score highly. Therefore, it's absolutely essential to take non-code aspects into account too. Admittedly, it's more difficult to measure these, but they simply cannot be ignored. Cultural compatibility is something that should be tackled before the hire is made, so management should be confident that the software engineer fits in with the values of the firm. Other metrics that should be continually assessed include their ability to collaborate in a team environment, their business impact and their own awareness of how their work affects the whole company's performance. Judging these can be somewhat subjective, so to aid in their judgement a company could look at factors like the software engineer's communications, the meetings they've attended and their pull requests. A potential flaw to consider is measurement dysfunction. This essentially means that if an organisation's goals are to improve software development by tracking metrics, this may inadvertently put pressure on their software engineers to produce 'good' metrics while actually reducing their performance.

COMPUTING THE DATA:

Now that we've identified some useful metrics to measure, let's consider some data collection methodologies. In many organisations, a data collection system is often an integral part of the software configuration or project management system, without which the chance of success of large, complex projects will be reduced. If a company is trying to understand the interplay of important variables on their software engineer's performance, there should be competent platforms available to compute such data.

Earlier research into these methods deal with more manual processes. Taking into account the rapid advancements that have occurred in software engineering since these papers were published, some of the methodologies described may seem inefficient compared to more modern, big data driven approaches. Nonetheless, they identify some vital aspects of the data collection process.

Basili and Weiss:

In 1984, Basili and Weiss outlined "a methodology for collecting valid software engineering data". After establishing the goals of the data collection and developing a list of questions of interest, they recommend a schema to follow.

1. Establish a complete and consistent data categorisation scheme.
2. Design and test the data collection form. A conflict can arise between the desire to collect detailed data and the need to minimise the time and effort involved in supplying the data. They advise fitting the form on one piece of data and giving the programmer some flexibility in their responses.
3. Collect and validate the data. Validation consists of checking the forms for correctness, consistency and completeness.

Personal Software Process:

Developed by Watts Humphrey, the **Personal Software Process (PSP)** is a technique that's intended to help software engineers better understand and improve their performance by tracking their predicted and actual development of code. Several modifications of the process exist, but most require a lot of manual input from the software engineer. One version requires them to fill out 12 forms, including a project plan summary, a time-recording log, a defect-recording log, a process improvement proposal, size and time estimations and a design checklist. These forms usually contain over 500 distinct values which developers must manually calculate. Collecting and managing this data requires substantial effort. Interestingly, Humphrey actually embraced the manual nature of the PSP: "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 to in the near future".

Extensive research has been conducted into this process, especially by the Collaborative Software Development Laboratory at the University of Hawaii. They found that its manual nature created significant potential for human error, and this could result in data quality problems. Another problem with manual collection is that any assessment feedback is temporarily detached from the original context by months or years. As PSP cannot be fully automated, many believe it doesn't provide enough return on investment. To address this problem, the University of Hawaii developed the Leap Toolkit.

Leap:

The Leap Toolkit tries to address the data quality issues of PSP by automating and normalising data analysis. LEAP is actually an acronym of its design principles - light-weight, empirical, anti-measurement dysfunction and portable. The software engineer is still required to manually input data, but unlike PSP, the subsequent analysis is automated. It creates a repository of personal process data that developers can keep with them as they move from project to project, and even between organisations. Though certain analytics became easier to obtain, the flexibility of PSP was lost and others became increasingly difficult to collect, as a whole new toolkit would need to be designed and implemented.

Hackystat:

The next project that came to fruition was Hackystat. It's an open-source framework that implements a 'service oriented' architecture. Sensors attached to development tools gather process and product data, send it to a server and then other services can analyse. Hackystat has four important design features. The first is both client- and server-side data collection. The second is unobtrusive data collection - users shouldn't notice that data is being collected. Next is fine-grained data collection - data can be collected in real time, even tracking a developer as they edit code. The final feature is both personal and group-based development - it can track the interplay among developers. Collecting data on a second-by-second basis is one of the benefits of Hackystat, but also a significant obstacle to industry adoption of the software. Many software engineers are uncomfortable with the idea of providing management with access to such data. One user even described it as "hacky-stalk".

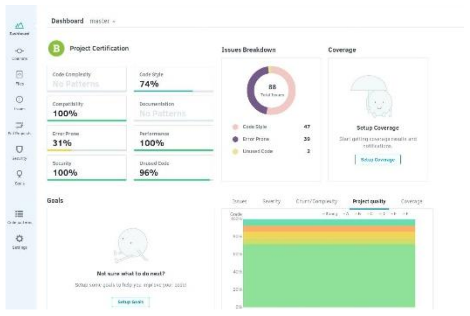
Outsourcing data analysis is becoming increasingly popular, and the analysis provided is hugely comprehensive. "Data Analytics as a Service" has become a popular trend in technology. Companies have been established to focus solely on providing data analytics, and even more specifically ones that analyse software engineers, such as Code Climate, Code Beat or Codacy. They offer automated code review tools, which assess the programmer's work on a number of different levels, then make recommendations about what can be improved. Effectively, the field of measuring software engineer has sparked the creation of a whole new market, and opportunists are capitalising on this. Let's consider an examples of such companies:

Code Climate:

Code Climate's mission statement claims that they incorporate 'fully-configurable test coverage and maintainability data throughout the development workflow, making quality improvement explicit, continuous and ubiquitous'. The company's services have been utilised by organisations such as Salesforce to measure the performance of their software engineers. Now, over two billion lines of code are analysed each day. Code climate delivers comprehensive analytical tools that can flag potential code errors, security vulnerabilities and instances of flawed coding methodology, all packaged into a manageable platform. The platform enables automated analysis of complex codebases written in Javascript, Ruby, Python, PHP and CSS, among others. Code Climate's software can be integrated seamlessly with Github and provide test coverage information, duplication detection and activity reports. Automating this code review process allows the engineer to focus on what's important - developing software - without having to worry about any manual self-assessment.

Codacy:

Codacy is known for having a very clean user interface, where it's easy to find the important information and configure the main parameters. The code quality measures are grouped into eight categories - code complexity, compatibility, error-prone, security, code style, documentation, performance and unused code. Codacy claim to 'save developers thousands of hours of time in code review and code quality monitoring'. Adobe and PayPal are notable clients.

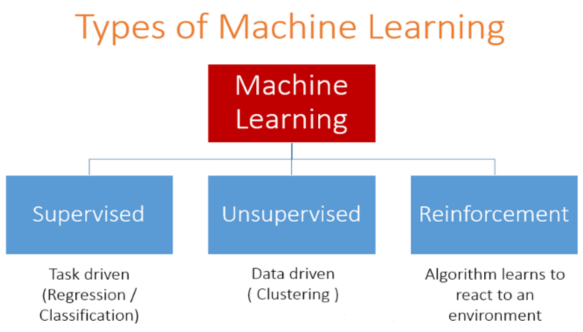


Code Complexity	No Patterns
Code Style	74%
Compatibility	100%
Documentation	No Patterns
Error Prone	31%
Performance	100%
Security	100%
Unused Code	96%

Code Style	47
Error Prone	39
Unused Code	2

ALGORITHMIC APPROACHES:

The computational platforms mentioned above have a number of algorithms running in the background. Once the data is gathered, an algorithm analyses it and produces useful results for the end-user. Manually identifying defects in a program can be difficult, especially in large scale projects. Analysing the data continuously through the use of algorithms is a more effective approach. It can provide us with a deeper understanding of the raw data and recognise any underlying structures or common themes. Predictive analysis looks at statistical methods which analyse data in order to make predictions about patterns which may occur in the future. Now, I'll discuss some of the algorithms that may be used for data analysis. Some companies like Google may have proprietary algorithms to analyse the performance of their staff, as mentioned earlier. There are a huge number of techniques available, but considering the prominence of machine learning, I'll focus mainly on the algorithms that drive this technique. In general, machine learning algorithms can be divided into **supervised**, **unsupervised** and **reinforcement** learning.


Types of Machine Learning
Machine Learning
Supervised
Unsupervised
Reinforcement
Task driven (Regression / Classification)
Data driven (Clustering)
Algorithm learns to react to an environment

Supervised Learning:

Supervised Learning is where the algorithm generates a function that maps inputs to desired outputs. It's 'supervised' because the data scientist acts as a guide to teach the algorithm what conclusions it should be drawing from the data. To teach it, they use what's called a 'training dataset', where there's a number of inputs and their corresponding outputs. The possible outcomes should be identified in advance. Common applications of supervised learning algorithms include:

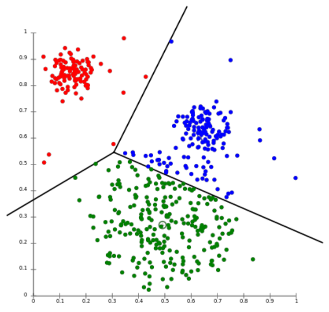
- **Classification**: The goal is to get the computer to learn a classification system. Once it processes the inputs, it should classify the observation into a specific group. If a company wants to group programmers based on their given inputs, this algorithm could be useful.
- **Linear Regression**: This is a commonly known statistical method, but can be applied to machine learning. A regression line is fit to the data, and from this a formula can be derived so that for each input X we can estimate the output Y.

Unsupervised Learning:

Many people believe this is more synonymous with actual machine learning, because the goal is to have the computer learn how to do something we don't tell it how to do. The idea is that the algorithm can learn to identify complex processes and patterns without guidance from a human. Examples of widely used unsupervised learning algorithms include:

- **Principal Components Analysis** is a method used to compress a dataset with a lot of dimensions into something that captures the essence of the original data. The goal is to identify what variables explain most of the variance in the data. This method could help identify relationships between variables. For example, if the software engineer has a high value for lead time, PCA may tell us that we'd expect them to have a high level of code churning too.

- **k-Means Clustering** can reveal is there is a group structure in the dataset. If the algorithm establishes that there is, such as productive engineers and unproductive engineers, we can examine the characteristics of such clusters. This is an iterative method that divides the data into k distinct groups so that observations within a cluster are similar, whilst observations between groups are different. The default method is the Hartigan-Wong algorithm, which defines the total within cluster variation as the sum of squared Euclidean distances between items and their corresponding centroid.



Reinforcement Learning:

This type of learning trains the algorithm to make a sequence of decisions. The algorithm faces a game-like situation. Through trial-and-error, it attempts the task with the goal of maximising long-term reward. Either a penalty or a reward is taken on for each action the system performs. The models behind controlling autonomous cars are a potential application of reinforcement learning - it should make the best decision without guidance from a human. Throughout the training process, the car should learn from its errors and not make the same mistake again.

The capabilities of these algorithms are increasing rapidly. The goal of the learning algorithms is to minimise the error with respect to the given inputs. So as time goes on and more inputs are fed into the system, the accuracy of the decisions made by these algorithms will improve. However, they lack a human element, so I'd recommend using them in conjunction with human judgement when analysing the performance of a software engineer. Perhaps these will continue to develop so that one day they won't require any human intervention, but I don't think we're at that stage yet.

Along with assessing the performance of engineers currently working at a company, these algorithms could also potentially be used in the hiring process. They can mine GitHub repositories to identify subject matter experts.

ETHICS:

Now we must address an important question: Is all of this ethical? At the moment, I think it's fair to say that this is a grey area. In traditional disciplines like medicine, law and psychology, the ethical problems have been long recognised and a regulatory framework has been established. In the field of software engineering, however, the lines are somewhat blurred between what practices are ethical and what is an intrusion of privacy. The boundaries on what information is acceptable to collect are constantly being tested. How much is too much? In my opinion, the debate about ethics will continue to intensify as trends like artificial intelligence, big data, machine learning and data analytics become an integral part of our society. Changing laws like the introduction of GDPR are adding a sense of urgency to the debate. Technological capabilities are advancing at a rapid pace, but regulation is struggling to adapt at the same rate. Businesses themselves end up struggling to understand the inexplicit requirements, laws and regulations that they're supposed to adhere to.

There's an obvious trade-off between 'easily obtained analytics and richer analytics with privacy and overhead concerns'. When measuring the work of their software engineers, management need to strike a balance between gathering enough useful information while respecting the privacy of the employee, but this is easier said than done. The company must be utilising the information they gather appropriately. We've outlined earlier how tracking certain metrics can help boost productivity, so it's important that this goal is always kept in mind, as straying from it can raise ethical concerns. It's becoming increasingly common for companies to analyse very personal data, like healthcare records, financial information and search history. Furthermore, I don't think that measuring analytics such as age, nationality or race should be permitted, as it could give rise to prejudice depending on who's interpreting the data. The software engineer's right to keep some of their information private should be respected. Management should focus of the important data without delving into details that are not useful for management purposes and compromise the software engineer's privacy. Workers should feel as ease, not like they're in some sort of terrifying Big Brother scenario, where they're constantly being scrutinised by algorithms. For example, Workday, a HR technology company, have developed a tool that tells your employer if you're likely to leave your job soon - even before you may know it yourself. It can then give companies recommendations like "give Liam a promotion" to allow them to retain their most talented employees.

Data analytics are a wonderful tool to improve productivity, but controversy often ensues if they're used as a way of identifying the weakest link. Can an algorithm really make this decision. Can creativity be taken into account? Not really. Could those who don't fit the cookie-cutter models fall by the wayside? Probably.

Data sovereignty and data ownership are grey areas that may raise ethical issues. Data sovereignty is the concept that information which has been converted and stored in binary digital form is subject to the laws of the country in which it is located. It's important to have clarity around where the data is located and what laws its subject to. Cloud computing services raise questions about data sovereignty. We also have to ask the question: who actually owns the data? If you post on Instagram, do you give them ownership of your photo? Data collected in relation to the performance of software engineers can be considered to be owned by the company, but even then, what person? The COO? The CEO?

When it comes to ethics, the sale of data is another cause for concern. Data is extremely valuable to companies, as it can provide them with interesting insights that lead to increased profits. A recent report by Oracle argues that a major factor in the success of companies like Google, Uber and Amazon is how they've embraced the mindset of "data as an asset". Data brokers, lurking in the shadows, collect information about consumers and sell this onto companies. There's a strong possibility that you may have never heard of these companies because they don't have a direct relationship with the people they're collecting data on, but you'll be shocked to find how much they know about you. The Cambridge-Analytica scandal also exposed a political consulting firm's ability to circumvent the rules and gather data about 87 million Facebook users, which left many consumers feeling uneasy.

Transparency in what's being collected, how it's being used and if it's being passed on to any external parties is vital. The metrics that management are analysing and their purpose for doing so should always be communicated to the software engineers. This can help break down the negative stigma that exists among employees regarding data collection, demonstrate its usefulness and alleviate concerns about any improper use of personal information. It may be a good idea to give the employees some level of control over what they wish to reveal. Perhaps a better approach could even be to provide management with aggregated data about a project, rather than individual details about a developer. Without transparency, the relationship between management and the software engineers can be frayed. Respecting their employee's rights to privacy and creating a good working environment can benefit the company too. Nowadays, there are plenty of platforms available for employees to voice their discontent, such as Glassdoor. A poor reputation could damage the company's attractiveness to top talent. The opposite is true if they manage to establish a transparent work environment where the software engineers are comfortable with the data being collected about them.

It's essential for clearly defined boundaries, laws and legislation to be established. These shouldn't just address the gathering of information, but also data use and retention. Europe's GDPR (General Data Protection Regulation) goes some way towards doing this. GDPR added a level of transparency to the field of data collection. Individuals must actively agree to their data being used, and this consent can be withdrawn at any time. Details must be permanently erased if consent is withdrawn. These regulations apply to all companies processing data of individuals living within the European Union, even if they're large multinationals headquartered in the U.S. As a result, companies like Facebook, Google and Amazon must adhere to these rules. The threat of massive fines for infringing ensures they keep an eye on where and to what ends their data is being used. I think that GDPR is a step in the right direction, but there's still a lack of transparency and some important questions need answering.

CONCLUSION

I think that measuring the performance of a software engineer can prove extremely beneficial. Management can track progress, evaluate effectiveness and use the data to aid management decisions. There are competent platforms and algorithms available for them to do so. Something that's crucial for this approach to be effective is **clarity**. The metrics that are going to be measured should be clearly established, the reasoning behind tracking such metrics should be firmly identified and this should all be communicated to the software engineers so that there's a level of transparency. If management fail to outline the purpose of gathering and analysing data, they'll waste time and resources. Ethical concerns must be taken into account, as it's not acceptable to have blurred lines between what's acceptable to measure and what's not. Laws need to state in black and white what companies can and cannot track.

BIBLIOGRAPHY

Lindquist, Matthew & Sauermann, (2015). Network Effects on Worker Productivity.

Fenton, Neil (1999). Software Metrics: successes, failures and new directions.

Grambow & Oberhauser (2013). Automated Software Engineering Process Assessment: Supporting Diverse Models using an Ontology.

Sillitti, Janes, Succi and Vernazza (2003). Collecting, integrating and analyzing software metrics and personal software process data.

Fenton & Bieman (2015). Software Metrics – A rigorous and practical approach.

Johnson, P. (2013). Searching under the Streetlight for Useful Software Analytics. IEEE Software, 30(4), pp.57-63.

Moore (2000). Lessons learned from teaching reflective software engineering using the Leap Toolkit.

Johnson, Zhang, Senin (2007). Experiences with Hackystat as a service-oriented architecture.

Watterson, W. (2018). When your boss knows everything about you. Available at: https://thenextweb.com/problem-solvers/2018/10/31/boss-knows-everything-people-analytics/ [Accessed 20 Nov. 2018].

(2018). *The Rise of Data Capital*. [online] Available at: http://www.oracle.com/us/technologies/big-data/rise-of-data-capital-wp-2956272.pdf [Accessed 20 Nov. 2018].

Castle, N. (2018). *Supervised vs. Unsupervised Machine Learning*. Available at: https://www.datascience.com/blog/supervised-and-unsupervised-machine-learning-algorithms [Accessed 20 Nov. 2018].

Kurkoski, J. (2018). *Hello science - meet HR*. Available at: https://ai.googleblog.com/2012/06/hello-sciencemeet-hr.html [Accessed 20 Nov. 2018].