Which machine learning method do you need?

Leandro L. Minku, Department of Computer Science, University of Leicester, UK

Learning Styles

An unarguable fact in teaching is that people do not all learn in the same way. Some of us spend hours reading instructions, whereas others just start trying things out to learn from the outcome. Some of us quickly adjust to new situations and technologies, whereas others tend to stick to the traditions. People also tend to use different approaches to learn different tasks. If we try to use an approach that is unsuitable for us, we will not be able to learn well or will take much longer time to learn.

This is no different in machine learning for software data analytics. There is a plethora of learning algorithms that can be used for several different purposes. Learning algorithms can give us insights into software processes and products, such as what software modules are most likely to contain bugs (Hall et al 2012), what amount of effort is likely to be required to develop new software projects (Dejaeger et al 2012), what commits are most likely to induce crashes (An and Khomh 2015), how the productivity of a company changes over time (Minku and Yao 2014), how to improve productivity (Minku and Yao 2014), etc.

Different types of learning algorithm are most adequate depending on the data and the environment being modelled. In order to create good data models, it is thus important to investigate the data analytics problem in hand before choosing the type of learning algorithm to be used. Here are a few useful questions to ask for that.

Do additional data arrive over time?

Databases containing software project data may not have a static size -- they may grow over time. For example, consider the task of predicting whether commits are likely to induce crashes (An and Khomh 2015). New commits and new information on whether they have induced crashes may become available over time. When additional data are produced over time, it is desirable to use such incoming data to improve data models. Online learning algorithms are able to update data models with incoming data in a continuous way. Chunk-based learning algorithms wait for a new chunk of data to arrive, and then use it to update the data models. Different from offline learning algorithms, online and chunk-based learning algorithms do not need to re-process old data or completely re-build the data model once new data become available (Gama and Gaber 2007). In this way, they are able to update data models faster. Given that re-building data models several times can be painfully slow when data sets are not small, this is particularly useful for larger data sets.

Are changes likely to happen over time?

Environments that produce data may suffer changes over time. For example, consider the data analytics tasks of software effort estimation (Minku and Yao 2012b) and software bug (defect) prediction (Ekanayake et al 2009). Software companies may hire new employees, may change their development process, may adopt new programming languages, etc. Such changes may cause old data to become obsolete, which in turn would cause old software effort estimation and software bug prediction data models to also become obsolete. Such changes may also bring back situations that used to occur in the past but were not occurring recently. Therefore, simply using all available data together can lead to contradictory and misleading data models. When temporal information about the data is available, change detection techniques can be used in combination with online or chunk-based learning algorithms (Gama and Gaber 2007) to handle changes. For instance, they can be used to (1) identify when a change that affects the adequacy of the current data model is occurring (Minku and Yao 2012a), (2) determine which existing data models best represent the current situation (Minku and Yao 2012b) and (3) decide how to update data models to the new situation (Minku and Yao 2012ab, 2014). When an environment has the potential to suffer changes, it is essential to collect additional data over time to be able to identify such changes and adapt data models accordingly.

If you have a prediction problem, what do you really need to predict?

In prediction / estimation problems, we wish to predict a certain category or quantity based on features describing an observation. It is important to decide what the target to be predicted really is, because this may affect both the predictive data models' accuracy and its usefulness. For example, we may wish to estimate the number of bugs in a software module based on features such as its size, complexity, number of commented lines, etc. For that, we should use a regression learning algorithm. Or, we may wish to predict whether or not a certain software module contains bugs. For that, we should use a classification learning algorithm. Or, we may wish to estimate the ranking of modules based on their bug-proneness. For that, we should use a rank learning algorithm. Depending on data availability, estimating the precise number of bugs may be more difficult than simply predicting whether or not a module is likely to contain bugs, but also less informative. If we do not really need to know the exact number of bugs, rank learning algorithms (Yang et al 2015) may be able to provide a balance between predictive accuracy and informativeness.

Do you have a prediction problem where unlabelled data are abundant and labelled data are

expensive?

In order to create predictive data models, supervised learning algorithms count with the existence of several past observations whose quantity / category to be estimated is known (labelled data). These learning algorithms are said to be taught these labels by a "teacher". Even though the existence of a teacher can help us to create good predictive data models, it is sometimes expensive to hire such a teacher, i.e., it is sometimes expensive to collect labels. This may result in few labelled data despite the existence of abundant unlabelled data, potentially causing supervised learning algorithms to perform poorly. An example of problem that suffers from this issue is software effort estimation, where the actual effort required to develop software projects is costly to collect, whereas other features describing software projects may be collected in an automated manner (Kocaguneli et al 2013ab).

Semi-supervised learning algorithms are able to use unlabelled data in combination with labelled data in order to improve predictive accuracy (Chapelle et al 2006; Kocaguneli et al 2013a). They typically learn the structure underlying the data based on the unlabelled data, and then combine this structure with the available labels in order to build predictive data models. If it is possible to request specific observations to be labelled, one may also opt for active learning algorithms. These learning algorithms are able to determine which observations are most valuable if labelled, instead of requiring all data to be labelled (Kocaguneli et al 2013b).

Are your data imbalanced?

In some cases, there may be abundant data representing certain aspects of an environment, but little data representing other aspects. Software bug prediction is an example of imbalanced problem, where there are typically less buggy software modules than non-buggy ones. When the data are imbalanced, learning algorithms tend to get biased towards the more common aspects and completely fail to model the less common ones. Learning algorithms specifically designed for imbalance learning should be used to deal with that (Wang and Yao 2012).

Do you need to use data from different sources?

Even though there may be little data from within the targeted environment, there may be more data available from other environments. Such data may be useful to improve data models for the targeted environment. For example, in software bug prediction, there may be little data telling whether modules within a given software are buggy, but a lot of data from other software. Learning algorithms able to transfer knowledge among environments can be used in these cases (Minku and Yao 2012b, 2014; Turhan et al 2009; Nam et al 2013).

Do you have big data?

Certain data analytics tasks may need to process large quantities of potentially complex data, causing typical learning algorithms to struggle in terms of computational time. This may be the case, for example, of modelling developers' behaviour based on software repositories hosting hundreds of thousands of projects. Online learning algorithms able to process each observation only once can help to build data models faster (Gama and Gaber 2007; Minku and Yao 2012a).

Do you have little data?

When there is not so much data to learn from, learning algorithms tend to struggle to create accurate data models. This is because the available data are not enough to represent the whole environment well. This is typically the case of software effort estimation data, but other software engineering problems may suffer from similar issues. When there is not much data, simpler learning algorithms that do not have too many parameters to be learned tend to perform better (Kocaguneli et al 2012).

In summary...

...examine your data analytics problem first, then chose the type of learning algorithm to consider! Given a set of learning algorithms to consider, it is also advisable to run experiments in order to find out which of them is best suited to your data.

References

- Le An and Foutse Khomh. An Empirical Study of Crash-inducing Commits in Mozilla Firefox. Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering, article no. 5, 10 pages, 2015.
- Olivier Chapelle, Bernhard Scholkopf and Alexander Zien. Semi-Supervised Learning. MIT Press, 2006.
- Karel Dejaeger, Wouter Verbeke, David Martens and Bart Baesens. Data Mining Techniques for Software Effort Estimation: A Comparative Study. IEEE Transactions on Software Engineering, vol. 38, no. 2, pages 375-397, 2012.
- Jayalath Ekanayake, Jonas Tappolet, Harald C. Gall and Abraham Bernstein. Tracking concept drift of software projects using defect

prediction quality. Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories, pp. 51-60, 2009.

- Joao Gama and Mohamed Gaber. Learning from Data Streams. Springer-Verlag Berlin Heidelberg, 2007.
- Tracy Hall, Sarah Beecham, David Bowes, David Gray and Steve Counsell. A Systematic Literature Review on Fault Prediction Performance in Software Engineering. IEEE Transactions on Software Engineering, vol. 38, no. 6, pp. 1276-1304, 2012.
- Ekrem Kocaguneli, Tim Menzies, Ayse Bener, Jacky Keung. Exploiting the Essential Assumptions of Analogy-Based Effort Estimation. IEEE Transactions on Software Engineering, vol. 38, no. 2, pp. 425-438, 2012.
- Ekrem Kocaguneli, Bojan Cukic, Tim Menzies and Huihua Lu. Building a Second Opinion: Learning Cross-Company Data. Proceedings of the 9th International Conference on Predictive Models in Software Engineering, article no. 12, 2013a.
- Ekrem Kocaguneli, Tim Menzies, Jacky Keung, David Cok, and Ray Madachy. Active learning and effort estimation: Finding the essential content of software effort estimation data. IEEE Transactions on Software Engineering, vol. 39, no. 2, pp. 1040-1053, 2013b.
- Leandro Minku and Xin Yao. DDD: A New Ensemble Approach For Dealing With Concept Drift. IEEE Transactions on Knowledge and Data Engineering, vol. 24, no. 4, pp. 619-633, 2012a.
- Leandro Minku and Xin Yao. Can Cross-company Data Improve Performance in Software Effort Estimation? Proceedings of the 8th International Conference on Predictive Models in Software Engineering, pp. 69-78, 2012b.
- Leandro Minku and Xin Yao. How to Make Best Use of Cross-company Data in Software Effort Estimation? Proceedings of the 36th International Conference on Software Engineering, pp. 446-456, 2014.
- Jaechang Nam, Sinno Pan and Sunghun Kim. Transfer Defect Learning. Proceedings of the International Conference on Software Engineering, pp. 382-391, 2013.
- Burak Turhan, Tim Menzies, Ayse Bener and Justin Distefano. On the Relative Value of Cross-Company and Within-Company Data for Defect Prediction. Empirical Software Engineering, vol. 14, no. 5, pp. 540-578, 2009.
- Shuo Wang and Xin Yao. Using Class Imbalance Learning for Software Defect Prediction. IEEE Transactions on Reliability, vol. 62, no. 2, pp. 434-443, 2012.
- Xiaoxing Yang, Ke Tang and Xin Yao. A Learning-to-Rank Approach to Software Defect Prediction. IEEE Transactions on Reliability, vol. 64, no. 1, pp. 234-246, 2015.