Review of Ensemble Learning Methods

Christian Caburian
COSC301

Overview

The explosion in popularity of machine learning algorithms is fueled by the age old fascination of merging man with machine, and how the results are often impressive and somewhat phenomenal. Artificial neural networks has become a popular example of the power in human simulation to devise statistical learning methods (Lantz, 2015). Rather than mimic biology, ensemble learning methods draw inspiration from the society of Ancient Greece. From the Enlightenment came the Condorcet Jury Theorem that proves the superior judgement of a jury over an individual authority, provided each juror is reasonably competent (Re & Valentini, 2014).

For many ensemble learning methods, it appears that the jury is still out on how and why they are so successful. Nevertheless, there appears to be a consensus on the success of ensemble learning methods on many sets of supervised learning problems, particularly those that involve structured data (Re & Valentini, 2014). According to Kaggle, the popular data science competition website, the ensemble method called xgboost outperformed non-ensemble methods by winning the most competitions last year for structured data challenges (Zhu, 2016).

Over time, ensemble learning algorithms have been spawned under and grouped into types of ensembles. An in-depth taxonomy of many of today's methods categorize each in a hierarchy of groups. In this review, algorithms will be examined under the three most popular types: boosting, bagging/random forests, and stacking. Boosting algorithms successively learn from each ensemble member, progressively attempting to tackle difficult areas to eventually converge on correct classification. Bagging and random forests treat each member independently, feeding randomly different versions of the training data to coax diverse opinions. Stacking expands on the concept of ensemble methods by combining ensembles to form a stronger ensemble (Re & Valentini, 2014).

The common practice of seeking a second opinion is motivated by the experience of receiving wildly varying advice from multiple experts on the same topic and the uncertainty associated in the scenario that they disagree. This is compounded when each expert has a paralyzingly high number of variables to consider, or when each expert has too few observations on which to base a decision. Often, a committee is the only option, where problems are so complex in nature that expecting reasonable performance from a lone expert is unrealistic. Experience may also have taught that some expert opinions should be valued more than others or that consulting a wide range of diverse opinions seems to build confidence when making a final decision. At this point, how does one reasonably factor in each and every expert opinion to reach a final decision? These concerns parallel those of the ensemble learning problem (Polikar, 2006).

Key features of ensembling

The most important principle for effective ensembling is that each learner makes different errors, otherwise, the ensemble would just be the equivalent to repeatedly asking the same person the same question. The attention to diversity remains somewhat notional; that is, although it can be the defining characteristic for certain types of ensembles, the explicit relationship between diversity and ensemble accuracy is yet to be identified (Polikar, 2012). Consider two experts of identical values, qualifications and experience. They are likely to give the same opinion if they're provided the same information. Likewise, the chance that their opinions diverge increases if they're provided different versions of the same information. The bagging ensemble method increases diversity this way, implemented by bootstrapping the data set so that each learner observes a (probably) unique random sample. Similarly, random forests provides different versions of the same information but extends this strategy by restricting each learner to consider a different subset of the variables. A more direct strategy involves employing a mixture of different types of algorithms. Manufacturing diversity in this manner has become almost a standard in situations where computational resources permit. In practice, most businesses don't consider the added complexity to be worth what may be slight gains in performance (Gorman, 2016).

The choice of individual learners is interrelated and therefore commonly dependent on the selected strategy for diversity. In bagging, a situation may arise where the difference between two samples is very slight, inducing the desire for greater diversity. This influences the choice of learner because selecting a particularly unstable learner that is more sensitive to small perturbations in the data creates higher variance between learners. Decision trees are a conventional choice in methods such as bagging for this reason, because modifying a few data points can dramatically alter the tree's decision boundary and thus its final decisions. For direct diversity strategies, the tunable parameters of neural networks offer the advantage of more methodical approaches to diverging each learner, such as assigning each learner a different number of hidden layer nodes (Polikar, 2012). In boosting, choosing an individual learner can be theoretically inconsequential, as long as it performs slightly better than a random guesser (Schapire, 1990). In similar cases, learner selection can move away from considering diversity and towards questions of convenience, convention, or application.

Finally, after all learners have had their say, a final decision must be made by somehow combining everyone's advice. The combination rule determines this, which is either trainable or non-trainable and ultimately dependent on whether each learner's output is continuous or not. In the trainable setting, combining learners usually involves some iterative process of tuning weights, or representations of importance, assigned to each learner (Polikar, 2006). The idea is to attribute varying importance to some learners over others. For example in AdaBoost, weights are used to focus learning efforts to areas in the feature space that are most difficult to classify. Real life is simulated here, where focusing on the harder parts of a problem converges more likely to a solution (Winston, 2010). In

stacking methods, a so called meta learner is trained on the performance of each ensemble member to determine their relative importance (Sill, Takacs, Mackey & Lin, 2009).

Non-trainable combination rules refer to the simple process of immediately reaching a final decision as a result of statically applying some algebraic formula on the learner outputs. This covers the simplest combination rule - simple majority voting - where the class that receives more votes than all others wins as the final decision. If the learner's output is continuous (e.g. the learner is a multi-layer perceptron network), then it is usually interpreted as an estimate of the posterior probability for that class. These can then be arranged into what's called a decision profile matrix that represents the degrees of support given by each classifier to each class. The many ways to combine each output has great impact as it can lead to significantly different final decisions using the same ensemble that is fed the same data (Polikar 2006).

Weak Learner Algorithms

Theoretically, many ensemble methods such as bagging and boosting, allow for a variety of algorithms to use as weak learners. The choice of certain algorithms over others is centered on the concept of encouraging diversity among their outputs. Decision trees are a popular choice for their inherent instability without any explicit tuning required to manually induce diversity. Generally, decision trees stratify the feature space in simple regions by performing recursive binary splitting on the data set. The end result can be visualised in a structure that resembles an upside down tree. The top is called the root node and its branches below represent the stratified regions known as terminal nodes or leaves (Nielsen, 2016).

For classification, the prediction is based on some measure that defines the best possible point to split the region into two classes. The most simple measure is the classification error rate, which is the proportion p of the observations in that region m that do not belong to the most common class (James, Witten, Hastie & Tibshirani, 2017). Here, k denotes the class belonging to p:

$$E = 1 - \max_k(\hat{p}_{mk}).$$

This measure greedily maximizes accuracy at each split, often resulting in fitting on noise that yields overfitted tree models. In practice, the Gini index is preferred which somewhat counteracts overfitting by measuring the total variance across the K classes (James et al., 2017). In other words, the purity of the node is measured:

$$G = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk}),$$

The most impure case would be when there is an equal number of observations for each class in a given region, resulting in G = 0.5. Perfect purity occurs when a node contains observations of a single class, i.e. G = 1 or 0 (James et al., 2017).

As the procedure for growing a tree is recursive, the number of times the feature space is split into new regions is an adjustable tuning parameter. This parameter is referred to as the tree depth because this controls the number of leaves a tree will have where a depth too large gives an overfit tree. For this reason, it is common to select a depth of one -- these trees are called decision stumps. In AdaBoost, using decision stumps is equivalent to omitting interaction terms in the model formula. This favours simplicity (as the model is additive) and works to keep overfitting at bay while relying on the ensemble's aggregation mechanism to converge to the optimal prediction (James et al., 2017).

Neural networks map a set of weighted input signals to a single output signal through a network of neurons that may have multiple layers. Generally, they are defined by three characteristics: an activation function, a network topology, and a training algorithm. The activation function refers to a mapping process that transforms a combined input into a single output. The network topology refers to how many neurons there are, how they are connected, and how many layers are used in its structure. The training algorithm refers to the procedure of assigning the weights that control the effect each has on the activation function (Lantz, 2015). A tractable approach to building an ensemble of neural networks involves holding each of these characteristics constant in each network. Diversity can be achieved by setting a randomly different set of initial weights for each network. It has been found that applying this approach without bagging (i.e. training each network on the same data set) often produces results as good as if bagging was applied using neural networks or decision trees (Opitz & Maclin, 1999).

Neural networks are also a popular choice as a weak learner for their potential instability and high variance. Compared to decision trees, these algorithms offer more options for tuning which opens up the ability to manually induce diversity. However, this added control comes at a cost of added complexity that often does not translate to worthwhile performance gains. However, neural networks can at times be simply better suited for some types of data to warrant such a tradeoff. An empirical comparison of neural networks and decision trees as weak learners in boosting and bagging showed that one's prediction performance is not conclusively better than the other. In many cases, the learner that performed better on data over the other was found to form an ensemble that performed better than its counterpart. This suggest that performance of the ensemble depends on the selection of learner algorithm and more importantly its performance as a lone classifier on specific types of data (Opitz & Maclin, 1999).

K-nearest neighbours (KNN) is a clustering algorithm that distinctly segments the feature space based on some distance metric between observations. KNN can either be used in an unsupervised setting for exploratory clustering or in a supervised learning setting as a weak learner that can segment based on classes of observations. The algorithm works by progressively assigning each observation to a distinct class by recursively comparing pairwise distances. Running the algorithm a second time after slightly changing some observations (e.g. as a result of bootstrapping) often results in very little change to the resulting classifications (James et al., 2017). This makes KNNs fairly stable and robust

against resampling in comparison to decision trees and neural networks. However, KNN exhibits instability when the subspace of features differs between learners. Therefore, KNNs can be made viable to great effect in certain relevant methods. The Random KNN is one such ensemble method that randomly feeds each of its KNN members a different feature subspace. Unlike other similar methods, it does not require the data to be bootstrapped, greatly simplifying its implementation. For high dimensional problems that suffer from small sample sizes, the Random KNN method has been found to perform faster and more stable than the Random Forest method discussed in the next section (Li, Harner & Adjeroh, 2011).

Bagging and Random Forests

Bootstrap aggregating, or bagging, is the simplest ensembling technique that rests on the same solid mathematical foundation as the well-established bootstrapping technique. Bagging has proven so powerful on real world applications that combining some form of it with other methods remains prevalent on popular complex problems. For a given training data set, bootstrapping is applied by repeatedly taking random samples and training each learner on an independent sample. Being based on bootstrapping, randomness is injected such that the enhancement of diversity in the ensemble is statistically tractable with useful side effects that permits convenient testing. Diversity can be further enhanced by selecting a notably unstable learner, where each learner makes a notably more varied decision based on slightly different versions of the same data set. Although bagging can theoretically accept any type of learner for an ensemble, decision trees have become the conventional choice for their high sensitivity to slight variation in data (Re & Valentini, 2014).

Motivated by the trend of increasingly high dimensional data, random forests extended the method of bagging to enhance diversity by appending the practice of randomly restricting the number of input variables each learner considers. For example, in an election of a president, the attribute of campaign contributions may be hidden from some learners, and the attribute of the candidate's stance on a particular issue may be hidden from other learners. Surprisingly, randomizing which attributes are hidden from which learners improves over bagging (Breiman, 2001). This appears particularly pronounced in the event that the data consists of some strong variable that dominates over all others. A significant proportion of learners will excessively focus on this strong variable, when it may be desired that they attribute a fair degree of importance to all variables. In the bagged setting, each learner considers all variables, meaning that each learner sees the strong variable and is therefore inclined to recognize its strength. In the random forest setting, only a random fraction of learners even know that the strong variable exists. Those learners that see the strong variable are decorrelated to those learners that don't. Under this setting, the process of averaging, or taking a majority vote, of the decorrelated learner's outputs produces a more reliable final decision as a consequence of less variability (James et al., 2017). It has been argued that this decorrelation effect is an important factor on prediction performance in applications such as medical diagnosis and document retrieval. Here, not only are there usually thousands of variables, each of them contributes a small amount of information that other statistical learning methods struggle to usefully account for (Breiman, 2001).

Fitting a random forest model is simple, requiring attention to adjusting only a few parameters. In practice, the most impacting parameters are the number of trees and the number of variables in the random subset. For bagging, the subset covers all variables (the subset size is tuned to equal the number of variables) and so only the number of trees requires tuning. Generally, it has been found that performance grows with the number of trees, raising two concerns; one of computational practicality and one of overfitting. Conveniently, it has been found that bagging and random forests are robust to overfitting and so the concern will usually focus on the minimum trees that produce acceptable performance under computational limits and application requirements (Breiman, 2001). To this end, the best way to determine this minimum is to compare performance of a forest to the performance of a subset of the forest. The optimal number of trees is the number used in the subset that performs comparably well with the full forest (Liaw & Weiner, 2002).

A similar approach is taken to select the number of variables for each tree, the subset size p. The suggested starting point is p/3 for regression and $p^{1/2}$ for classification. The random forest is then tested using this subset size, half of this size, and then double the size, to find which performs the best. When p is very large, and when most variables are expected to be relatively insignificant, larger subset sizes are recommended to give better performance. In rare cases, one might find that restricting this subset to one variable gives good performance for some data (Liaw & Weiner, 2002).

The randomForest package in R provides a user-friendly interface to run this ensemble method along with added conveniences for tuning performance and forest analyses. Due to its random handling of variables, bagging and random forests can be difficult to interpret if approached like single decision trees. Included in this package is the capability to determine the relative importance of certain variables over others. This is achieved by measuring the changes in performance as a consequence of entirely omitting certain variables to produce a random forest. Important variables are identified by marked increases in performance as a result of their omission. Adding these capabilities to the powerful prediction ability of random forests adds an attractive advantage over other ensembling methods (Liaw & Weiner, 2002).

While the era of big data has pressured computational power to exponential growth, the accompanying explosion of embedded devices has simultaneously demanded that computational resources are efficiently utilized. The relative simplicity and embarrassing parallelism of the random forest algorithm leverages advantages where computing memory is constrained. The algorithm Random Patches proposes that the subsetting nature of random forest can be exploited to efficiently segregate processing to sections of data. Under significant memory constraints this has been found to perform at least as good as unconstrained, standard random forests (Louppe, 2014).

Perhaps the most famous example of a successful application of the random forest is in its use in human pose detection for the gaming system Microsoft Kinect. This system has

several strict operating requirements. First, it must run on consumer hardware; considerably less powerful than the usual computing resources employed in statistical applications. Second, it must be responsive in real-time, where the margin for acceptable lag is extremely narrow. Finally, there is a degree of accuracy that, although permitted to be imperfect, ranks close to responsiveness in its impact on user satisfaction; interpreting a foot as an arm would be considered an utter failure. Random forests of decision trees were selected for this application to great success that has been attributed to its capacity for fast and efficient implementation on the GPU architecture accessible at a consumer level (Shotton et al., 2013).

Alongside its competitive prediction accuracy, model interpretability has seen this method endure as a popular machine learning algorithm. Particularly in the field of bioinformatics and among the computational biology community, random forests remain an intensely active area of research where problems often involve quantifying the importance of thousands of variables to select the features that matter. The high potential for complex interactions compound the difficulty. Problems such as gene expression classification, biomarker discovery, and statistical genetics are some examples of highly complex statistical problems that have been handled exceptionally well by the random forest method and its variants (Qi, 2012).

Boosting and AdaBoost

Boosting is perhaps the most historically important concepts in the larger field of machine learning. From its inception in the early days of ensembling, the concept of boosting has remained one of the best performing ideas today. Many variants have spawned from boosting, some of which to great success such as the gradient boosting method in R (gbm) and the multiple competition winner xgboost. Its importance in the larger field of machine learning has seen its original implementation, AdaBoost, used as a conventional benchmark in the research community, consequently subjecting it to intense theoretical study and empirical testing (Schapire, 2012).

The essential distinction between boosting and other ensemble methods is that each learner is dependent on other learners in the same ensemble. However, it is still built on the fundamental ensembling principle that individual learners are better than random guessers, and that the aggregation of these so-called weak learners produces a relatively strong classifier. Unlike forests, the notion of the relationship between learners is sequential, where instead of each learner being considered its own instance in a forest of similar instances, the learners in boosting are considered to successively build upon previous learners. To achieve this, the designated weak learning algorithm is repeatedly called. After each weak learner decision, the next learner will be trained on a modified version of the training set. This new version is chosen such that the next learner is forced to train on a relatively difficult area of the problem. The way in which this is handled is by assigning heavier weights to the distributions of those learner's that produced misclassifications; this is the key its success. For example, when a human learns a language, practicing a conversation

using words that the student considers difficult ensures that new words will be needed on a regular basis to maintain a state of learning. This leads to a progressive exploration of the entire language. Otherwise, practicing easy words leads to equivalently practicing the same words, eventually leading to a state that cannot be considered learning (Schapire & Freund, 2012).

Interestingly, AdaBoost continues to learn past the point of mastery. When the training error of a boosting session reaches 0%, it is intuitive to assume that there will be no more improvements to be made on further sessions. But much like a language expert makes slight gains in fluency when comfortably conversing in a well-practiced language, AdaBoost also improves its confidence after it has perfected its accuracy. This behaviour empowers AdaBoost with resistance to overfitting and a keen sense for distinguishing outliers (Schapire & Freund, 2012). More importantly, these properties hold in the high dimension setting, where other methods either struggle and/or require approaches fundamentally more complex that are dependent on inter-learner interactions. AdaBoost's algorithm aggregates learner results in an additive manner. At the same time, it does not rely on the relative distances between data points. Boosting works to distinguish data points through adaptive adjustment of neighbourhoods defined by learned boundaries. That is, the feature space is initially attacked globally, then certain areas are progressively identified as interesting neighbourhoods to focus on. This gradual increase in model flexibility through feature space localization proves to be the effective factor in beating the curse of dimensionality (Nielson, 2016).

Although AdaBoost theoretically specifies no tuning parameters, the algorithm's inherent weaknesses call for some attention to certain details that can affect its performance in practice. Depending on the nature of the data and the application, test performance may vary under different learning conditions. Primarily, the controllable learning conditions that have the most impact are the number of repetitions to call the base learner, and the rate at which the ensemble learns. The base learner should be called a sufficient number of times to achieve good performance, but restricted before signs of overfitting appear. This is interrelated to the rate of learning, because the faster the ensemble learns, the higher likelihood that overfitting is encountered prematurely. Conservatively choosing a slower rate provides some assurances that the algorithm detects subtler features of the unknown distribution while keeping overfitting at bay (Schapire & Freund, 2012).

In practice, it is suggested to base the number of repetitions on the rate of learning; slower rates call for more repetitions to reach comparable performance to sessions that used faster rates with less repetitions. These precautions serve to compensate for known AdaBoost issues that arise from its dependency on the data and its learners. Boosting simply fails when the amount of data is insufficient or excessively noisy. This responsibility falls on the user, and its only solution involves directly addressing the problem as opposed to attempts to compensate via tuning parameters. Crucial to this concern is that the data has been transformed accordingly with respect to relative scalings of the features (Schapire & Freund, 2012).

The other inherent weakness to boosting comes from its dependence on the quality of the individual learner. Learners that are overly complex or fundamentally weak invalidate the essential assumption that each learner must be sufficiently competent. This responsibility is perhaps easier to manage, and although packages usually provide simple measures to enable control over this concern, it usually does not require attention because package authors design for this concern “out of the box” (Ridgeway, 2012).

The gbm package in R is a popular implementation of the AdaBoost algorithm and conveniently enables the tuning of these controllable learning conditions. The rate of learning is adjusted by the shrinkage parameter, lambda, with a default setting of 0.001. It has been shown that setting this as high as 0.2 can lead to a slightly lower test MSE than the default setting (James et al., 2017). As mentioned, this parameter should be adjusted in coordination with the number of base learner repetitions, similarly easy to tune in this package.

Additionally, this package enables the user the control over the learner complexity issue. Because the learners are decision trees, adjusting learner complexity corresponds to the adjustment of tree depth. Because AdaBoost follows an additive model, this corresponds to a tree depth of one, that is, the decision trees are constrained to being decision stump. Setting this greater than 1 introduces interactions, and along with it added complexity. Therefore, this parameter must be carefully accounted for and its adjustment must be reasonably justified according to the context of the data (Ridgeway, 2012).

Partial dependence plots can also be provided that illustrates the marginal response effect of certain variables after they are integrated out. Similar to variable importance in random forests, these plots enable interpretation of the relative importance of certain variables over others, and can serve to perform feature selection (Tibshirani, 2015).

Boosting and AdaBoost have become the standard benchmark in the literature of ensembling algorithms, so much so that its historic academic application could be argued to be its most successful application. This has been extended into the data science competition community, where the most successful ensembling method to this date is considered to be the boosting variant known as xgboost. The popular data science competition website Kaggle profiled their top performers, and they reveal that although algorithm selection depends on application, a common favourite to pick as the best is xgboost. The key distinction in this variant is that tree depth is allowed to vary between trees and the degree of randomization between trees can be tuned; both serving to further decorrelate trees to improve diversity. Further, neighbourhoods are better determined because xgboost employs Newton boosting rather than gradient boosting, resulting in a higher likelihood of learning more appropriate tree structures. The competition record speaks for itself: 17 out of 29 kaggle challenge winners of 2015 used xgboost, followed by the tremendously more complex method of deep neural networks (used 11 times). In the same year, all top 10 solutions in the KDD Cup used xgboost (Nielsen, 2016).

The benchmark designation appointed to boosting and AdaBoost has ensured its presence and influence in a vast array of fields with difficult classification and regression learning problems. One of the most popular of these is the field of computer vision noted for its unique complexity and the associated fascination with modelling human behaviours. Reflecting this is the tremendous contribution made to the OpenCV framework, an open source programming library created by Intel. The libraries in-built face detection component, called the Haar Classifier, is based on boosting. Within this component, AdaBoost is found to be used in a cascading arrangement that performs effectively in locating human faces in a visual image. The documentation for OpenCV claims that (at the time) this simple implementation of AdaBoost can compete with the best available face detection algorithms in terms of accuracy with the added advantage of fast running times (Bradski & Kaehler, 2008).

Stacking

Stacked generalization, or stacking, expands on the concept of ensembling by using a similar argument; that an ensemble (or stack) of ensembles can improve on a single ensemble. The key idea in stacking is that the ensemble integrates a second layer of learning with the capacity to determine an optimal combination of learners. Under this framework, the performance of each learner is individually and distinctly considered in such a way that mathematically encourages the employment of different types of algorithms for the base learners. Naturally, the method evolved to additionally incorporating a mixture of entirely different ensembling algorithms as base learners. Because of this, stacking has become a stalwart consideration to improve any ensembling application in situations where computational resources are abundant (Risdal, 2017).

The early stacked generalization methods involve an additional learning layer, or meta-learner, that uses the outputs of the base learners as input features. Rather than simply aggregating their results into a final decision, through some untrained scheme (e.g. majority voting), the second tier meta-learner is trained on the behaviour of the ensemble itself. This is made possible by applying the base learners to the training data in a cross validation arrangement, where each learner has designated some unique portion of the data for testing and reporting its performance. After the base learners are trained, predictions are then generated and supplied alongside the correct classifications to form the training dataset used by the meta-learner (Wolpert, 1992).

An improvement of the original stacked generalization method proposed that since the predictions are supplied with the known answers, they can be used to calculate the degrees to which the incorrect predictions differ from the correct answers. This amounts to supplying the meta learner with a confidence associated with each learner's predictions (Džeroski & Ženko 2004). Each learner's confidence is represented by a probability distribution over the possible output class values that are determined by the highest probability. The meta-learner then aggregates these results by constructing a

multi-response linear regression model that assigns a separate regression problem to each, and in place of, the original output class values. Each problem is formulated to predict a binary variable that indicates if the corresponding class has been classified or not. Classifying a new data point then involves solving each of these regression problems on that data point. Thus, that which outputted the highest value corresponds to the class that becomes the final decision (Ting & Witten, 1997).

The H2O framework is a popular open source machine learning library that implements stacking with support for the powerful distributed computing platform Hadoop. Because stacking can incorporate different algorithms for base learners, the selection and tuning of the learners to construct the ensemble can be considered as tuning parameters. In H2O, the parameters available for tuning are primarily related to the meta-learner because the user is externally responsible for the set of base learners, or base models, used to construct the ensemble. Prior to stacking in H2O, the user is expected to train and cross-validate the set of base models to generate the training data frame used by H2O's meta learner. Options for the meta-learner algorithm include generalized linear models with or without non negative weights, gradient boosting, random forest, and deep learning. Depending on this selection, certain parameters relevant to the model are available for further tuning such as tree size and depth if the boosting method is selected (LeDell, 2017). Although stacking can improve performance in almost all instances where ensembling is used, the intense computational power that can be required can often render training times too long to be practical. H2O can mitigate these weaknesses by leveraging the power of distributed computing (LeDell, 2015).

A particularly lucrative application of ensemble learning has been in the area of customer preference prediction. The daily availability of enormous amounts of new data has become part of the problem, where the separation of noise, and the identification of useful information is a constant struggle. Stacking has proven to be one of the few available statistical methods to perform acceptably well in tackling these modern problems. A notable example of a successful application was the Netflix Prize. This was an open competition for the best collaborative filtering algorithm to predict user ratings for films, based on previous ratings without any other information about the users or films. The winner was awarded one million dollars for an impressive model, made up of a stacked ensemble of stacked ensembles (Sill et al., 2009).

The second place model (considered as good as the winner) also utilized a method of stacking, known as feature-weighted linear stacking that linearly combines model predictions using linear functions of meta-features as coefficients. The number of user and movie ratings, the number of items the user rated on a particular day, the date to be predicted, and various internal parameters extracted from some of the recommendation models were used as meta-features in these models. Complex feature interactions were modelled, such as the fact that the information gathered about a particular movie depends not only on the number of users who rated it but also on whether or not those users have rated many movies. Incorporating these meta-features involved sub-stacks that employed a

neural network or a gradient boosted decision tree. Linear regression, model trees, and bagged model trees were used as stacking algorithms. The meta-features that were found to show the most benefit were the number of user ratings and number of item ratings. The winning model made use of only these two meta-features, where each movie was specified its own coefficient, and parameters were combined by employing a stochastic gradient descent approach rather than a linear regression approach (Sill et al., 2009).

References

Bradski, G., & Kaehler, A. (2008). *Learning OpenCV*. Sebastopol (CA): O'Reilly.

Breiman, L. (2001). Random Forests. *Machine Learning*, *45*(1), 5-32. doi: 10.1023/a:1010933404324

Džeroski, S., & Ženko, B. (2004). Is Combining Classifiers with Stacking Better than Selecting the Best One?. *Machine Learning*, *54*(3), 255-273. doi: 10.1023/b:mach.0000015881.36452.6e

Gorman, B. (2016). A Kaggler's Guide to Model Stacking in Practice. Retrieved from http://blog.kaggle.com/2016/12/27/a-kagglers-guide-to-model-stacking-in-practice/

James, G., Witten, D., Hastie, T., & Tibshirani, R. (2017). *An Introduction to Statistical Learning* (6th ed.). New York [etc.]: Springer.

Lantz, B. (2015). *Machine Learning with R* (2nd ed., pp. 219-239). Birmingham: Packt Pub.

LeDell, E. (2015). *Scalable Ensemble Learning and Computationally Efficient Variance Estimation* (PhD). University of California.

LeDell, E. (2017). Stacked Ensembles — H2O 3.20.0.2 documentation. Retrieved from http://docs.h2o.ai/h2o/latest-stable/h2o-docs/data-science/stacked-ensembles.html

Liaw, A., & Weiner, M. (2001). Classification and Regression by RandomForest. *R News*, *2*(3), 18-22.

Liaw, A., & Weiner, M. (2002). Classification and Regression by RandomForest. *R News*, *2*(3), 18-22.

Louppe, G. (2014). *Understanding Random Forests: From Theory to Practice* (PhD). Université de Liège.

Li, S., Harner, E., & Adjeroh, D. (2011). Random KNN feature selection - a fast and stable alternative to Random Forests. *BMC Bioinformatics*, *12*(1), 450. doi: 10.1186/1471-2105-12-450

Nielsen, D. (2016). *Tree Boosting With XGBoost - Why Does XGBoost Win "Every" Machine Learning Competition?* (Master). Norwegian University of Science and Technology.

Opitz, D., & Maclin, R. (1999). Popular Ensemble Methods: An Empirical Study. *Journal Of Artificial Intelligence Research*, *11*, 169-198.

Polikar, R. (2006). Ensemble based systems in decision making. *IEEE Circuits And Systems Magazine*, *6*(3), 21-45. doi: 10.1109/mcas.2006.1688199

Polikar, R. (2012). Ensemble Learning. In C. Zhang & Y. Ma, *Ensemble Machine Learning* (pp. 1-34). New York: Springer.

Qi, Y. (2012). Random Forest for Bioinformatics. In C. Zhang & Y. Ma, *Ensemble Machine Learning* (pp. 307-323). New York: Springer.

Re, M., & Valentini, G. (2014). *Advances in Machine Learning and Data Mining for Astronomy*(pp. 563-594). Milan: Chapman & Hall.

Ridgeway, G. (2012). Generalized Boosted Models: A guide to the gbm package. Retrieved from https://pdfs.semanticscholar.org/a3f6/d964ac323b87d2de3434b23444cb774a216e.pdf

Risdal, M. (2017). Stacking Made Easy: An Introduction to StackNet by Competitions Grandmaster Marios Michailidis (KazAnova). Retrieved from http://blog.kaggle.com/2017/06/15/stacking-made-easy-an-introduction-to-stacknet-by-competitions-grandmaster-marios-michailidis-kazanova/

Schapire, R. (1990). The strength of weak learnability. *Machine Learning*, *5*(2), 197-227. doi: 10.1007/bf00116037

Schapire, R., & Freund, Y. (2012). *Boosting: Foundations and Algorithms*. Cambridge, Mass.: MIT Press.

Shotton, J., Sharp, T., Kipman, A., Fitzgibbon, A., Finocchio, M., & Blake, A. et al. (2013). Real-time human pose recognition in parts from single depth images. *Communications Of The ACM*, *56*(1), 116. doi: 10.1145/2398356.2398381

Sill, J., Takacs, G., Mackey, L., & Lin, D. (2009). Feature-Weighted Linear Stacking. *Arxiv E-Prints*, (0911.0460). Retrieved from https://arxiv.org/abs/0911.0460

Ting, K., & Witten, I. (1997). Stacked Generalization: when does it work?. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (II)* (pp. 866-871). Nagoya: International Joint Conferences on Artificial Intelligence.

Winston, P. (2010). *Lecture 17: Learning: Boosting - MIT OpenCourseWare*. Lecture, Cambridge.

Wolpert, D. (1992). Stacked generalization. *Neural Networks*, *5*(2), 241-259. doi: 10.1016/s0893-6080(05)80023-1

Zhu, N. (2016). XGBoost: Implementing the Winningest Kaggle Algorithm in Spark and Flink. Retrieved from https://www.kdnuggets.com/2016/03/xgboost-implementing-winningest-kaggle-algorithm-spark-flink.html