Documento de Trabajo

Machine Learning for Applied Economic Analysis: Gaining Practical Insights

MATTHEW SMITH
FRANCISCO ÁLVAREZ

Documento de Trabajo 2025/03
Abril de 2025

fedea

Las opiniones recogidas en este documento son las de sus autores y no coinciden necesariamente con las de Fedea.

Machine Learning for Applied Economic Analysis: Gaining Practical Insights*

Matthew Smith† Francisco Alvarez ‡

March 25, 2025

Abstract

Machine learning (ML) is becoming an essential tool in economics, offering powerful methods for prediction, classification, and decision-making. This paper provides an intuitive introduction to two widely used families of ML models: tree-based methods (decision trees, Random Forests, boosting techniques) and neural networks. The goal is to equip practitioners with a clear understanding of how these models work, their strengths and limitations, and their applications in economics. Additionally, we briefly discuss some other methods, as support vector machines (SVMs) and Shapley values, highlighting their relevance in economic research. Rather than providing an exhaustive survey, this paper focuses on practical insights to help economists effectively apply ML in their work.

1 Introduction

The central question in empirical economic analysis is how to use data to uncover relational structures between variables. The primary goal may be to understand the structure itself, as in causal inference, or to use it for prediction, policy implications, or theory testing.

Over the past decades, the standard econometric framework for data analysis in economics has been challenged by two major developments. First, we have witnessed an explosion in the availability of new datasets, often accessible with just a few clicks. Notably, the rise of microdata (covering individuals or households), high-frequency data

*We thank Diego Rodriguez for very helpful comments. Any remaining error is our responsibility. Smith is supported by Ramón Areces Foundation grants for the employment of young people with a PhD. Alvarez is supported by the grant PID2023-147391NB-I00 funded by MCI-U/AEI/10.13039/501100011033/FEDER, UE

†Research Group, Esade Business School, Universitat Ramon Llull, Av. de Pedralbes, 60-62, 08034 Barcelona; e-mail: mattonline1@gmail.com

‡Department of Economic Analysis & ICEI, Universidad Complutense, 28223 Pozuelo de Alarcón, Madrid, Spain; e-mail: fralvare@ccee.ucm.es

(for time series analysis), and unstructured data (such as text) has introduced a wealth of information with substantial economic relevance, while also demanding more flexible analytical tools. Second, there has been an unprecedented increase in computational power, whether from standalone machines or distributed networks, with cloud services like AWS making large-scale computation widely accessible at relatively low cost.

Machine learning (ML) has emerged as a natural response to this evolving landscape, offering a broad set of classification and prediction methods. While closely related to both traditional econometrics and other AI techniques, ML is often distinguished by its focus on predictive accuracy rather than causal inference. See [7], [6], [37], [57] and [70].

In this paper, we introduce some of the most widely used ML methods in economics. Rather than aiming for an exhaustive survey, we prioritize an intuitive explanation of their essential components, working under the premise that practitioners benefit most when they understand the fundamental logic behind each method. We focus on two major families of models, tree based methods and neural networks, and provide basic Python examples to illustrate their implementation. Once these foundations are in place, practitioners can more easily navigate the vast array of available ML techniques and applications.

It is fair to mention at least two areas of ML or related literature that this paper does not cover, even though their relevance to economics is rapidly growing. First, in this paper, we focus on prediction and classification problems, both of which constitute an *ex-post* analysis. Even when the aim is to predict the future, our approach relies on analyzing past data. In contrast, a branch of ML known as *Reinforcement Learning* explicitly addresses problems where decision-making and learning interact dynamically over time,[1] [53]. Second, we do not deal with extracting information from unstructured data, such as text mining. A seminal work on *Latent Semantic Analysis* is [32], which laid the groundwork for *Natural Language Processing* (NLP)—a field that, in our view, belongs to a distinct domain of research. In addition, the work by [22] surveys the anomaly detection, which this paper does not cover and, obviously, can distort any data analysis.

While we provide some additional model-specific references along the text, again without being exhaustive, some ideas on the overall applicability of ML are in order here. The relative advantage of ML often depends on both the nature of the data and the goal of the analysis. While some specific issues are addressed in further sections on regard to specific methods, some general comments on the applicability of ML are in order.

First, ML tends to perform well when working with large datasets, particularly in

[1]Reinforcement Learning has strong connections to classical branches of Optimal Control theory such as Active Learning. It also overlaps with some problems in Operations Research such as Multi-armed Bandit or the Exploration *vs* Exploitation trade-off.

cases where economic relationships exhibit significant nonlinearity or high-dimensional interactions. Traditional econometric models often impose strong parametric assumptions for tractability, whereas ML methods, particularly tree-based models and neural networks, can flexibly capture intricate patterns in the data without requiring explicit functional form assumptions [103]. This makes ML particularly useful for analyzing consumer behavior, labor market dynamics, and financial risk assessment, where complex interactions between variables are likely to exist.

Second, the strengths of ML are particularly evident in predictive tasks. Many economic applications involve forecasting rather than structural estimation, and ML has shown strong performance in domains such as inflation forecasting [67], credit scoring [61], demand prediction [38], measuring productivity [21] or agriculture [90]

Finally, ML methods often excel in rare-event prediction, where conventional econometric models may struggle due to class imbalance. Examples include predicting bankruptcy [111], fraud detection [105], and default risk [19]. Tree-based methods such as Random Forests and boosting techniques are particularly useful in these settings, as they can handle highly skewed datasets through resampling techniques and loss function adjustments [18].

There is a relatively large number of textbooks on ML as discipline. To the best of our knowledge, but also reflecting our preferences, [45] remains the classical reference, complemented by [72], which offers a very detailed and rigorous treatment of the statistical learning theory. This theory lays the foundations for all ML methods, which we do not cover in the paper.

The rest of the paper is organized as follows. The Section 2 introduces some basic concepts. Section 3 presents single tree models. Section 4 starts with multi-tree models, in particular Random Forest. Section 5 continues with multi-tree models, using boosting methods to combine trees. Section 6 deals with Artificial Neural Networks. Section 7 presents an XGBoost-based case study. Finally, Section 8 concludes.

2 Overall concepts

This section introduces some standard ideas that will ease the later presentation at further sections and will position ML with respect to classical Econometrics.

Econometrics typically starts with an *ex-ante*, relatively simple, and specific hypothesis about the underlying distribution of the data. It often assumes a particular model or functional form for how variables are related (e.g., linear relationships, specific error structures), and it aims to estimate parameters based on these assumptions. This model-driven approach is focused on understanding the population through the lens of a predefined statistical model.

In contrast, ML tends to be less specific *ex-ante*. Instead of assuming a particular distribution or model structure, ML methods often operate with more flexibility. They focus on finding patterns in the data without rigid assumptions about the underlying distribution or the exact nature of the relationships between variables. The approach is data-driven, emphasizing prediction and generalization based on the observed data rather than presupposing a specific form for the model.

Both Econometrics and ML are rooted in strong statistical foundations, but they emphasize different aspects. Econometrics is grounded in *statistical inference*, which focuses on drawing conclusions about population parameters based on sample data. It relies heavily on probability theory and statistical models to make inferences about causal relationships and to estimate the properties of the underlying population from which the sample is drawn. This involves testing hypotheses, estimating parameters, and making predictions with well-defined assumptions about the data generation process. ML, on the other hand, is built on *statistical learning*, which focuses on using data to learn patterns and make predictions. Statistical learning theory provides the theoretical framework for understanding how algorithms generalize from data and make predictions. While it also involves estimation (e.g., estimating a function that maps inputs to outputs), it typically does so with fewer *ex-ante* assumptions about the underlying data structure compared to Econometrics. ML emphasizes the performance of algorithms, particularly in terms of their ability to generalize from training data to unseen data.

Some comparison between standard notation in Econometrics and ML is useful. Let $\boldsymbol{x}$ denote a vector of variables we use to explain, classify or predict a another variable y. We will assume a dataset of paired observations $(\boldsymbol{x}_i, y_i)$ for $i \in \{1, \dots, N\}$ is available. The presentation of the different methods in this paper abstracts away from the nature of the index. While there are specific algorithms for either cross-section or time series, the core ideas -which are our target- apply equally to both dimensions. In Econometrics, it is standard to denote $\boldsymbol{x}$ as the vector of *regressors* or *explanatory variables*, whereas y is the dependent variable. In ML it is common to use the terms *attributes* and *response* for $\boldsymbol{x}$ and y, respectively. Additionally, each observation in ML is usually referred to as an *instance*. Finally, in ML is important to distinguish an attribute, which is an element of $\boldsymbol{x}$, say x_1, from a *feature*, which is a value (or set of values) of an attribute. For instance $x_1 = 5$, $x_1 \leq 3$ or $x_1 = \{Red\}$ are features. In the sequel we use indistinctly the Econometrics or the ML notation.

Roughly, ML searches within a space of functions the *best fit* for the data. That *best fit* minimizes some loss function. Methods differ from one another on the space of function under consideration, on the search method or in the definition of loss. In that search, some features are common to all of the ML methods considered in this paper. First, recall we are assuming a paired sample. Using ML terminology, we observe the response for each instance. This is called *supervised learning* and it is crucial as the loss

function usually contains some metrics of prediction errors.

Second, we must distinguish *parameters* from *hyperparameters*. The parameters are optimized along the search process, while the hyperparameters are set by the practitioner. In order to clarify this distinction, suppose that we restrict the space of functions to be linear, say, for a sample with two attributes and for each observation i:

$$y_i = \beta_0 + \beta_1 x_{1,i} + \beta_2 x_{2,i}$$

Like in classical Econometrics, let $\boldsymbol{\beta} := (\beta_0, \beta_1, \beta_2)$ be the vector of parameters of the model. The search consists of finding an optimal $\boldsymbol{\beta}$, according to some loss function. Consider, for instance, a loss function that penalizes two components: (i) the prediction error and (ii) the norm of the estimated parameter vector, denoted as[2] $||\boldsymbol{\beta}||$. Let $\hat{y}_i$ denote the prediction for an observation in which the attributes are $\boldsymbol{x}_i$, and let y_i be the corresponding actual response. A loss function that will be used is:

$$L = \sum_i (y_i - \hat{y}_i)^2 + \tau ||\boldsymbol{\beta}|| \tag{1}$$

The first term is known as the *mean squared error*. The weight τ measures the relative importance of the penalty on model complexity, measured by $||\boldsymbol{\beta}||$, compared to the prediction error. We say that τ is a hyperparameter: it is set by the practitioner and remains fixed throughout the search process, whereas the elements of $\boldsymbol{\beta}$ are the parameters that the algorithm optimizes, or learns. As a matter of fact, the loss function in (1) is at the core of some widely used models in ML and Econometrics, as we discuss in a later section.

Third, ML methods typically partition the available sample into a *training* and a *test* set: the former is used to train the model, while the latter evaluates its predictive performance on unseen data. However, a given split may introduce *selection bias*, which can result in an unbalanced representation of patterns between the training and test sets, potentially leading to misleading performance estimates. To mitigate this, we can use a *cross-validation* procedure.

In *k-fold cross-validation*, the sample is divided into k *folds*, typically with $k = 5$ or $k = 10$. Each iteration uses $k - 1$ folds for training and the remaining fold for testing, yielding a model along with its performance on that test fold. This process is repeated k *times*, each time selecting a different fold as the test set. As a result, we obtain k different performance evaluations. Instead of combining the k models, we *aggregate their performance metrics*, e.g., averaging accuracy or RMSE, to be explained in a further section, to estimate how well the model generalizes. This process helps in selecting optimal hyperparameters and assessing model generalization before training a final version

[2] Here we refer to the concept of norm rather than to a specific norm function. Sometimes the choice of the norm has important implications. If so, we will make explicit the norm under consideration.

on the full dataset, ensuring it benefits from all available data. The Section A, in the Appendix, contains some further details. Additionally, [58] analyses extensively this question.

A final concept that applies to all ML methods is *over-fitting*, a long-standing issue in Econometrics. Essentially, when a model has too many parameters, it can perfectly fit the training data, but at the cost of losing its ability to provide relevant insights into the target question. This idea directly parallels the problem in ML. In the space of possible functions, the search process might end up selecting an overly complex function, one that fits the training set perfectly but performs poorly on out-of-sample data. To combat over-fitting, many loss functions incorporate a penalty on the model complexity, as in the above presented loss function, helping to balance fit with generalisability. The opposite to over-fitting is under-fitting, and the trade-off between them is commonly referred to as *varianze vs bias*.

3 Single tree models

As mentioned, most of the paper is devoted to two families of models, namely, the tree-based and neural network models. This section begins with the first of those families. Our pace is deliberately slow. In this section we just deal with models consisting on a single tree. Further sections will consider the combination of trees to form more complex models.

3.1 Decision trees

A decision tree is a supervised learning model used for classification problems, popularized by [16]. Tree-based models partition the feature space into a set of rectangles and then fit a simple model in each one. The Figure 1 provides an example. Suppose our sample has paired observations of $\boldsymbol{x} = (x_1, x_2)$ and y, the latter being a *categorical* variable, say it can either be *green* or *red*. Roughly, our sample will look like the points plotted in the right panel of the figure. Now, how can we efficiently *classify* our sample in order to predict the response on the basis of observed values of x_1 and x_2? Graphically, we can partition our data set in rectangles as the figure shows. Notice that within each rectangle, approximately all points have the same color. In addition, more rectangles would add little (over-fitting), while less would make us incurr in too much of mixing colors within a given rectangle (under-fitting). Now, other than graphically watching, how can we write it down? We can use the decision tree shown in the left panel.

Generally speaking, decision trees are non-linear models made up of piecewise linear components in each neighborhood. It is useful in classifying non-linearly separable data in which the response variable is categorical, but not necessarily binary. When categorical, each possible value of the response variable is a *class*. As virtually all of the the other ML methods, they abstract away from the existence of a specific underlying

statistical model generating the data. They are also able to handle different data types in the attributes. Transformations of the variables are also not a requirement and it can therefore be useful in identifying outliers, variable interactions and important variables.

In mathematical terms, trees are a particular kind of *graphs*, which is the essential concept in *graph theory*. A tree is a collection of *nodes* connected by *edges* such that: there is a path between any two nodes, contains no cycles[3] and it is *hierarchical*, which means there is a unique *root* node, at which the tree starts, and several *child* -equivalently *parent*- and *terminal* nodes, at which the tree ends. If there is an edge that connects node V_0 to node V_h, we say that V_h is a child of V_0 or, equivalently, V_0 is the parent of V_h. A node is terminal if it has no children. In addition, the number of nodes is equal to the number of edges plus one, which requires that each node except the root has just one parent. The root node has no parent. Sometimes terminal nodes are also referred as *leaves*. As we see in Figure 1, nodes other than the terminal nodes contain features that define a line in the scatterplot of the right panel. Each terminal tree classifies or predicts a class for the instance that falls in it.

How trees are constructed? While this is mostly an internal question of the software being used, some ideas on the essential trade-off under consideration might be helpful to the practitioner. On the one hand, we want to minimize prediction errors, on the other hand, we want to avoid over-fitting. Clearly, the purpose is to classify, so that within each leaf, or terminal node, we would like to have as few different classes as possible. If at a given node all of the observations were the same class, there is no need to further split from it, that is, to generate more children from that node. A node in which all observations have the same class is referred to as a *pure* node. There are different measures of *impurity* of a node, such as *Gini impurity* or *entropy*, that are used to decide both, whether to continue splitting or not and, if continue, how to split.[4] But then comes the other element to balance: over-fitting. If we end up with a tree having as many leaves as observations, that is, each leaf has exactly one observation, all of the leaves are pure and there are no prediction errors in the training sample. We say the tree *remembers perfectly* that training sample, but very likely it will perform badly in the testing.

In order to avoid over-fitting, there is a number of hyperparameters can be set up, such as the *maximum depth* the tree can have, which can be measured as the maximum distance -number of nodes- from the root to a terminal node. In line with that, it can be

[3] More formally, acyclicity means there is no path that goes from node V_0 to itself such that: (i) it visits at least a node different to V_0 and (ii) no edge is walked more than once.

[4] Just as an example, let us assume only two categories are possible, say red and green, and let p_1 and p_2 denote the percentage of observations of each class in a given node. The Gini impurity (for a given node) is defined as $G := 1 - (p_1^2 + p_2^2)$. Thus, the impurity is zero, that is, $G = 0$ if either $p_1 = 1$, thus $p_2 = 0$, or reversely. Contrarily, the maximum impurity occurs at $p_1 = p_2 = \frac{1}{2}$. Roughly, the gain from further splitting from one node is the difference between the impurity of that node, which would be a parent node with respect to each of his children. The algorithms compute gains from all possible splits. Alternatively, the measure of impurity can be based on the concept of entropy. For any given node, assuming binary response, its entropy is $H = -(p_1 lnp_2 + p_2 lnp_1)$.

fixed the minimum number of observations each split and or each terminal node must have.

In addition, it is possible to *prune* the tree: to cut branches, or splits, that add little gain to the overall performance. The pruning can be done either during the construction of the tree or once it has been built.

Let us present a short review of literature connected to these ideas. [69] came up with one of the earliest concepts of a regression tree, the *Automatic Interaction Detection* (AID) which recursively splits data depending on impurity and stops splitting when a certain level of impurity is reached. [68] extended this idea to classification problems and developed *THeta Automatic Interaction Detection* (THAID) which recursively splits the data in order to maximise the number of observations in each modal class. [55] developed the *Chi-square Automatic Interaction Detection* (CHAID) originally developed for classification and then extended to incorporate regression problems. [16] improved on AID and THAID and developed the *Classification and Regression Tree* (CART) which improved accuracy through instead of using stopping rules, it grows the tree and then prunes it to a size that has the lowest cross-validation estimate of error. The *Iterative Dichotomiser 3* (ID3) [76] was later developed. It uses information entropy in order to quantify impurity and is used to compute the gain ratio for binomial decision classifiers. [77] developed the successor to the ID3 algorithm which can handle multi-class problems and laid the groundwork for further developments.[5] [64] provides a comprehensive literature review of the main developments in decision tree models over the past 50 years.

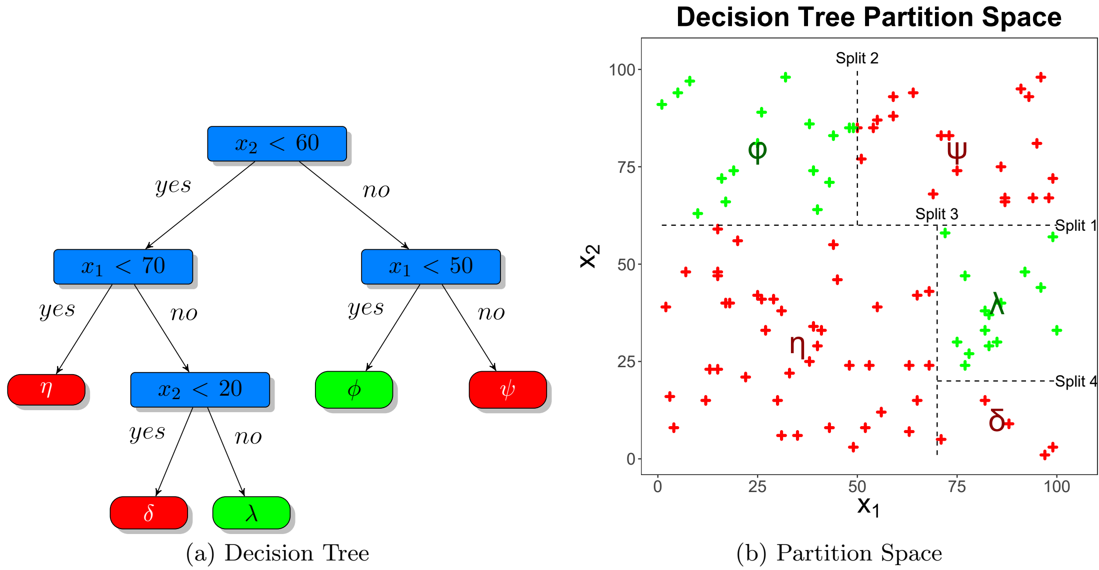
3.2 Confusion matrix

The previous subsection has presented the basic ideas on how decision trees are built and interpreted. Still, there is a question that remains: how to evaluate the performance of the model. This subsection presents the standard evaluation metrics for models with a categorical response variable. This metrics is called *confusion matrix*, and it applies -but is not necessarily restricted- to tree based models. For this reason, it is a natural follow up to the previous subsection, but it must be a subsection in itself. For sake of expositional simplicity, this subsection focuses on a binary response variable, that is, the response has two possible classes, but the main concepts extend straightforwardly to multi-class models. In order to use standard notation, the classes will be *positive* and *negative*.[6]

The confusion matrix is depicted in Table 1, see [74]. For a binary response, it is a

[5] We reference the original 1993 edition, which remains the standard citation, although subsequent editions have been published.

[6] This notation is particularly suitable for economic applications in which, roughly speaking, one of the classes has some strong implication. For instance, the state of a firm might be either *bankruptcy* or *healthy*, and we train a model that aims to effectively classify firms in order to predict bankruptcy. The bankruptcy state has the strong implication, and that is the *positive* class.



(a) Decision Tree (b) Partition Space

Figure 1: Decision tree and corresponding scatter plot partition space

2×2 matrix, by rows it has the actual classes whereas by columns it has the predicted ones. For instance, upper left entry is the number of observations for which the actual class is positive and the predicted one is also positive. The diagonal elements contain the number of observations correctly classified: *true positives* in the upper left and *true negatives* in the lower right entries, denoted as T_p and T_n, respectively. The off-diagonal elements contain the possible errors. If the model predicts as negative an observation that is actually positive, that is a *false negative*, or F_n, in the upper right corner. Analogously, the *false positives*, or F_p, are the lower left entry.

The confusion matrix in the table is enlarged with some standard statistics computed from the entries of the table. The two key metrics are *precision* and *recall.* Precision measures out of all of the predicted positives, which are actually positive. Recall measures out of all of the actual positives, which are correctly predicted positive. Continuing the example in the previous footnote, suppose observations are attributes of firms, and the response is that each firm can be either bankrupt (positive) or healthy (negative). Most firms are healthy, so we assume healthiness unless the data provides enough evidence against it. In other words, the null hypothesis is H_0 : *"the firm is healthy"*. Then:

- Type I error or false positive: Wrongly predicting bankruptcy for a healthy firm (rejecting H_0 incorrectly). A high precision implies fewer Type I errors, that is, fewer false positives. Conversely, a low precision means more false positives.
- Type II error or false negative: Failing to predict bankruptcy for an actually bankrupt firm (not rejecting H_0 when we should). A high recall implies fewer type

II errors, that is, fewer false negatives. Conversely, a low recall means more false negatives.

		Predicted class		
		Positive	Negative	
Actual class	Positive	True positive T_p *(Correct)*	False negative F_n *(Incorrect)*	Sensitivity/Recall $\frac{T_p}{T_p+F_n}$
	Negative	False positive F_p *(Incorrect)*	True negative T_n *(Correct)*	Specificity rate $\frac{T_n}{T_n+F_p}$
		Precision $\frac{T_p}{T_p+F_p}$	Negative predictive value $\frac{T_n}{T_n+F_n}$	Accuracy $\frac{T_p+T_n}{T_p+T_n+F_p+F_n}$

Table 1: Confusion Matrix

A number of other metrics can be computed from the confusion matrix. For instance, when both types of errors are equally important, we may want a single number that balances precision and recall. The usual way to achieve this is by computing their harmonic mean, known as the *F1 score*.[7] Another common metric is *accuracy*, defined as the proportion of correctly classified observations over the total number of observations. However, accuracy is not always a reliable measure, specially when dealing with imbalanced datasets. For instance, if the percentage of bankrupt firms is very small, a trivial classifier that predicts all firms as healthy would achieve high accuracy while completely failing to detect bankruptcies, resulting in a very low recall.

We can take an slightly different approach in analyzing the trade-off among errors. What is the prediction of a decision tree? It might be, for each terminal node, the modal class, but perhaps it is more *neutral* to take the percentage of positives. More generally, the outcome from a model can be, and typically is, the -percentage- probability of positive for each terminal node. Let us denote that probability as $\hat{y}^{(k)}$ for the terminal node k, this predicted probability is usually called *score*. Now, actual values are either positive or negative. We code the classes as 1 or 0, respectively. How do we then compare scores to actual values? We define a threshold, say $y^* \in (0,1)$, such that the prediction from terminal node k, the score, is assigned to 1 if $\hat{y}^{(k)} \geq y^*$, while it is assigned to 0 otherwise. This threshold is chosen at random and, varying it, all else equal, the confusion matrix also varies. Since the threshold is arbitrary, it makes sense to plot, for

[7]The harmonic mean is particularly sensitive to very low values: if either precision or recall is close to zero, the F1 score is also very low, which discourages extreme imbalances between the two.

each value, the *true positive rate* (TPR), which is another name for recall, *vs* the *false positive rate* (FPR), defined as:

$$\text{FPR} = 1 - \text{Specificity} = \frac{F_p}{T_n + F_p}$$

As the threshold increases, the number of predicted positives decreases, so we reduce both T_p and F_p or, equivalently, we reduce both TPR and FPR, respectively. This creates the ROC curve,[8] an example of which is shown in Figure 2. In that curve, the diagonal corresponds to the outcome of a random classifier, that is, a classifier that predicts purely at random between positive and negative.[9] In order to measure the distance from the ROC curve to the diagonal, or alternatively, how close the ROC is to the perfect prediction, which is the upper left corner, where FPR=0 and TPR=1, we have the area under the curve (AUC), also shown in the legend of the figure.

Complementarily to the ROC, the precision-recall (PR) curve helps to evaluate the performance of a binary classification model. As for the ROC curve, the PR compares predicted scores to true labels by varying the threshold that transforms scores into binary predictions. Especially in imbalanced datasets, the PR curve provides a helpful evaluation of the model's performance. [39] provides an introduction to ROC, whereas [31] is a classical reference for the comparison between ROC and PR curves.

The Figures 2, 3 and 4 show, as an example, the ROC, PR and confusion matrix for the data in Table 2. The first row in the table contains observed values of the response, while each column in the second row shows the corresponding score, that is, predicted probability of a response $y = 1$. One of the points with this example is to illustrate that the computation of these metrics abstracts away from how the scores, predictions, have been computed. The Listing 1, in Section B of the Appendix, contains the Python code for generating these figures.

Observed	0	0	1	1	0	1	0	0	1	1
Score	0.1	0.4	0.35	0.8	0.2	0.9	0.05	0.5	0.7	0.6

Table 2: Input data for the ROC, PR and confusion matrix

3.3 Regression Trees

In essence, a decision tree is used whenever the response variable is categorical. Regression trees extend the main ideas outlined for decision trees to a continuous response

[8] ROC stands for Receiver Operating Characteristic. The name comes from the World War II, where a radar operator had to decide whether a signal came from, say, an enemy aircraft (positive), or it was just background noise (negative).

[9] Looking at the confusion matrix, a classifier that chooses randomly between predicting positive and negative would select $T_p = F_n$ and $F_p = T_n$, which trivially lead to $TPR = \frac{1}{2}$ and $FPR = \frac{1}{2}$, respectively.

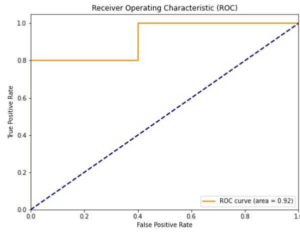


Figure 2: ROC

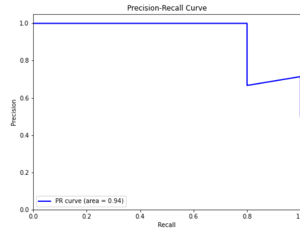


Figure 3: Precision-Recall

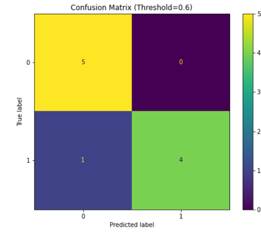


Figure 4: Confusion (threshold = 0.5)

variable.

As for the decision tree, the criterion to further split from a given node is based on a measure of heterogeneity of the observations lying on that node. As said, for a categorical response, the impurity accounts for the heterogeneity. In contrast, for a continuous response the standard measure is the *mean squared error* (MSE), defined for a node which has the set of observations $\{y_1, \ldots, y_N\}$ as:[10]

$$MSE = \frac{1}{N} \sum_{i=1}^{N} (y_i - \overline{y})^2 \tag{2}$$

where $\overline{y}$ is the average of the observations in the node. The gain from creating children from a parent node is measured as the difference between the MSE of the parent minus the weighted MSE of the children. Though there are some exceptions, both decision and regression trees either create exactly two children from any node, or the node is a terminal node. We must notice that any morphology with more than two children per parent has a classification-wise equivalent counterpart with exactly two children from each non-terminal node. Usually this restriction on the number of children makes the tree building process computationally efficient and helps to avoid over-fitting.

4 Random Forest

The previous section has been restricted to decision trees as standalone models. Yet, one of the most powerful aspects of ML is its ability to combine multiple models to improve accuracy and robustness. This approach, known as *ensemble learning*, leverages the strengths of individual models while mitigating their weaknesses. This section focuses on *Random Forests*.

Random Forest is a widely used ensemble learning method that builds on the principles of *Bagging (Bootstrap Aggregating)* to improve predictive performance and stability.

[10] Alternatively, we can use the *mean absolute error* (MAE), in which the absolute value replaces the square function, or the usual coefficient of determination.

By combining multiple decision trees, Random Forest reduces variance, enhances robustness and mitigates over-fitting.

First, M subsamples are bootstrapped with replacement from the training sample. Then a *single learner*, a tree, is trained on each of those subsamples, so that M trees learn independently from one another, tough the individual training samples might partially overlap. Let us denote the set of trees as $\mathcal{H} := \{h_1, h_2, \ldots, h_M\}$. Now consider an observation $(\boldsymbol{x}, y)$, where $\boldsymbol{x}$ is the vector of attributes and y is the response. Let $h_m(\boldsymbol{x})$ be the predicted response by a generic tree h_m in $\mathcal{H}$. The predicted response from $\mathcal{H}$, say $\mathcal{H}(\boldsymbol{x})$, is the aggregation of individual predictions. If y is categorical, $h(\boldsymbol{x})$ is categorical as well, and then $\mathcal{H}(\boldsymbol{x})$ might be constructed using a majority voting rule, that is, $\mathcal{H}(\boldsymbol{x})$ is the mode in $\{h_1(\boldsymbol{x}), \ldots, h_M(\boldsymbol{x})\}$. If y is continuous, then it aggregates by averaging, that is:

$$\mathcal{H}(\boldsymbol{x}) = \frac{1}{M} \sum_m h_m(\boldsymbol{x})$$

Notice that for the particular case in which the response is categorical and binary, there is no loss of generality in labeling the possible responses as 0 or 1, so that $h_m(\boldsymbol{x}) \in \{0, 1\}$ for all $h_m \in \mathcal{H}$ and $\boldsymbol{x}$, and thus the above average is the percentage of 1's. The Figure 5 depicts the basic prediction procedure.

Generally speaking, bagging is particularly useful for datasets with high variance and potentially many noisy observations. More precisely, it is specially effective when the dataset contains instances where two observations with similar attributes have different responses. This kind of variability, often due to measurement error, inherent randomness in economic behavior, or unobserved confounding factors, can lead to over-fitting in single models. Bagging helps by averaging multiple models trained on different bootstrap samples, reducing sensitivity to such noisy fluctuations and leading to more stable predictions.

A very simple example helps on understanding how -in what sense- a Random Forest outperforms a single and eventually deep tree. Suppose, just to ease the exposition, the response variable is categorical, either 1 or 0. Assume further that for an specific vector of attributes, say $\boldsymbol{x}$, both responses are equally likely, that is:

$$\Pr(y = 1 \mid \boldsymbol{x}) = \Pr(y = 0 \mid \boldsymbol{x}) = \frac{1}{2} \tag{3}$$

Notice that we are not claiming that equation (3) is the *true* model that has generated our sample. It might be the case that, if we could observe attributes other than $\boldsymbol{x}$, the above conditional probabilities could be further structured. Thus, (3) simply reflects the maximum information that be elicited from the available sample. Now, in line with (3), assume we have in our sample a set of τ observations $\Omega := \{(\boldsymbol{x}, y_1), \ldots, (\boldsymbol{x}, y_\tau)\}$, that is, all observations in Ω have the same vector of attributes, but, in line with (3), different responses.

Consider first a single decision tree. Since all observations in Ω have the same attributes, any split based on those attributes will keep all of the observations in Ω together in the same branch. Suppose that, in fact, the decision tree comes to isolate Ω in a single node, say V. Then we apply some purity measure on V and conclude that we cannot accept that level of intra-node heterogeneity. What is then left to do? The only possible procedure is to further consider the observation index itself as an additional attribute -of course, artificial- so that the model can keep discriminating from V among observations in Ω. Clearly, if we allow enough further splits from V the model will perfectly predict and will have no intra-node heterogeneity. But, again from (3), all splits down from V are absolutely useless outside the training sample. That is over-fitting.

What does Random Forest do instead? Very simple. If observations in Ω are evenly distributed among the different bootstrap samples, roughly half of the trees will learn $\boldsymbol{x}$ leads to $y = 1$ and the other half will learn the complementary outcome. Thus, aggregating by average, that is, the percentage of ones, the Random Forest will learn (3), which is, as said, the maximum amount of information that can be extracted from the available sample.

Application of Random Forest to economics include: macroeconomics, see [12], financial forecasting, [109], agriculture [51], urban economics, [2], causal inference, [106], credit risk, [44] and [107], and consumer demand analysis, [75] and [63]. The list of pioneering papers on the methods, Random Forest and bagging, includes: [4], [13], [15] and [33]. The works by [8], [14], [33] and [36] focus on ensemble methods in which subsamples are, somehow, randomized.

While Random Forest is the most well-known application of bagging, bagging itself is a general ensemble technique that can be applied to other models beyond decision trees. Its ability to reduce variance makes it a valuable tool in economic and financial modeling.

5 Boosting methods

The term *Boosting* refers to a family of ensemble methods that, as Random Forests, combine simple models in order to generate a larger -better- model. Within the boosting methods, each simple model is usually termed as *weak learner*, whereas the aggregation is the *strong learner*. While in Random Forest the weak learners are trained in parallel, in boosting methods weak learners are added and trained sequentially, such that each new weak learner is selected in a way that improves upon -*boosts*- the performance of the set of the already existing ones. Additionally, while in Random Forest each weak learner is trained on a bootstrapped subsample, in boosting methods each weak learner is trained on the whole training sample, though, as said, the focus of each weak learner

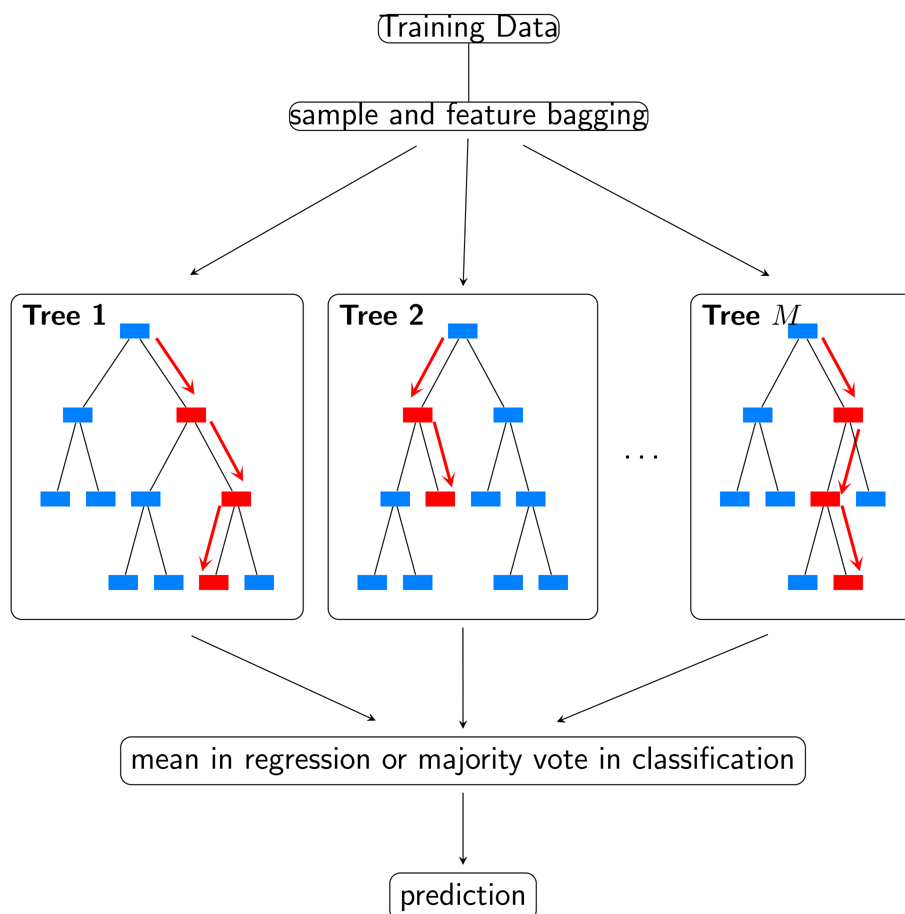


Figure 5: Random Forest model. Once the the model is trained, a given observation goes through different branches along different trees, represented by the red edges in each tree. In fact, the figure illustrates trees may have different morphology from one another. The prediction from each single tree is determined by the terminal node in which the observation is placed (nodes having no children). The prediction from the whole model is obtained by aggregating those individual predictions.

is on compensating the previous learners errors. [41] remains a classical reference for boosting methods. The next three subsections present three methods, roughly, ordered in increasing level of sophistication.

5.1 AdaBoost: Adaptive Boosting

This subsection presents the *AdaBoost*, which stands for Adaptive Boosting. In AdaBoost, each weak learner is typically a *stump*, which is a tree consisting of a root node with two children, both being terminal nodes. In other words, a stump has a unique split or, equivalently, it partitions the sample on the basis of a single feature. The essence of the method is simple. We start with all of the observations having the same weight, and

the first stump is trained on that sample. Then the weighted prediction errors from that stump are computed, and the weight for each observation is recomputed in a way such that observations with incorrectly predicted response receive a higher relative weight. Once new weights are available, a new stump is trained on the basis of those new weights. Again, weighted prediction errors are computed, new weights are computed on the basis of the new errors, and so forth.[11]

The mathematical description of the algorithm that adds new weak learners in AdaBoost requires some notation. Let $\Lambda := \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_N, y_N)\}$ denote the training sample, where, as before, $\boldsymbol{x}_i$ and y_i are the attributes and response of the i-th observation, respectively. While there are extensions that cover different types of response variables, here we restrict to a binary response. The possible responses are[12] -1 and 1. Finally, let us denote by t to a discrete boosting round, such that at every t a new weak learner is added, starting at $t = 1$, then $t = 2$, and so forth. Using this notation, the basic upgrading procedure is as follows. At $t = 1$, all observations in Λ are assigned equal weights, that is, $w_i^{(1)} = \frac{1}{N}$ for all $i \in \{1, \ldots, N\}$. Within each boosting round t, the procedure is described in Box 5.1.

Box 1: AdaBoost boosting round.

Step 1. Train a weak classifier h_t to minimize the weighted classification error, defined as:

$$\epsilon_t = \sum_{i=1}^{N} w_i^{(t)} \mathbb{I}(h_t(\boldsymbol{x}_i) \neq y_i),$$

where $\mathbb{I}$ is an indicator function that equals 1 if the prediction is incorrect, that is, $h(\boldsymbol{x}_i) \neq y_i$, and it equals 0 otherwise.

Step 2. h_t's contribution to the final model is given by:

$$\alpha_t = \frac{1}{2} \ln \left(\frac{1 - \epsilon_t}{\epsilon_t} \right).$$

Step 3. Update weights such that misclassified samples get higher weights to focus on harder-to-classify instances:

$$w_i^{(t+1)} = w_i^{(t)} \cdot e^{\alpha_t \mathbb{I}(h_t(\boldsymbol{x}_i) \neq y_i)} D^{(t+1)}$$

where $D^{(t+1)}$ is a normalizing factor.

[11] More precisely, each weak learner tries to *improve* on the errors made by his immediate predecessor, rather than by *all* of the predecessors. The underlying logic is that each learner already contains enough information on what his predecessors have failed.

[12] While not essential, this is the usual coding when describing the AdaBoost algorithm. Other than simplifying the maths, this notation emphasizes the core idea that errors predicting both classes are equally important.

After T iterations, the final prediction is made using a weighted majority vote:

$$H(\boldsymbol{x}) = \sum_{t=1}^{T} \alpha_t h_t(\boldsymbol{x}) \tag{4}$$

Consider the example in Figure 3. In panel (A) all of the weights are the same (indicated by the size of the + and −) and the model makes a first decision rule, given by the vertical line at $D1$. A stump is just either a vertical or an horizontal line in the sample region. Whatever the model correctly classifies is given less weighting in the next iteration. The model incorrectly partitioned 3 of the + observations and correctly classified all other points. Since the model incorrectly classified these 3 observations, their weights are increased, which we represent by an increase in the size of those points in panel (B). Consequently, the correctly classified points obtain less weight, represented by a size reduction in panel (B). The model makes a new classification depicted at decision rule $D2$. The model correctly classified the previous incorrectly classified + observations and their weights are subsequently decreased (depicted in panel (C)). The two left-most +'s and right-most −'s are again correctly classified and therefore their weights are further reduced in panel (C), given by a smaller + and − signs. Finally, in panel (C) the final classification is given by the horizontal line $D3$ and the weights are recomputed and updated. The final classification model is given in panel (D) in which all points have been partitioned correctly and this model is a stronger classifier than any individual model previous to it. That is, the model used a combination of linear classifiers in order to build a stronger non-linear classifier based on weighted voting.

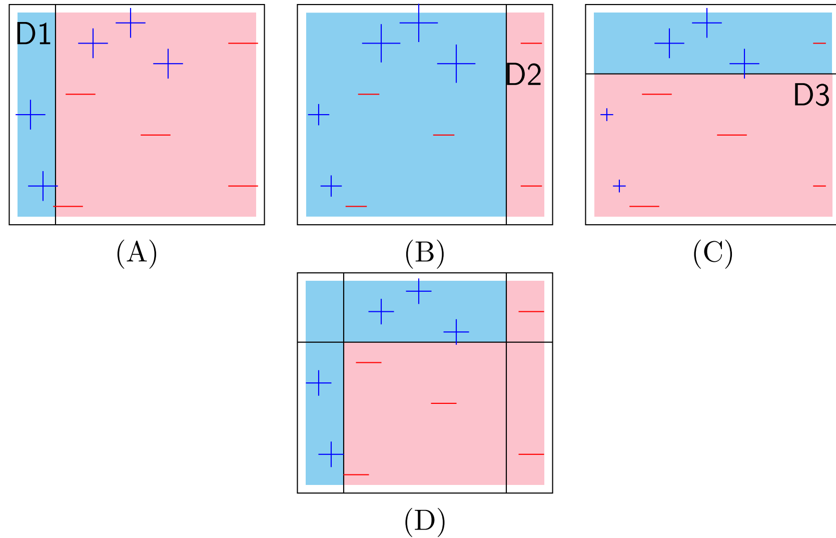


Table 3: AdaBoost classification weights illustration

A few final remarks are important. First, the algorithm automatically shifts attention to difficult-to-classify samples. Second, even if individual classifiers are slightly better

than random guessing (which corresponds to $\epsilon_t = 0.5$ in the previous box), AdaBoost can build a strong classifier. Third, since misclassified points get higher weights, outliers can have a large influence.

The list of references using Adaboost in economics is very extensive. It includes: labor economics [1], market design [40], housing markets [46], energy markets [59], credit scoring [62], health economics [78], digital economy [80], fraud detection [99] and demand forecasting [112].

5.2 Gradient Boosting Machines (GBM)

This subsection adds a new element to the boosting methods family: gradient based optimization. As said, in a boosting method the weak learners are added sequentially, each one trying to improve the performance of the already existing ones. All gradient boosting methods are primarily characterized by some loss function to be minimized. Given some existing set of weak learners, not only the current loss, but the direction of the steepest descent -maximal reduction- of loss is computed, and the new learner is fitted to that direction. In other words, each new weak learner classifies as best as possible the residuals of the previous learners. Typically, weak learners are trees, and here comes another major difference between AdaBoost and gradient based boosting. In AdaBoost the structure of the tree to be added is fixed, typically a stump. In contrast, in gradient based boosting, while limited by metaparameters, like the maximum depth of the tree or the minimum amount of samples each leaf must have, the structure of the tree is a product of the optimization process.

Other than differences in the criterion to add new weak learners, there is also a difference between AdaBoost and gradient boosting methods on how to aggregate across weak learners predictions. Let rounds be indexed by t, with $t \in \{0, 1, \ldots, T\}$, as previously. Let h_t be the weak learner added at round t and let $h_t(\boldsymbol{x})$ be its prediction for the instance -observation- $\boldsymbol{x}$. Now, let F_{t-1} be the aggregate of all weak learners prior to h_t and, analogously, let $F_{t-1}(\boldsymbol{x})$ be the prediction of F_{t-1} for $\boldsymbol{x}$. Then, for a continuous response variable, it is:

$$F_t(\boldsymbol{x}) = F_{t-1}(\boldsymbol{x}) + \eta h_t(\boldsymbol{x}) \tag{5}$$

where $\eta \in (0, 1)$ is a hyperparameter named the *learning rate* or *shrinkage*. Essentially, η scales the contribution of each weak learner. If we substitute recursively F's in (5) we might conclude that learners added at early rounds have a larger weight than those added at latter rounds. However, we must be cautious to not interpret weights as we did in AdaBoost. Roughly speaking, in AdaBoost each weak learner faces the whole training sample, and then learners receive weights depending on their relative performance. In contrast, in gradient boosting each weak learner faces -is constructed to attack- *just* the the errors that remain to be fixed when he is created.[13]

[13]The extension of (5) to a binary response variable is done using the *logit* function, which is the log-odds. Consider a response with two possible classes, either positive or negative, and let p denote the

Boosting methods differ from one another in a number of elements. Let us just list three relevant ones. First, methods differ on the loss function under consideration. For instance, mean squared error is typically used for regression while log loss is used for classification. Second, different methods might include different *regularization* techniques, intended to reduce over-fitting. Third, methods might also differ on how trees are grown, level-wise *vs* leaf-wise.

Because of its gradient-based upgrading procedure, boosting methods are exposed to well-known problems that are common to all numerical gradient-based optimization techniques: non-convexities created by either an strict local minimum, a saddle-point, or a flat region, called a *a plateau*. How to get out of the *trap* mostly goes down to how to recognize that you are in it. This is a very old question in the optimization literature, and an exhaustive treatment lies beyond the scope of this paper. As ML concerns, some standard tricks are in the regularization term, to be further explained in the next subsection for XGBoost, or an appropriate choice of the learning rate, or introducing some randomness along the optimization procedure. The learning rate, η in equation (5), can be roughly understood as a step size. A large step size might keep the algorithm jumping indefinitely around the minimum. In contrast, a too small step size might make the algorithm goes too slow in regions in which the steepest descent direction is *very clear*. Introducing randomness in the optimization procedure consists basically on trying some *random guess*, that is, not suggested by the gradient, as a sort of robustness check. This is named *stochastic gradient boosting*.

The three most widely used gradient boosting methods are XGBoost, LightGBM and CatBoost. We treat XGBoost in the next subsection. Roughly, while XGBoost is very robust, LightXBM is designed to perform faster, which is particularly useful for large datasets. CatBoost is designed for categorical responses.[14] A pioneering work on gradient boosting machines is [42]. A remarkably rich theory has evolved around boosting, with connections to a wide range of topics including statistics, game theory, convex optimization and information geometry. Some foundational works are [56], [83], [85], [84] and [100]. [35] applies boosting methods to predict economic recessions.

probability of positive, the logit function is $l(p) = ln\frac{p}{1-p}$. The prediction is taken to be on $l(p)$, so we are back to predict a continuous variable.

[14]Since gradient boosting methods rely on gradient computations, they generally do not natively support categorical variables, whether as attributes or as the response. If categorical variables are present in the dataset, we must either transform them into numerical form before applying gradient boosting or use CatBoost, which handles them directly. Some further comments on categorical variables will be done for Artificial Neural Networks, which suffer the same problem than boosting methods on this regard.

5.3 Extreme Gradient Boosting (XGBoost)

The subsection presents *Extreme Gradient Boosting* (XGBoost), [25], which is likely the most widely used gradient boosting method. There are several differences between XGBoost and other gradient boosting methods. First, the loss function adds two regularization terms, as we explain below. Second, the loss minimization algorithm considers not only the gradient but also hessian. Third, it includes a sophisticated post-pruning method. It all delivers a speed and accuracy that frequently outperforms other gradient based methods.

A detailed breakdown of the loss function used by XGBoost will help in understanding all of the effects taken into account. Prior to that, we need to introduce some notation. First, it is standard to denote by θ to the whole model, that is, the collection of trees, the structure of each tree, and the order in which they have been added to the model.[15] In addition, within each tree, each of the terminal nodes, or leaf, is given a *weight*, which is the prediction for instances that fall down into that leaf. We denote those weights by w. For example, say that a tree h for the instance $\boldsymbol{x}$ has a prediction $h(\boldsymbol{x}) = 5$. That is equivalent to say that in h the instance $\boldsymbol{x}$ falls in a leaf whose weight is 5. Finally, let $\boldsymbol{w}$ denote the whole set of weights. If, say, the model consists of just two trees, with weights $[3, 5, 2]$ and $[4, 2]$, respectively, then $\boldsymbol{w} = [3, 5, 2, 4, 2]$. Since it has a vector-like structure, the standard L_1 and L_2 norms apply, denoted $||\boldsymbol{w}||_1$ and $||\boldsymbol{w}||_2$, respectively.[16] The loss to be minimized is:

$$\text{Obj}(\theta) = \sum_i L(y_i, \hat{y}_i) + \gamma T + \alpha ||\boldsymbol{w}||_1 + \lambda ||\boldsymbol{w}||_2^2 \tag{6}$$

The first term in the right hand side of (6) is the loss due to prediction errors. The sum runs over instances of the training sample. For a continuous response variable, the usual metrics for the prediction errors is the mean squared error: $L(y_i, \hat{y}_i) = (y_i - \hat{y}_i)^2$, where $\hat{y}_i$ is the predicted response for the observation $(\boldsymbol{x}_i, y_i)$. For a binary response variable, say with classes 0 and 1, the method normally takes a logistic, binary cross-entropy, loss: $L(y_i, \hat{y}_i) = y_i ln\hat{p}_i + (1 - y_i) ln(1 - \hat{p}_i)$, where $y_i \in \{0, 1\}$ whereas $\hat{p}_i$ is the predicted probability of the event $y_i = 1$.

The last three terms in (6) conform a regularization term, usually jointly denoted as $\Omega(\theta)$, where the notation emphasizes its dependence on the tree structure but not on

[15]The order matters. Consider two different models, say $\mathcal{H}_A$ and $\mathcal{H}_B$, represented by θ_A and θ_B, respectively. Suppose $\mathcal{H}_A$ added first tree h_1 and then tree h_2, whereas $\mathcal{H}_B$ added first h_2, then h_1, with $\mathcal{H}_A$ and $\mathcal{H}_B$ being identical one another thereafter. Then $\theta_A \neq \theta_B$. Also the structure matters. Suppose that $\mathcal{H}_A$ and $\mathcal{H}_B$ only differ in the tree added in the t-th round, in which $\mathcal{H}_A$ added h_A and $\mathcal{H}_B$ added h_B, where h_A and h_B have the same terminal nodes but different splits to go from the root to the terminal nodes. Then again $\theta_A \neq \theta_B$ holds.

[16]For the example: $||\boldsymbol{w}||_1 = |3| + |5| + |2| + |4| + |2|$ and $||\boldsymbol{w}||_2 = \sqrt{3^2 + 5^2 + 2^2 + 4^2 + 2^2}$. In (6) we have the square of $||\boldsymbol{w}||_2$.

the instances. So, we equivalently can write (6) as:

$$\text{Obj}(\theta) = \sum_i L(y_i, \hat{y}_i) + \Omega(\theta)$$

where, obviously, $\Omega(\theta) := \gamma T + \alpha||\boldsymbol{w}||_1 + \lambda||\boldsymbol{w}||_2^2$. Overall, the regularization term penalizes *too complex* structures. First, T is the total number of leafs. For instance, if the model has just two trees with three and two leafs, respectively, as in the previous example, then $T = 5$.

The L_1 and L_2 norms are as defined before, and aim for *sparsity* and *inter*-tree balance, respectively. Let us consider first L_1. If some leafs have a small weight in absolute value, the L_1 term tends to push them towards zero. It essentially induces global sparsity, since it detect features that are mostly irrelevant for prediction, that is, features leading to leafs with small weights. Indirectly, we might say that seeks for *intra*-tree balance. In contrast, L_2 looks for inter-tree balance. In terms of L_1, $\boldsymbol{w} = [10, 0]$ and $\boldsymbol{w'} = [5, 5]$ are equally good. In terms of L_2, $\boldsymbol{w'}$ is better off. Thus, L_2 penalizes models in which one leaf of a given tree has much more importance than all of the others. This helps in preventing over-fitting. Finally, γ, α and λ are the hyperparameters that establish the relative impact of each regularization term with respect to prediction errors.

As explained, the loss function depends on -must be derived with respect to- θ, which fully describes the model. Notice that θ is not a usual finite dimensional real vector, but a function. Thus, derivatives must be taken in a space of functions, which is a *functional* derivative, in particular the *Fréchet* derivative. However, the algorithm is based on an approximation to that derivative using derivatives in an euclidean space.

As a graphical summary of the previous concepts, consider Figure 6 and how changes in the regularization term controls the complexity of the model and its response to over-fitting. The objective is to fit a step function given in the upper left quadrant. We want a simple and predictive model. The figure in the upper right quadrant is too complex and over-fits the data, the figure in the lower left quadrant is simple but not very predictive. Therefore, the most simple and most predictive model is in the lower right quadrant. As mentioned in a previous section, the trade-off between a simple and predictive model is also referred to as the bias-variance trade-off: the upper right quadrant has a high variance (over-fits) whereas the lower left one has a higher bias (under-fits).

A merely illustrative brief list on the wide range of economic applications of XGBoost is: bankruptcy prediction [10] and [88], marketing [24], economic stability [43], price forecasting [49], energy prices [96] and bond default risk [110].

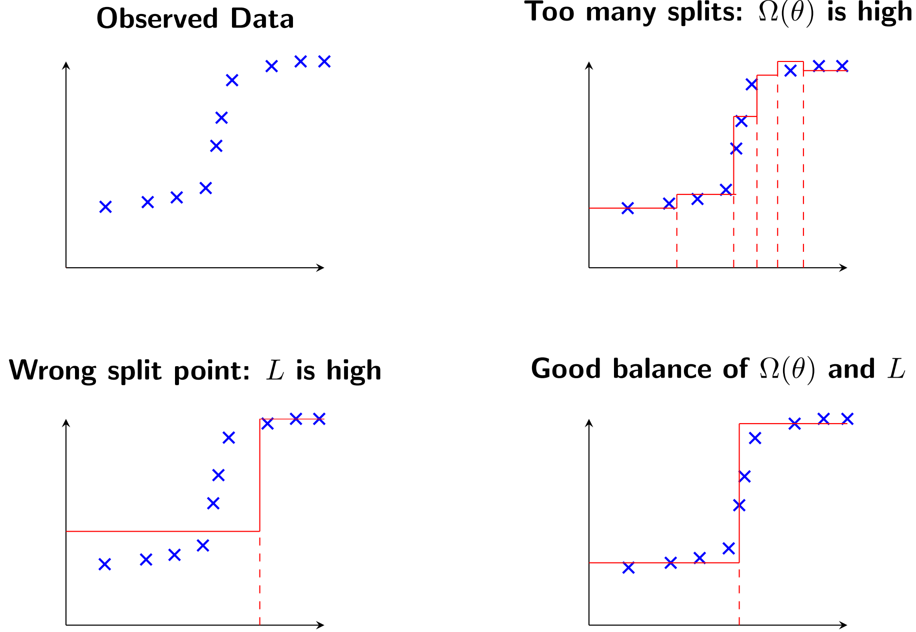


Figure 6: XGBoost Regularisation

6 Artificial Neural Networks (ANN)

In the previous sections we have presented tree based methods, either considering trees in isolation or an ensemble of trees. When ensemble, the procedure to add new trees is fully data-driven. *Artificial Neural Networks* (ANN), or simply neural network, have a fundamentally different approach, to certain extent aiming to replicate the functioning of a biological brain. First, the unit of learning is a *neuron*, which receives, processes and sends information. Second, in ANN, as it happens in a brain, the architecture by which each neuron is connected with the others is fixed *ex-ante*, that is, it is not data-driven. That architecture consists of a hierarchical organization in *layers*. Not all of the neurons are *in touch* with the data. Instead, only the neurons in the so-called *input* layer are.

Mathematically, a neuron is a function which receives N inputs, say $\boldsymbol{x} = (x_1, \ldots, x_N)$, which can be either features of a dataset or outputs from *previous* neurons in the network. Each of the input is given a weight. Let us denote the vector of weights as $\boldsymbol{w} = (w_1, \ldots w_N)$. The neuron internally computes the weighted sum of inputs and adds to the sum a *bias*, denoted as b. It then applies an *activation* function, denoted as σ, over the weighted sum to generate the output, denoted as a. In just one expression, it is:

$$a := \sigma\Big(\sum_n w_n x_n + b\Big) \tag{7}$$

The name of the activation function relates to the idea that it determines how much the *next* neuron is *activated*. The bias is not strictly an input but an *internal* value that allows the next neuron to be activated even though the current neuron receives all inputs equal to zero. The activation function is defined *ex ante*. The usual activation functions are: sigmoid, hyperbolic tangent (*tanh*), rectified linear unit (ReLU) and leaky ReLU.[17] The flow of incoming and outgoing information is depicted in Figure 7.

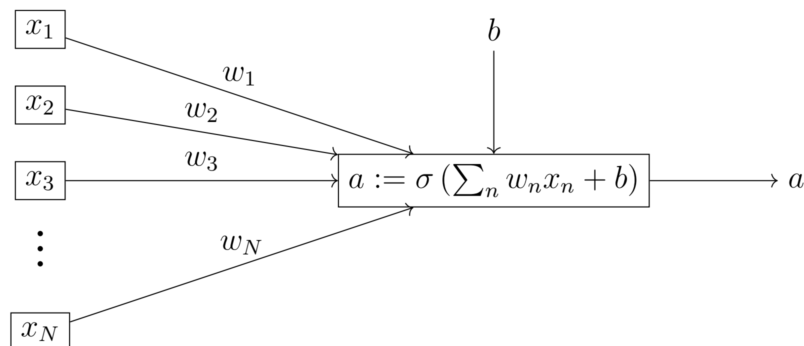


Figure 7: Neural Network Neuron example

Notice that in the description of what a neuron is we have implicitly made use of the architecture of the network, as we have used the terms *previous* and *next* neurons. Let us clarify these terms. The Figure 8 presents two examples of *feedforward* neural networks. Essentially, the information flows from left to right in the figure. Within each panel, the starting nodes are the *input* layer, which are the sample features, the final point is the *output* layer, which is the prediction the network computes for each instance. For example, suppose we have a dataset in which each observation gathers characteristics of a house and its price as a response variable. More specifically, the characteristics we observe are just the size and the number of bedrooms. For an instance in which the size is $100m^2$ and the number of bedrooms is 3, we will have 100 placed in x_1 and 3 in x_2, and no more components in the input layer. In addition, the figure represents the bias as x_0, in a different color to emphasize that it is an input for the neurons but it is not a part of the sample. All layers other than the input and the output are *hidden*. The figure shows that neurons are grouped on hidden layers. All connections are unidirectional, and there are connections only between neurons of adjacent layers. In the usual terminology, *shallow* and *deep* neural networks refers to networks with just one or more than one hidden layer, respectively.[18]

The hyperparameters of a neural network include its structure: how many layers and

[17]The sigmoid is $\sigma(z) = (1 + e^{-z})^{-1}$. Furthermore, $tanh(z) = (e^z - e^{-z})(e^z + e^{-z})^{-1}$, ReLU is $\sigma(z) = \max\{0, z\}$ and the Leaky ReLU is $\sigma(z) = \max\{\alpha z, z\}$, where α is some small positive constant.

[18]In this paper we describe the basic architecture of a *forward* Neural Network. There are plenty of variations that consider more complex dependence flow between layers.

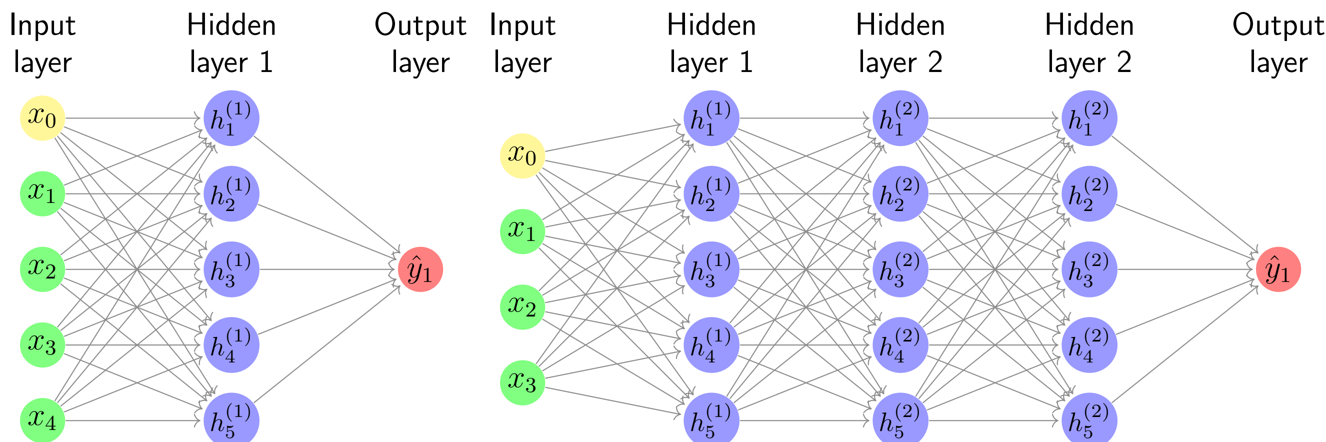


Figure 8: Shallow (left) and Deep (right) Neural Network example. In both panels x_0 represents the bias, which is an input not from the sample.

how many neurons per layer. In contrast, the weights and the bias are learned by the network. Roughly, the neural network learns on those parameters by minimizing a loss function which account for prediction errors. The learning process alternates between a *forward* and a *backward pass*, the latter commonly referred to as *backpropagation*. For a given set of weights and biases and a specific input instance, the forward pass generates a prediction. The prediction error is then computed. The backpropagation consists of computing the gradient of the loss function with respect to the weights and biases. Then the weights and biases are updated: each weight is adjusted by subtracting the learning rate (a hyperparameter, η as in the boosting methods) multiplied by the corresponding partial derivative. This forward and backpropagation process is repeated for each training instance, with the weights and biases being updated sequentially for each new instance, or *batch* of instances. A complete cycle through the entire training set is called an *epoch*. The number of epochs is another hyperparameter of the model. See Section C in the Appendix for a detailed example on how weights and biases are updated.

A key question in neural networks is that, as shown, arithmetic operations are performed with the attributes, so we need to feed numerical values, however, there are plenty of economic applications in which data on attributes are primarily categorical. Two clear examples are education level or place of birth as attributes to predict wages. Traditional econometric techniques have developed several methods. When the attribute has a low number of classes, as for education level, we might create a vector of dummies, such that for each instance the vector has 1 in the position for the corresponding category and 0 elsewhere. This is named *one-hot encoding*. A clear problem is the loss of *neighbourhood*: it might be the case that category A is naturally closer to B than to C, but this encoding fully ignores it. Alternatively, we might simply assign 1 to A, 2 to B and 3 to C, which keeps some idea of neighborhood, but it induces an artificial cardinality. When there many categories, as potentially occurs with the place of birth,

the one-hot encoding induces a high dimensionality of the vector of attributes. Relatively recently, spatial econometrics builds on the concept of spatial correlation on a distance matrix based on geolocation. The approach is not free from subtleties as long as neighborhood between attributes is not systematically mapped into similar responses.

Neural networks offer an alternative approach, which is using *embedding layers*. The core idea is simple: categorical values are embedded into vectors, whose dimension is an hyperparameter. Unlike the encoding approaches outlined above, where the vector of values assigned to each category is fixed by the practitioner, embedding layers allow the network to learn these vectors. The network learns to do assignments that preserve some neighborhood, but at the same time might learn that neighbor attribute values do not have similar responses. The embedding layer approach is particularly powerful when dealing with a large number of categories, as it avoids the curse of dimensionality associated with one-hot encoding.

The foundation of artificial neural networks traces back to [66], who set the concept of artificial neuron as a form to model logical functions, and [81], whose *perceptron* was the first neural network model. The literature has largely developed since those pioneering works in multiple directions, with periods of growth and stagnation. Some papers that have been critical on the theoretical development are [11], [47], [48], [60] and [82]. Regarding the list of economic applications, the topics are similar to the cited for the tree based models. Demand forecasting [3], energy [9], inflation forecasting [26] and [73], financial crisis [28], renewable energy [54], energy demand [71], stock prices [79], macroeconomic forecasting [87], GDP growth [97] and oil price forecasting [108] constitute illustrative examples.

We outline the basic ingredients of three additional methods in Section D, in the Appendix. First, we present LASSO and a family of related regressions, whose common element is that the loss function weights prediction errors against some regularization term. Second, we briefly present the basics of Support Vector Machines, which essentially deals with the characterization of a minimal set of hyperplanes that separate the sample attributes according to values of the response variable. The basic idea is -to some extent- close to the visually presented in the panel on the right in Figure 1, but an hyperplane is a more general and flexible concept than the splits in that figure. Third, we present Shapley values, a concept taken from cooperative game theory, which measures the *value*, in terms of prediction, each feature adds to each possible *coalition* of features.

7 A case study with XGBoost

This section walks through Python code and explains step-by-step how to set up a classification model using *Extreme Gradient Boosting* (XGBoost). In order to ease the reading, the Python code is relegated to Section E, in the Appendix.

7.1 Generating synthetic data

In Listing 2 we first import the required packages and then generate some synthetic classification data with 10 variables and 1000 observations. The function that allows for synthetic generation of a categorical response is `make_classification`. In our view, this function serves as an important testing ground for the practitioner to learn how the method performs in a variety of scenarios.[19]

In order to give an idea on how the training and testing data may look like Figure 9 shows the training and test scatter plot. Table 4 shows the first 5 observations along with the *target* variable we aim to train the model to predict, the inputs or variables used in training are labeled from 0 to 9 (10 variables), these would be standard economic variables in other datasets. The code to generate this data, split the data between training and testing and set up the plotting configuration can be found in Listing 2. The general convention is to follow standard econometric literature where `X` denotes the dependent variables and `y` denotes the independent variable we want to predict.

Table 4: First 5 observations of synthetic data

	0	1	2	3	4	5	6	7	8	9	target
0	-1.010	0.778	-1.555	0.003	-0.720	-0.276	0.248	-0.648	-1.070	-0.017	0
1	0.027	0.219	1.461	-1.100	1.443	0.469	-0.854	0.140	0.346	0.385	1
2	1.498	0.591	0.252	0.743	-0.756	-0.439	0.994	-0.092	0.165	1.156	0
3	-0.110	-0.038	-0.025	-1.055	1.496	-0.792	-0.431	0.668	0.072	-0.268	1
4	2.269	-0.589	0.175	-1.815	1.478	-1.438	-0.399	0.767	-0.252	0.430	1

7.2 Hyperparameters

As explained, XGBoost model requires some hyperparameters. The values must be set by the practitioner or optimized through a grid search, which increases computational time exponentially. A comprehensive overview of each hyperparameter is not discussed here, though some of them have been already explained in a previous section.[20]. In particular, `eta` helps to reduce over-fitting by shrinking variable weights, `gamma` sets the minimum loss reduction required to make further partitions, `max_depth` helps to control for the tree complexity, large values will lead to over-fitting trees, `subsample`

[19] Some of the arguments allowed are `n_features`, `n_informative`, `n_redundant`, which control the number of features, which of them are informative and which redundant, respectively. In our example, there are ten features, among which only two are relevant. In addition, `n_clusters_per_class` controls, roughly, how many regions in the space of attributes contain instances of the same class. The easiest scenario for the method is when just one region contains all of the instances of a given class. The arguments `flip_y` and `class_sep` also deal with separability between classes. `make_classification` also supports multi-class responses, whereas `make_regression` generates a continuous response.

[20] A full list can be found at https://xgboost.readthedocs.io/en/stable/parameter.html

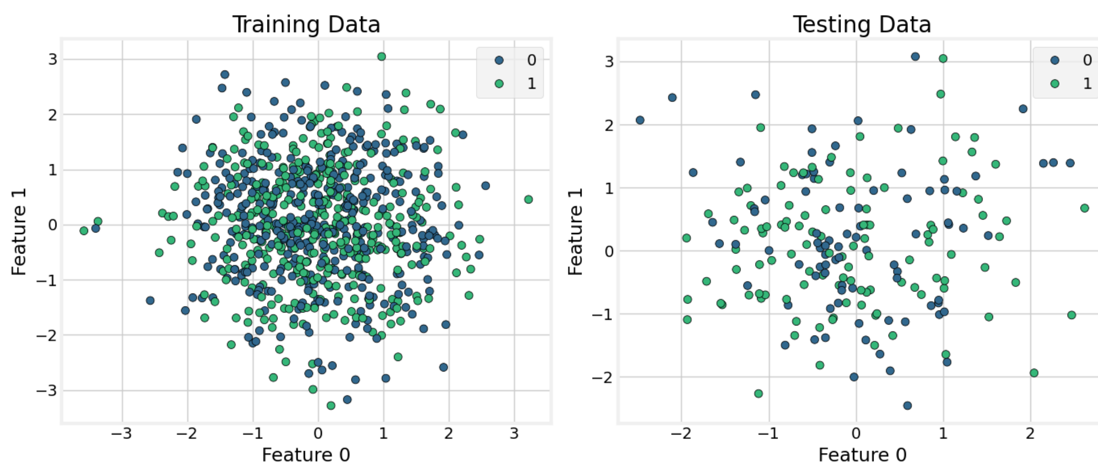


Figure 9: Training and Testing Synthetic Data

sets the percentage of the training data used to build a tree, i.e. a value of 0.6 will use 60% of the training data to build a tree, the data is then re-sampled and 60% is taken to build a new tree, `colsample_bytree` sets the fraction of columns (or variables) used to grow a tree.

There are many hyperparameters required for an XGBoost model and it is difficult to decide which ones should be optimised prior to training the model. This is where re-sampling and cross-validation comes in useful. The code in Listing 3 applies cross-validation and a grid-search for hyperparameters within a pre-defined grid. It selects the best model performance based on some metric such as the model's accuracy or RMSE.[21]

7.3 Applying the model

Now that we have obtained optimal parameters from a grid-search we can train our final model and apply it to our held-out test dataset and from here we can run model statistics such as RMSE, MAPE, MAE for regression type problems or construct the confusion matrix and the associated metrics for classification type problems.[22]

An important graphic we can construct from the model is the feature importance plot. It gives an estimate of how much each feature contributes to the overall predictive performance of the mode. In other words, it measures which feature is most influential in the model. The importance can be computed in several different ways. The default

[21]The grid-search is just one method of searching optimal hyperparameters, we could have used `RandomizedSearch` or Bayesian Optimisation `BayesSearchCV`.

[22]The code for ROC, PR and confusion matrix would mimic the presented in Section B in the Appendix, and thus we omit it here.

method is the gain and it measures the model's accuracy, reduction in the loss function, when a given feature is used for splitting. Higher *gain* values indicate that the feature is more importance for making predictions. Additionally, feature importance can be computed by *weight* or the number of times a feature is used in all the trees, if a feature appears more frequently, then it is considered more important.

The code to plot the feature importance plot can be found in Listing 4. Figure 10 shows the feature importance from the best XGBoost model after cross validation. Note that the model is able two identify the two features that actually determine the response variable, which is in line with the specification when generating the data with `make_classification`.

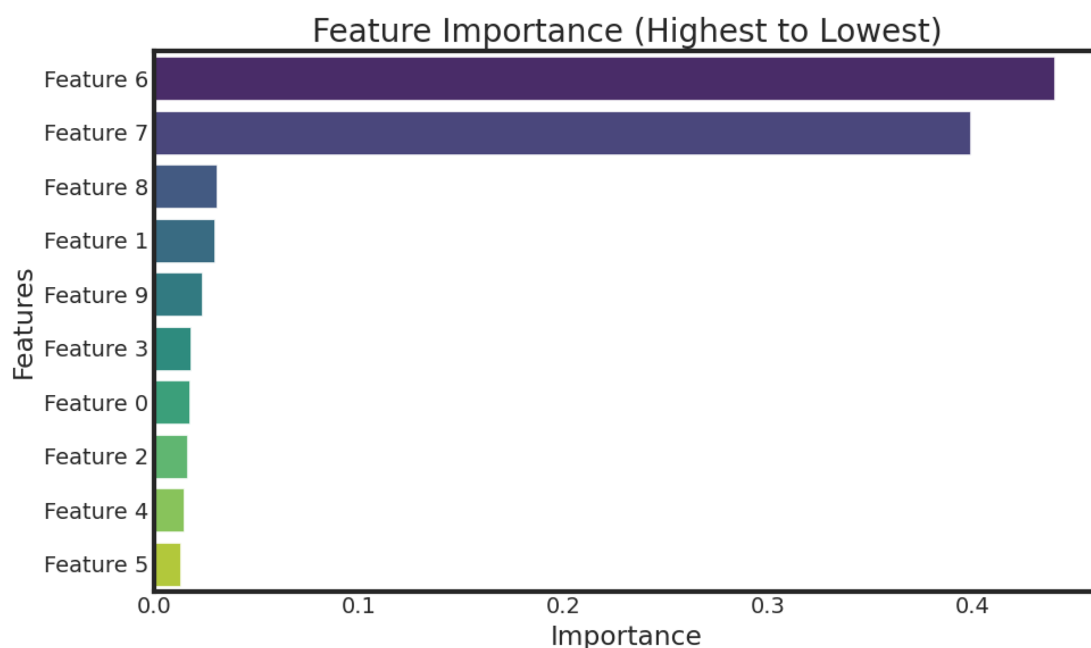


Figure 10: Feature Importance

The Figures 11 to 14 show different forms to compute importance of a variable using the Shapley value, which essentially averages its marginal contribution across all possible coalitions of features. The Python code is in Listing 5. As mentioned, in Section D, in the Appendix, we present the concept of Shapley value. Leaving for that section a more precise explanation on the concept, we see that both Figures 11 and 12 are in line with Figure 4 in the sense that all those figures point to the same features as the *important* in terms of prediction. The Figures 13 and 14 take a step further. Even if, say, feature 6 is important, perhaps its importance is not constant along its whole range of values in our sample. These latter figures show importance conditional on specific values for each of the relevant features.

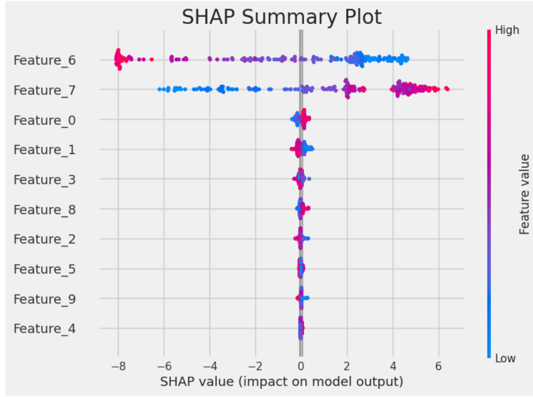


Figure 11: Shapley Summary Plot

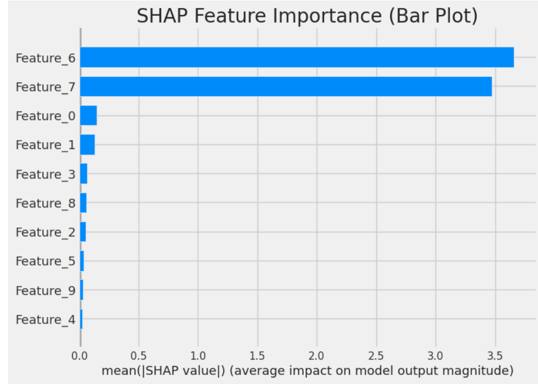


Figure 12: Shapley Global feature importance

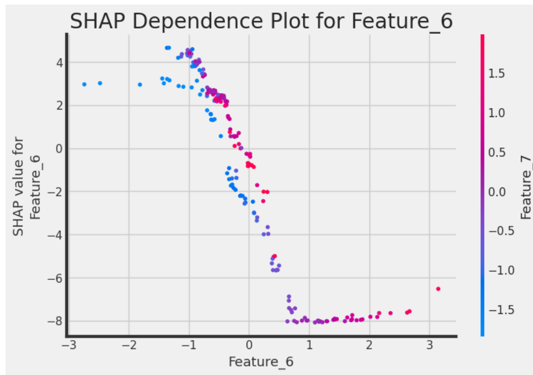


Figure 13: Shapley Conditional Feature 6 importance.

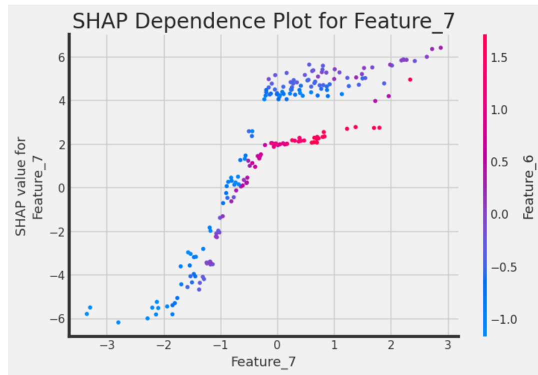


Figure 14: Shapley Conditional Feature 7 importance.

7.4 Some final remarks

This XGBoost case study aimed to show how one of the most popular ML models can be applied to a typically structured dataset often faced by economists - i.e. a tabular dataset with columns as variables and rows as observations. We generated a synthetic dataset for this illustration, which allows the practitioner to control the sample on which methods are applied.

We demonstrated how hyperparameter tuning can be applied in the form of a grid search and how cross-validation is applied in order to find the best model. Additionally, the out-of-sample error was computed using an accuracy score on the predictions compared with the actual observed results. Finally a number of metrics on feature importance have been presented.

Finally, in order to provide a more general view, the Listings 6 and 7 contain the

basic implementations for an ANN and a Random Forest, respectively, excluding hyperparameter optimization. In both cases, the listings assume there are already defined some train and test samples, specifically: `X_train`, `y_train`, `X_test` and `y_test`.

8 Conclusions

This paper has provided a practical and intuitive introduction to applying Machine Learning (ML) methods in economics, with a primary focus on the core ideas behind tree-based models and neural networks. Our objective has been to offer economists a clear understanding of how these models work, their theoretical underpinnings, and how they differ from, yet complement, traditional econometric approaches. Additionally, we have provided Python scripts to facilitate the implementation of these techniques in real-world applications.

Both ML and econometrics share a strong foundation in statistical theory but differ in their primary goals and methodologies. While econometrics emphasizes structural interpretation and causal inference through predefined functional forms, ML focuses on predictive accuracy and the flexible identification of patterns in data. This distinction makes ML particularly effective in high-dimensional settings and when relationships between variables are complex and non-linear.

A key takeaway from our analysis is that ML methods excel in prediction tasks, such as credit risk assessment, demand forecasting, and policy evaluation. Techniques like cross-validation and hyperparameter tuning further enhance the robustness of these predictions and help prevent overfitting. However, the absence of explicit parametric forms in ML models can limit their ability to directly provide causal insights, a central aim of many econometric analyses.

While we included a case study using XGBoost to illustrate the practical application of ML techniques, the main contribution of this paper lies in demystifying the core concepts behind popular ML methods and equipping practitioners with the tools to apply them effectively. This case study demonstrates how ML can be integrated into empirical economic analysis, though the insights extend well beyond a single application.

In conclusion, ML and econometrics should be seen as complementary rather than competing approaches. Each offers unique advantages depending on the research question and the nature of the data. By incorporating ML into their analytical toolkit, economists can expand their capacity to uncover complex relationships, improve prediction accuracy, and enhance empirical analysis in ways that traditional methods alone may not achieve.

APPENDIX

A Re-sampling and cross validation

Re-sampling is the process of sampling more than once from a dataset and then re-fitting a model using this newly sampled data. The most widely used are bootstrap and cross-validation. Since ML models require that the trade-off between bias and variance be satisfied such that the model does not over-fit the data, a common practice in supervised learning problems is to hold out part of the data as a *testing* dataset. The moment we apply our model evaluation on the test data is the moment we introduce look-ahead bias into our model since we might be tempted to adjust the parameters of the model to give more satisfactory test results.

Ideally, the approach would be to split the data into three randomly divided parts *training*, *validation* and *test* sets. The training set is used when fitting the model, the validation set is used when estimating the prediction error from the model selection. From these two splits of the data we estimate the performance for a number of different models in order to see which model performs the best. Finally, the test set is then used at the end to asses the generalization error of the chosen model - which was selected at the training and validation stage.

In practice there is usually not enough data in order to simply split between training, validation and testing datasets. In order to select the best model and search for the most optimal parameters we can randomly shuffle the data and then split the training data up into k equal sized validation parts or k-folds. Suppose we split the training and validation data into 10-folds as depicted in Figure 15. The model is trained on the section indicated by the white space and validated on the grey space in each partition.

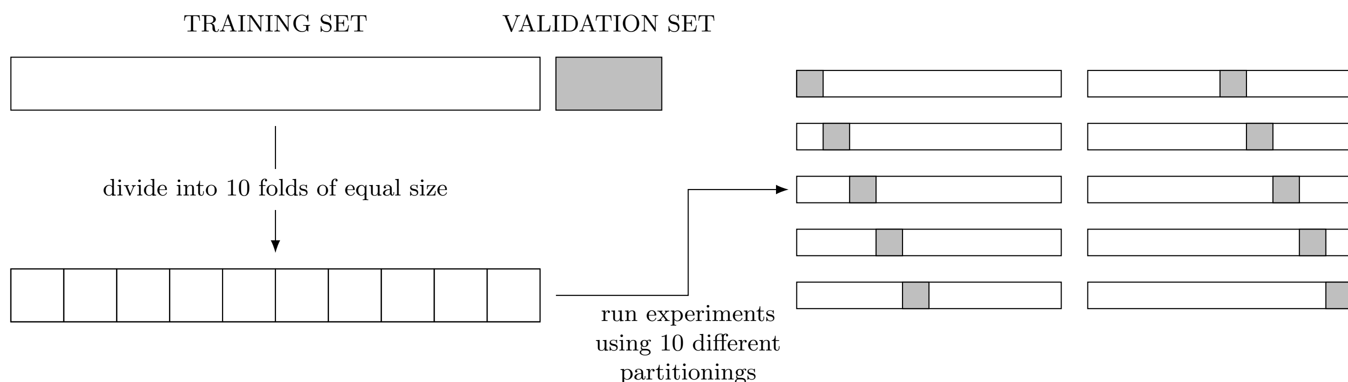


Figure 15: k-fold Cross Validation where $k = 10$

Cross validation allows us to obtain a robust performance estimation we can see the average performance over the different subsets of data which gives us more confidence when applying the model to the held-out test set. Additionally, we can see the variance of the model's performance across the different sub-samples. A low variance across the sub-samples indicates consistent performance whereas a high variance may signal sensitivity to the data samples. We can also identify if the model overfits on any of the sub-samples.

B Python code for ROC, PR and Confusion matrix

Listing 1: ROC, precision-recall and confusion matrix

```

import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, precision_recall_curve, auc,
    confusion_matrix, ConfusionMatrixDisplay

def plot_roc_curve(y_true, y_scores, filename="roc_curve.png"):
    """Plots and saves the ROC curve."""
    fpr, tpr, thresholds_roc = roc_curve(y_true, y_scores)
    roc_auc = auc(fpr, tpr)

    plt.figure(figsize=(8, 6))
    plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area =
        %0.2f)' % roc_auc)
    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver Operating Characteristic (ROC)')
    plt.legend(loc="lower right")
    plt.savefig(filename)
    plt.close()

def plot_pr_curve(y_true, y_scores, filename="pr_curve.png"):
    """Plots and saves the Precision-Recall curve."""
    precision, recall, thresholds_pr = precision_recall_curve(y_true, y_scores)
    pr_auc = auc(recall, precision)

    plt.figure(figsize=(8, 6))
    plt.plot(recall, precision, color='blue', lw=2, label='PR curve (area =
        %0.2f)' % pr_auc)
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('Recall')
```

```
    plt.ylabel('Precision')
    plt.title('Precision-Recall Curve')
    plt.legend(loc="lower left")
    plt.savefig(filename)
    plt.close()

def plot_confusion_matrix(y_true, y_scores, threshold=0.5,
    filename="confusion_matrix.png"):
    """Plots and saves the confusion matrix."""
    y_pred = (y_scores >= threshold).astype(int)
    cm = confusion_matrix(y_true, y_pred)
    disp = ConfusionMatrixDisplay(confusion_matrix=cm)

    plt.figure(figsize=(8, 6))
    disp.plot(ax=plt.gca())
    plt.title(f'Confusion Matrix (Threshold={threshold})')
    plt.savefig(filename)
    plt.close()

# Example usage:
y_true = np.array([0, 0, 1, 1, 0, 1, 0, 0, 1, 1])
y_scores = np.array([0.1, 0.4, 0.35, 0.8, 0.2, 0.9, 0.05, 0.5, 0.7, 0.6])

plot_roc_curve(y_true, y_scores, filename="my_roc.png")
plot_pr_curve(y_true, y_scores, filename="my_pr.png")
plot_confusion_matrix(y_true, y_scores, threshold=0.6,
    filename="my_confusion.png")
```

C Forward and Backpropagation in ANN's

This section presents the updating procedure for a given ANN and a given instance. As mentioned in the main text, the updating procedure depicted here is carried out for all the training sample. Once a whole cycle, all instances, has been completed, we have done an *epoch*. the practitioner must decide the number of epochs, or set some termination criterion, as hyperparameter. The Box C sets the required data

Box 2: Architecture, initial values, loss and instance.

Architecture:

- 2 input neurons (x_1, x_2)
- 1 hidden layer with 2 neurons (h_1, h_2)
- 1 output neuron y
- Sigmoid activation function, $\sigma(z) = \frac{1}{1+e^{-z}}$

Initial values for the network:

- For h_1: $w_{1,1} = 0.15$, $w_{2,1} = 0.2$, $b_1 = 0.35$,
- For h_2: $w_{1,2} = 0.2$, $w_{2,2} = 0.3$, $b_2 = 0.35$,
- For y: $w_{h,1} = 0.4$, $w_{h,2} = 0.45$, $b_h = 0.6$,

Loss function and learning rate:

- Mean Squared Error loss function, $L = \frac{1}{2}(y - y_a)^2$, where y is the predicted value (output from the network) and y_a is the actual value
- Learning rate: $\eta = 0.5$

Instance (observation):

- $x_1 = 0.05$, $x_2 = 0.1$, $y_a = 0.01$

C.1 Forward pass

The prediction is computed as follows. See Figure 16 for an illustration.

Step 1. The weighted sums at the hidden layer:

$$z_{h1} = w_{11}x_1 + w_{21}x_2 + b_1 = (0.15 \times 0.05) + (0.25 \times 0.1) + 0.35 = 0.3775$$
$$z_{h2} = w_{12}x_1 + w_{22}x_2 + b_2 = (0.2 \times 0.05) + (0.3 \times 0.1) + 0.35 = 0.3925$$

Step 2. Applying the sigmoid activation:

$$h_1 = \sigma(z_{h1}) = \frac{1}{1 + e^{-0.3775}} = 0.5933$$
$$h_2 = \sigma(z_{h2}) = \frac{1}{1 + e^{-0.3925}} = 0.5968$$

Step 3. The weighted sum at the output neuron:

$$z_y = w_{h1}h_1 + w_{h2}h_2 + b_y = (0.4 \times 0.5933) + (0.45 \times 0.5968) + 0.6 = 1.1059$$

Step 4. Applying the sigmoid activation:

$$y = \sigma(z_y) = \frac{1}{1 + e^{-1.1059}} = 0.7513$$

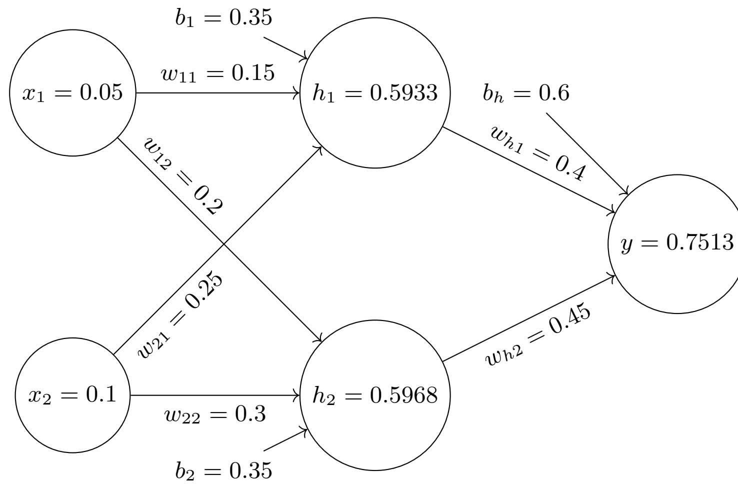


Figure 16: Flow diagram for the prediction

C.2 Backpropagation

Step 1. Output Layer Error Gradient. The loss function is the squared error:

$$E = \frac{1}{2}(y - y_a)^2, \quad \text{where } y_a = 0.01.$$

Its derivative with respect to y:

$$\frac{\partial E}{\partial y} = y - y_a = 0.7513 - 0.01 = 0.7413.$$

Derivative of the activation function:[23]

$$\frac{\partial y}{\partial z_y} = y(1 - y) = 0.7513(1 - 0.7513) = 0.1868.$$

Applying the chain rule:

$$\delta_y = \frac{\partial E}{\partial z_y} = \frac{\partial E}{\partial y} \cdot \frac{\partial y}{\partial z_y} = 0.7413 \times 0.1868 = 0.1384.$$

Step 1. Hidden Layer Error Gradients:

For h_1:

$$\begin{aligned}\frac{\partial E}{\partial z_{h1}} &= h_1(1 - h_1) \cdot w_{h1} \cdot \delta_y \\ &= 0.5933(1 - 0.5933) \times 0.4 \times 0.1384 = 0.2412 \times 0.4 \times 0.1384 = 0.0133.\end{aligned}$$

[23] We use intensively that for the sigmoid function it is: $\sigma'(z) = \sigma(z)(1 - \sigma(z))$.

For h_2:

$$\frac{\partial E}{\partial z_{h2}} = h_2(1-h_2) \cdot w_{h2} \cdot \delta_y$$
$$= 0.5968(1-0.5968) \times 0.45 \times 0.1384 = 0.2407 \times 0.45 \times 0.1384 = 0.0149.$$

Step 3. Weight Updates, taking learning rate $\eta = 0.5$

For w_{h1}:

$$w'_{h1} = w_{h1} - \eta \cdot h_1 \cdot \delta_y$$
$$= 0.4 - 0.5 \times 0.5933 \times 0.1384 = 0.3590.$$

For w_{h2}:

$$w'_{h2} = w_{h2} - \eta \cdot h_2 \cdot \delta_y$$
$$= 0.45 - 0.5 \times 0.5968 \times 0.1384 = 0.4087.$$

For w_{11}:

$$w'_{11} = w_{11} - \eta \cdot x_1 \cdot \delta_{h1}$$
$$= 0.15 - 0.5 \times 0.05 \times 0.0133 = 0.1483.$$

For w_{21}:

$$w'_{21} = w_{21} - \eta \cdot x_2 \cdot \delta_{h1}$$
$$= 0.25 - 0.5 \times 0.1 \times 0.0133 = 0.2467.$$

For w_{12}:

$$w'_{12} = w_{12} - \eta \cdot x_1 \cdot \delta_{h2}$$
$$= 0.2 - 0.5 \times 0.05 \times 0.0149 = 0.1981.$$

For w_{22}:

$$w'_{22} = w_{22} - \eta \cdot x_2 \cdot \delta_{h2}$$
$$= 0.3 - 0.5 \times 0.1 \times 0.0149 = 0.2926.$$

D Other methods

This section presents very briefly other ML methods. Some of them are closely related to some of the ideas already presented, while other tackle the problem from a different angle. As we have done through the main text, we make special emphasis on presenting clearly the core idea of each method rather than trying to cover -even mention- all the existing extensions.

D.1 LASSO, Ridge and Elastic Net Regression

A multiple linear regression model can be seen as a particular case of ANN in which there is no hidden layer. Thus, the predicted output is directly a weighted sum of instances. The weights -parameters in the econometric jargon- are learned by the network as to minimize a loss function. The usual criterion in Econometrics is to take as loss function the mean squared error. This model suffers from two well-known issues: over-fitting and multicollinearity. The first, as mentioned, refers to a model that fits the training data too closely, which is useless out of the training. The second refers to a lack of capacity to identify individual effects of each feature in the response variable.

In order to address those problems, LASSO,[24] Ridge and Elastic Net Regression add a regularization term to the loss function in a completely analogous way as we have explained for XGBoost. The loss function is that in (1), which is just the sum of the mean squared prediction error and a regularization term which is the norm of the estimated parameter vector. The norm is L_1 for the LASSO regression, L_2 for the ridge regression, and both L_1 and L_2 terms for the elastic net regression.

The underlying reason for the regularization term is as commented for the XGBoost algorithm. In general, the L_1 regularization (LASSO) pushes to zero coefficients that in absolute value are equal to zero. In this sense, the LASSO model works well for feature selection, and it also prevents over-fitting. The L_2 regularization (Ridge) shrinks all of the coefficients. It helps in dealing with multicollinearity problems in the following sense. Suppose there are two feature variables, say x_1 and x_2, such that there is a whole region of combination of parameter estimates, say (β_1, β_2), that performs similarly well in terms of prediction errors. Adding the L_2 regularization to the loss function pushes to select, within the equally-good-prediction region, small values of β_1 and β_2 (shrinkage), but it also pushes to select $\beta_1 = \beta_2$, which is a quite natural convention if the corresponding variables are collinear. Notice using only the L_1 regularization does not *force* the equality among coefficients. Elastic Net combines the L_1 and L_2 regularization penalties of LASSO and Ridge, respectively. This hybrid approach offers a balance between feature selection (LASSO) and coefficient shrinkage (Ridge). It's particularly useful when dealing with datasets that have both highly correlated features and many irrelevant ones, as it can simultaneously shrink coefficients and perform feature selection.

For analyzing factors influencing housing prices in a metropolitan area, LASSO regression could be valuable. You might have numerous features like square footage, number of bedrooms, proximity to schools, and local amenities, but only a subset truly drives price variations. LASSO's feature selection would help identify the most influential factors. In macroeconomic forecasting, where you're predicting GDP growth based on various economic indicators like interest rates, inflation, and unemployment, Ridge regression would be suitable. These indicators are often highly correlated, and Ridge's

[24]It stands for Least Absolute Shrinkage and Selection Operator.

shrinkage property would stabilize coefficient estimates, preventing extreme fluctuations due to multicollinearity. Finally, when studying the impact of various government policies on firm performance, Elastic Net could be beneficial. You might have many policy variables, some correlated, and you'd want to both identify the most impactful policies and handle the potential collinearity among them, which Elastic Net accomplishes by combining L_1 and L_2 regularization.

Why should we restrict ourselves to L_1 or L_2 regularizations? Figure 17 shows different regularisation function isosurfaces for different values of p where the L_p regulariser is defined as $||\boldsymbol{\beta}||_p := (\sum_i |\beta_i|^p)^{\frac{1}{p}}$ in a $\overline{\beta_1\beta_2}$ plane. For the Euclidean distance $p = 2$ we have the circle with radius 1, when $p = 1$ we obtain the sum of the absolute values and the isosurface corresponds to the star shape.

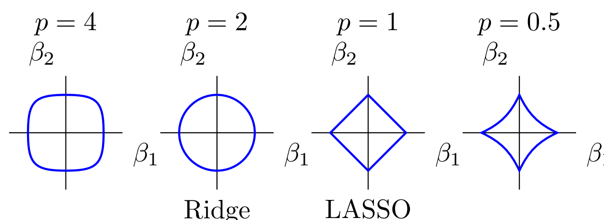


Figure 17: Isosurfaces of different L_p regularisers

The Figure 18 shows the balance between the regularizer isosurface and the prediction error isosurface, represented by the red curves. At an interior solution, both isosurfaces are tangent one another. L_1 penalties can incorporate sparse parameter vectors (a vector which lies exactly on an axis as in the *Opt* point in the left plot in Figure 18 - i.e. $\beta_1 = 0$, which emphasizes the feature selection character of this regularization. L_2 penalties will only be sparse if the minimum mean square error point $(\widehat{\beta})$ is also exactly on the axis. Therefore the L_1 optimum can be on the axis even if the $\widehat{\beta}$ is not and L_1 regularized solutions encourages some level of sparsity, making the model more efficient which helps determine which variables are most important in prediction. L_p norms form a balance between sparsity and convexity, when $p \geq 1$ the norm is convex and for $p \leq 1$ it induces some sparsity. Therefore, the L_1 norm is the only norm which both induces sparsity and remains convex for easier optimization.

As in previous sections, we conclude with some papers to illustrate the range of applications. Topics include firm analysis [27] or bankruptcy prediction [93]. Much of the emphasis is on the capacity of the LASSO loss function to perform variable selection, see [17], which is not a paper on ML, for a broad discussion on variable selection itself. Related to this issue, we must cite [95] and [113].

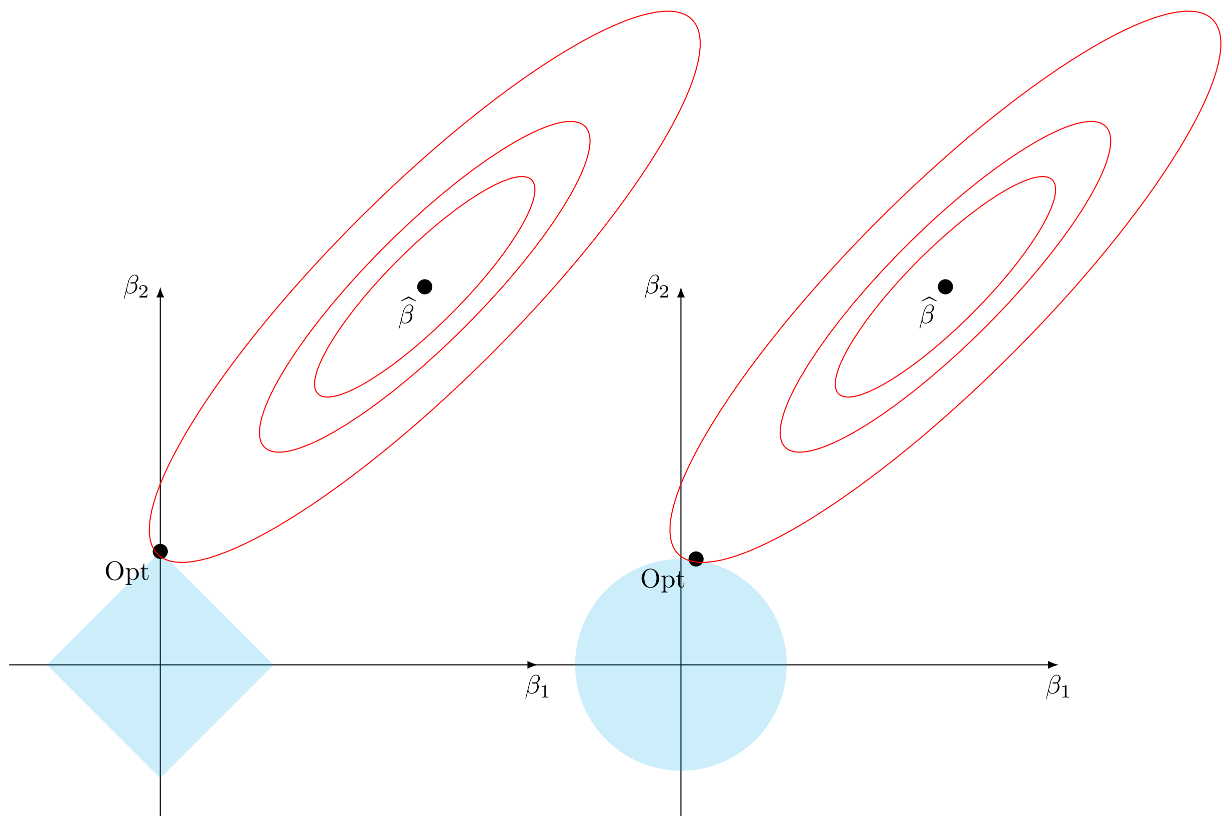


Figure 18: LASSO Ridge: trade-off between regularization (blue) and prediction error (red). The compromise solution is the Opt point: similar but not exactly the same as with the LASSO reguralization $\beta_1 = 0$, while it is not so with the Ridge. The prediction error is common to both plots.

D.2 Support Vector Machines (SVM)

Support Vector Machines (SVM) can be used for both regression and classification problems. To keep things simple, suppose that we have a binary response variable and two explanatory variables, denoted by x_1 and x_2, or, as a vector, $\boldsymbol{x} = (x_1, x_2)$. The response classes, say, are coded as 1 and -1 or, black and blue, respectively, so that our sample is as depicted in Figure 19. The target of Support Vector Machine is to find a *separating* hyperplane, that is, an hyperplane that leaves all of the instances of one class on one side and all of the instances of the other class on the other side. In a two-dimensional space of explanatory variables, as in the figure, the hyperplane is a line, but our discussion and mathematical development in this subsection does not imply a particular dimension.

Let us considering a case, as in the figure, in which there is such a separating hyperplane. Of course, if there is one, there is more than one, and then the question is, roughly, how can I characterize the maximum distance between classes. The core idea

is as follows. Consider two hyperplanes parallel one another such that they define a *separating strip*. The question is what are the hyperplanes such that the corresponding separating strip has the maximum width? This maximum width is commonly denoted as *the margin*. So, rephrasing, we want to find paralell separating hyperplanes that maximize the margin, i.e., the distance between them. In doing so, some sample points, red coloured in the figure, actually touch one of the hyperplanes, these points are called the *support vectors*.

The mathematical characterization is as follows. First let us denote the hyperplanes as $H_{(1)} = \{\boldsymbol{x} : \boldsymbol{w} \cdot \boldsymbol{x} + b = 1\}$ and $H_{(-1)} = \{\boldsymbol{x} : \boldsymbol{w} \cdot \boldsymbol{x} + b = -1\}$, where $\boldsymbol{w}$ is a vector of coefficients, $\boldsymbol{w} \cdot \boldsymbol{x}$ denotes the dot product between $\boldsymbol{w}$ and $\boldsymbol{x}$ and b is some constant. Since $\boldsymbol{w}$ is common to both hyperplanes, they are paralell one another.[25] The distance between $H_{(1)}$ and $H_{(-11)}$ is[26] $\frac{2}{||\boldsymbol{w}||}$, where the denominator is the norm of $\boldsymbol{w}$. Thus, in order to maximize the margin between hyperplanes, we have to minimize $||\boldsymbol{w}||$. Now we must restrict to hyperplanes which separate instances by classes. Let us take the convention that instances $(\boldsymbol{x}_i, y_i)$ with $y_i = 1$ lie *above* $H_{(1)}$, which happens whenever $\boldsymbol{w} \cdot \boldsymbol{x}_i + b \geq 1$ holds. Clearly, for those instances the previous inequality is equivalent to:

$$y_i \left(\boldsymbol{w} \cdot \boldsymbol{x}_i + b\right) \geq 1 \tag{8}$$

Then, it must occur that instances with $y_i = -1$ lie *below* $H_{(-1)}$, which happens whenever $\boldsymbol{w} \cdot \boldsymbol{x}_i + b \leq -1$ holds, which for those instances is also equivalent to (8). Then, the problem can be formulated as minimizing $\boldsymbol{w}$ subject to (8) holding for all instances, which are indexed by i.

Naturally, the training data might be not separable. If so, the called *soft margin* comes into scene. Essentially, the method defines some slack variables, such that some of the instances are allowed to by misclassified. There is a trade-off between maximizing the margin, as before, and allowing some instances to lie out of the boundaries where they *should be*. Alternatively, if data are not separable in the original dimension, they can be transformed into a higher dimensional space in which they are. Once transformed, instead of using the dot product in the higher dimensional space, the method uses a kernel function.

Some of the classical works on SVM are [29], [52], [89] and, as one of the main developers, [102]. Additionally, one textbook from this last author, [101] offers a detailed description of SVM's and, more generally, statistical learning. Regarding the application

[25] Obviously, no $\boldsymbol{x}$ can satisfy both the equality that characterizes $H_{(1)}$ and the one for $H_{(-1)}$. In other words, $H_{(1)}$ and $H_{(-11)}$ do not cross, so they are parallel.

[26] In effect, take $\boldsymbol{x}_1 \in H_{(1)}$ and take $\boldsymbol{x}_2 \in H_{(-11)}$, that is, $\boldsymbol{w} \cdot \boldsymbol{x}_1 + b = 1$ and $\boldsymbol{w} \cdot \boldsymbol{x}_2 + b = -1$. Subtracting one equality from the other, we have $\boldsymbol{w} \cdot (\boldsymbol{x}_1 - \boldsymbol{x}_2) = 2$ or, writing the dot product as the product of the norms times the cosine of the angle between, say α, we have $||\boldsymbol{w}|| \times ||\boldsymbol{x}_1 - \boldsymbol{x}_2|| cos\alpha = 2$. Now, it is well known that $\boldsymbol{w}$ is normal to $H_{(1)}$ and to $H_{(-1)}$. Thus, if $\boldsymbol{x}_1 - \boldsymbol{x}_2$ is a vector of minimum length that connects both hyperplanes, it has to be collinear with $\boldsymbol{w}$, and thus $cos\alpha = 1$ and, in addition, $||\boldsymbol{x}_1 - \boldsymbol{x}_2||$ is the distance between the hyperplanes.

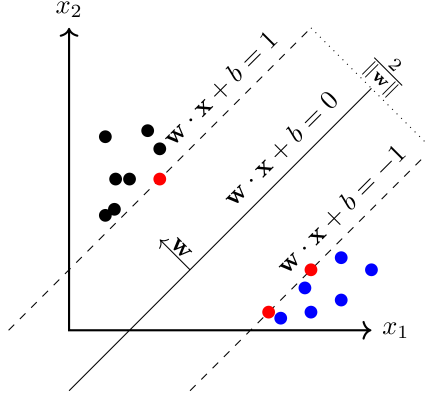


Figure 19: Support Vector Machine example

in economics, we have: currency crisis [23], stock prices [34], recessions [50] and financial markets [98]. [94] surveys a wide range of other applications.

D.3 Shapley Values

Shapley values, originally from cooperative game theory, [86], quantify the contribution of each feature to a model's prediction. Unlike methods that retrain models, Shapley values analyze the impact of a feature's value on individual predictions within a pre-trained model. This analysis considers all possible combinations of feature contributions, providing a comprehensive measure of feature importance.

To be precise, let F denote the pre-trained ML model. Let $\boldsymbol{x}_i$ represent the vector of features for instance i. A *coalition of features*, or simply a coalition, is any subset of the entries of $\boldsymbol{x}_i$. Let $F(\boldsymbol{s}_i)$ denote the prediction of y_i using the coalition $\boldsymbol{s}_i$. The contribution of feature, say, $x_{k,i}$ to a coalition $\boldsymbol{s}_i$ in which $x_{k,i}$ is included is $F(\boldsymbol{s}_i) - F(\boldsymbol{s}_i/\{x_{k,i}\})$, that is, the difference between the prediction using $\boldsymbol{s}_i$ and the prediction using $\boldsymbol{s}_i$ excluding $x_{k,i}$. The Shapley value of feature $x_{k,i}$ for instance $\boldsymbol{x}_i$ is the average of its contribution to all possible coalitions the can be formed with $\boldsymbol{x}_i$.

We illustrate the computation with an example. Let us consider a model with three attributes, so features for instance i are $\boldsymbol{x}_i = (x_{1,i}, x_{2,i}, x_{3,i})$. For simplicity, let us assume the response in continuous. The table on the left shows all possible coalitions and the corresponding prediction. In particular, the $\emptyset$ coalition is a coalition containing no features of *any* instance, so it must be interpreted as the unconditional prediction, which we normalize to be 0. Thus $F(x_{1,i}) = 80$ must be read as: using a coalition formed just by $x_{1,i}$ the prediction is 80 units above the unconditional prediction. This left table has the raw information to compute Shapley values. The table on the right will help us to compute them. Consider the first row in the table. If, starting from the $\emptyset$ coalition, I add

first $x_{i,1}$ the variation in -or marginal- prediction is $F(x_{i,1}) - F(\emptyset) = 80$. If, after that, I add $x_{2,i}$, the variation in prediction is $F((x_{1,i}, x_{2,i})) - F(x_{1,i}) = 0$ and, finally, if, after that, I add $x_{i,3}$, the variation is $F(\boldsymbol{x}_i) - F((x_{1,i}, x_{2,i})) = 10$. Thus, the first cell on the row contains the order in which I add features, while second contains the corresponding marginal predictions. The first column in the table on the right contains all possible permutations, that is, all possible paths to add the features. Now, for computing the Shapley value of, say, $x_{1,i}$ we must see where it is located on the row on the first column and take the corresponding number from the second column, referred to the table on the right. In order to ease the reading, we have highlighted the position and numbers for $x_{1,i}$ in red. The Shapley value of that feature is the average of those numbers:

$$\text{Shap}(x_{1,i}, \boldsymbol{x}_i) = \frac{1}{3!}(80 + 80 + 24 + 18 + 15 + 18) = 39.2$$

where 3! stands for three factorial, that is, the number of possible permutations with three features.

$\boldsymbol{s}_i$	$F(\boldsymbol{s}_i)$
$\emptyset$	0
$(x_{1,i})$	80
$(x_{2,i})$	56
$(x_{3,i})$	70
$(x_{1,i}, x_{i,2})$	80
$(x_{1,i}, x_{i,3})$	85
$(x_{2,i}, x_{i,3})$	72
$\boldsymbol{x}_i$	90

Permutation	Marginal pred.
$(x_{1,i}, x_{2,i}, x_{i,3})$	$(80, 0, 10)$
$(x_{1,i}, x_{i,3}, x_{2,i})$	$(80, 5, 5)$
$(x_{2,i}, x_{1,i}, x_{i,3})$	$(56, 24, 10)$
$(x_{2,i}, x_{i,3}, x_{1,i})$	$(56, 16, 18)$
$(x_{i,3}, x_{1,i}, x_{2,i})$	$(70, 15, 15)$
$(x_{i,3}, x_{2,i}, x_{1,i})$	$(70, 2, 18)$

The above computations lead to a Shapley value for each feature and for each instance. How do we aggregate across instances? There are essentially two ways. First, we can simply average across *all* instances, which leads to the *global feature importance*:

$$\text{Shap}(x_1) = \frac{1}{N} \sum_i \text{Shap}(x_{1,i}, \boldsymbol{x}_i)$$

where N denotes the sample size. Secondly, we can condition by features. As an example, we can average across instances having, say, $x_{1,i} \leq 5$, which leads to the *conditional feature importance*. This is very appealing in economics, where is natural to think that the impact of a variable in prediction depends on the values of that variable. For instance, conditional Shapley value analysis could reveal that wealth's contribution to predicting loan repayment is not uniform, and might demonstrate a diminishing effect for high-wealth individuals relative to those with low wealth.

Finally, we must notice that the above explanation abstracts away from the model itself. In particular, it is implicitly assumed that the model can generate predictions for any coalition, even for the $\emptyset$ coalition. In other words, the model, at least once trained,

must be able to handle missing values, which could be an issue for some methods.

Other than the cited Shapley's paper, which is really a paper on cooperative game theory, there is a literature on the use of the concept in ML, which relates Shapley value to the concept of *importance* or connects it to the idea of gradient. The essential list in this line includes [20], [30], [65], [91], [114] and [92]. [5] combines XGBoost and Shapley values for the retail sector. [104] extends the theoretical framework with an angle on application.

E Python code for the XGBoost study case

Listing 2: Creating synthetic data and define plot parameters

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import accuracy_score
from xgboost import XGBClassifier

random_state = 123456789
X, y = make_classification(
    n_samples = 1000,
    n_features = 10,
    n_informative = 2,
    n_redundant = 0,
    n_clusters_per_class = 1,
    flip_y = 0,
    class_sep = 0.5,
    random_state = random_state)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,
    random_state = random_state)

# Generate data for the decision tree boundaries
plt.style.use('fivethirtyeight')
plt.figure(figsize=(10, 6))

x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.01),
                     np.arange(y_min, y_max, 0.01))
```

Listing 3: XGBoost Cross Validation and Hyperparameter Search

```
param_grid = {
    'max_depth': [3, 5, 7],            # Maximum depth of each tree.
    'learning_rate': [0.1, 0.01, 0.001], # Step size shrinkage used to update
        weights.
    'n_estimators': [50, 100, 200],    # Number of trees the model builds,
        e.g., 50 trees.
    'subsample': [0.8, 1.0],           # Fraction of samples used for fitting
        individual trees.
    'colsample_bytree': [0.8, 1.0]     # Fraction of features/variables used
        when constructing each tree.
}

xgb = XGBClassifier(random_state=random_state, use_label_encoder=False,
    eval_metric='logloss')

grid_search = GridSearchCV(estimator=xgb,
                        param_grid=param_grid,
                        cv=5,
                        scoring='accuracy',
                        verbose=1)

grid_search.fit(X_train, y_train)

best_xgb = grid_search.best_estimator_

y_pred = best_xgb.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)
print("Best Parameters:", grid_search.best_params_)
print("XGBoost Classifier Test Accuracy:", accuracy)
```

Listing 4: Feature importance

```
importances = best_xgb.feature_importances_
feature_names = [f"Feature {i}" for i in range(X.shape[1])]

# Get indices for sorting the importances in descending order
sorted_indices = np.argsort(importances)[::-1]
sorted_importances = importances[sorted_indices]
sorted_feature_names = [feature_names[i] for i in sorted_indices]

plt.figure(figsize=(10, 6))
sns.barplot(x=sorted_importances, y=sorted_feature_names, palette="viridis")
plt.title("Feature Importance (Highest to Lowest)")
plt.xlabel("Importance")
plt.ylabel("Features")
plt.tight_layout()
plt.show()
```

Listing 5: Shapley values

```
import pandas as pd
import shap

feature_names = [f"Feature_{i}" for i in range(X.shape[1])]
X_train_df = pd.DataFrame(X_train, columns=feature_names)
X_test_df = pd.DataFrame(X_test, columns=feature_names)

explainer = shap.TreeExplainer(best_xgb)
shap_values = explainer.shap_values(X_test_df)

shap.summary_plot(shap_values, X_test_df, show=False)
plt.title("SHAP Summary Plot")
plt.show()

shap.summary_plot(shap_values, X_test_df, plot_type="bar", show=False)
plt.title("SHAP Feature Importance (Bar Plot)")
plt.show()

for feature in feature_names:
    shap.dependence_plot(feature, shap_values, X_test_df, show=False)
    plt.title(f"SHAP Dependence Plot for {feature}")
    plt.show()


shap.initjs()
sample_index = 0 # Change this index to analyze different instances.
force_plot = shap.force_plot(explainer.expected_value,
    shap_values[sample_index, :],
                            X_test_df.iloc[sample_index, :])
shap.save_html("force_plot.html", force_plot)

shap.waterfall_plot(
    shap.Explanation(values=shap_values[sample_index, :],
                     base_values=explainer.expected_value,
                     data=X_test_df.iloc[sample_index, :],
                     feature_names=feature_names)
)

shap.decision_plot(explainer.expected_value, shap_values[sample_index, :],
                   X_test_df.iloc[sample_index, :])
plt.show()

interaction_values = explainer.shap_interaction_values(X_test_df)
if isinstance(interaction_values, list):
    interaction_values = interaction_values[1]
```

```
shap.summary_plot(interaction_values, X_test_df, plot_type="dot", show=False)
plt.title("")
plt.show()
```

Listing 6: Simple Neural Network

```
from sklearn.neural_network import MLPClassifier

nn_model = MLPClassifier(hidden_layer_sizes=(20, 10), activation='relu',
    solver='adam',
                         max_iter=500, random_state=random_state)

nn_model.fit(X_train, y_train)

y_scores = nn_model.predict_proba(X_test)[:, 1]

y_pred = nn_model.predict(X_test)

nn_accuracy = accuracy_score(y_test, y_pred)
print("Neural Network Test Accuracy:", nn_accuracy)
```

Listing 7: Random Forest

```

from sklearn.ensemble import RandomForestClassifier

rf_model = RandomForestClassifier(n_estimators=100, random_state=random_state)

rf_model.fit(X_train, y_train)

y_scores_rf = rf_model.predict_proba(X_test)[:, 1]

y_pred_rf = rf_model.predict(X_test)

rf_accuracy = accuracy_score(y_test, y_pred_rf)
print("Random Forest Test Accuracy:", rf_accuracy)
```

References

[1] Benjamin Abadi, Ewa Poirier, and George Westerman. Adaboost in labor economics: Predicting labor market transitions. *Journal of Labor Economics*, 36(3):655–680, 2018.

[2] Gabriel M. Ahlfeldt et al. (decision) trees and (random) forests: Urban economics, historical data, and machine learning. *CEPR*, 2019.

[3] Tawfiq Al-Saba and Ibrahim El-Amin. Artificial neural networks as applied to long-term demand forecasting. *Artificial Intelligence in engineering*, 13(2):189–197, 1999.

[4] Yali Amit and Donald Geman. Shape quantization and recognition with randomized trees. *Neural Computation*, 9(7):1545–1588, 1997.

[5] Evgeny A. Antipov and Elena B. Pokryshevskaya. Interpretable machine learning for demand modeling with high-dimensional data using gradient boosting machines and shapley values. *Journal of Revenue and Pricing Management*, 20:26–41, 2021.

[6] Susan Athey. Beyond prediction: Using big data for policy problems. *Science*, 363(6424):374–376, 2019.

[7] Susan Athey. The impact of machine learning on economics. In Ajay Agrawal, Joshua Gans, and Avi Goldfarb, editors, *The Economics of Artificial Intelligence*, pages 507–547. University of Chicago Press, 2019.

[8] Susan Athey, Mohsen Bayati, Guido Imbens, and Zhaonan Qu. Ensemble methods for causal effects in panel data settings. In *AEA Papers and Proceedings*, volume 109, pages 65–70, 2019.

[9] Damilola Elizabeth Babatunde, Ambrose Anozie, and James Omoleye. Artificial neural network and its applications in the energy sector: an overview. *International Journal of Energy Economics and Policy*, 10(2):250–264, 2020.

[10] Sami Ben Jabeur, Nicolae Stef, and Pedro Carmona. Bankruptcy prediction using the xgboost algorithm and variable importance feature engineering. *Computational Economics*, 61(2):715–741, 2023.

[11] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning deep architectures for ai. *Foundations and Trends in Machine Learning*, 2(1):1–127, 2007.

[12] Daniel Borup and Jonas Schütte. The macroeconomy as a random forest. *arXiv preprint arXiv:2006.12724*, 2023.

[13] Leo Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

[14] Leo Breiman. Stacked regressions. *Machine learning*, 24(1):49–64, 1996.

[15] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[16] Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. *Classification and regression trees.* CRC press, 1984.

[17] Leo Breiman and Philip Spector. Submodel selection and evaluation in regression. the x-random case. *International statistical review/revue internationale de Statistique*, pages 291–319, 1992.

[18] Ian Brown and Christophe Mues. Practical application of machine learning in credit scoring. *Expert Systems with Applications*, 39(10):7288–7298, 2012.

[19] Roman Bussmann, Leonardo Iania, and Michael Sigmund. Machine learning for credit risk prediction: a comparative study. *Journal of Banking & Finance*, 129:106208, 2021.

[20] J. Castro, D. Gómez, and J. Tejada. Individual feature selection in linear regression models. *Group Decision and Negotiation*, 18(1):47–67, 2009.

[21] Aaron Chalfin, Oren Danieli, Andrew Hillis, Zubin Jelveh, Michael Luca, Jens Ludwig, and Sendhil Mullainathan. Productivity and selection of human capital with machine learning. *American Economic Review*, 106(5):124–27, 2016.

[22] Vasant Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM Computing Surveys (CSUR)*, 41(3):1–58, 2009.

[23] Arindam Chaudhuri. Support vector machine model for currency crisis discrimination. *arXiv preprint arXiv:1403.0481*, 2014.

[24] Huijun Chen. Enterprise marketing strategy using big data mining technology combined with xgboost model in the new economic era. *Plos one*, 18(6):e0285506, 2023.

[25] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 785–794. ACM, 2016.

[26] Xiaohong Chen, Jeffrey Racine, and Norman R Swanson. Semiparametric arx neural-network models with an application to forecasting inflation. *IEEE Transactions on neural networks*, 12(4):674–683, 2001.

[27] Alex Coad and Stjepan Srhoj. Catching gazelles with a lasso: Big data techniques for the prediction of high-growth firms. *Small Business Economics*, 55:541–565, 2020.

[28] Philip K. Coats and Lawrence F. Fant. Recognizing financial distress patterns using a neural network tool. *Financial management*, pages 142–155, 1993.

[29] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[30] Ian Covert, Scott M. Lundberg, and Su-In Lee. Explaining by removing: A unified framework for model explanation. *Journal of Machine Learning Research*, 22(235):1–90, 2021.

[31] Jesse Davis and Mark Goadrich. The relationship between precision-recall and roc curves. In *Proceedings of the 23rd international conference on Machine learning*, pages 233–240. ACM, 2006.

[32] Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407, 1990.

[33] Thomas G Dietterich. Ensemble methods in machine learning. In *International workshop on multiple classifier systems*, pages 1–15. Springer, 2000.

[34] Zhu Ding. Application of support vector machine regression in stock price forecasting. In *Business, Economics, Financial Sciences, and Management*, pages 395–403, 2012.

[35] Jörg Döpke, Ulrich Fritsche, and Christian Pierdzioch. Predicting recessions with boosted regression trees. *International Journal of Forecasting*, 33(4):745–759, 2017.

[36] Bradley Efron. Bootstrap methods: another look at the jackknife. In *Breakthroughs in statistics*, pages 569–593. Springer, 1992.

[37] Liran Einav and Jonathan Levin. The data revolution and economic analysis. *Innovation Policy and the Economy*, 14(1):1–24, 2014.

[38] Liran Einav and Jonathan Levin. Economics in the age of big data. *Science*, 346(6210):1243089, 2014.

[39] Tom Fawcett. An introduction to roc analysis. *Pattern Recognition Letters*, 27(8):861–874, 2006.

[40] Jonathan M. Freeman, Hongtao Zha, and Xiang Weng. Boosting auction theory models with machine learning techniques: Applications in market design. *Operations Research*, 60(2):480–494, 2012.

[41] Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, 55(1):119–139, 1997.

[42] Jerome H. Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

[43] Yulia Granitsa. Assessment of factors of regional economic stability using the xgboost algorithm: 018. In *Dela Press Conference Series: Economics, Business and Management*, number 001, pages 7–7, 2022.

[44] Yuqing Gu. The application of random forest in individual credit risk management. *Proceedings of the International Conference on Management, Computer and Software*, 2019.

[45] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* Springer, New York, 2nd edition, 2009.

[46] Zhe He, Jun Liu, and Shuang Xu. Predicting housing market trends using adaboost and decision trees. *Journal of Real Estate Finance and Economics*, 58(4):551–573, 2019.

[47] Geoffrey E. Hinton, Simon Osindero, and Yee Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[48] John J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79(8):2554–2558, 1982.

[49] Sami Ben Jabeur, Salma Mefteh-Wali, and Jean-Laurent Viviani. Forecasting gold price with the xgboost algorithm and shap interaction values. *Annals of Operations Research*, 334(1):679–699, 2024.

[50] Alexander James, Yaser S. Abu-Mostafa, and Xiao Qiao. Nowcasting recessions using the svm machine learning algorithm. *arXiv preprint arXiv:1903.03202*, 2019.

[51] Jig Han Jeong, Jonathan P Resop, Nathaniel D Mueller, David H Fleisher, Kyungdahm Yun, Ethan E Butler, Dennis J Timlin, Kyo-Moon Shim, James S Gerber, Vangimalla R Reddy, et al. Random forests for global and regional crop yield predictions. *PLoS One*, 11(6):e0156571, 2016.

[52] Thorsten Joachims. Text categorization with support vector machines: Learning with many relevant features. *Machine Learning*, 1398:137–142, 1998.

[53] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[54] Soteris A Kalogirou. Artificial neural networks in renewable energy systems applications: a review. *Renewable and sustainable energy reviews*, 5(4):373–401, 2001.

[55] Gordon V Kass. An exploratory technique for investigating large quantities of categorical data. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 29(2):119–127, 1980.

[56] Michael J. Kearns and Leslie G. Valiant. Cryptographic limitations on learning boolean formulae and finite automata. *Journal of the ACM (JACM)*, 41(1):67–95, 1994.

[57] Jon Kleinberg, Jens Ludwig, Sendhil Mullainathan, and Ziad Obermeyer. Prediction policy problems. *American Economic Review*, 105(5):491–495, 2015.

[58] Ron Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, volume 14, pages 1137–1145. Montreal, Canada, 1995.

[59] Yunhua Lai, Jian Zhang, and Guang Wu. Boosting in energy economics: Predicting household energy consumption. *Energy Economics*, 56:210–218, 2016.

[60] Yann LeCun, Bernhard Boser, John S. Denker, Donnie Henderson, Richard E. Howard, Wayne Hubbard, and Lawrence D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[61] Stefan Lessmann, Björn Baesens, Hung-Chieh Chin, Pierre Rombouts, and Robert Stenius. Benchmarking state-of-the-art classification algorithms for credit scoring: Large-scale empirical analysis of logistic regression, decision trees, and support vector machines. *Expert Systems with Applications*, 42(1):655–673, 2015.

[62] Michael Littwin, Yang Dai, and Steven Sullivan. Credit scoring using adaboost and other machine learning techniques. *Journal of Banking and Finance*, 55:199–211, 2015.

[63] Inna Logunova. Random forest classifier: Basic principles and applications. *Serokell Blog*, 2022.

[64] Wei-Yin Loh. Fifty years of classification and regression trees. *International Statistical Review*, 82(3):329–348, 2014.

[65] Scott M. Lundberg and Su-In Lee. A unified approach to interpreting model predictions. *Advances in Neural Information Processing Systems (NeurIPS)*, 30, 2017.

[66] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[67] Marcelo C. Medeiros, Gabriel F. R. Vasconcelos, and Eduardo Zilberman. Forecasting inflation with a large number of predictors: is machine learning helpful? *International Journal of Forecasting*, 37(1):138–155, 2021.

[68] Robert Messenger and Lewis Mandell. A modal search technique for predictive nominal scale multivariate analysis. *Journal of the American statistical association*, 67(340):768–772, 1972.

[69] James N Morgan and John A Sonquist. Problems in the analysis of survey data, and a proposal. *Journal of the American statistical association*, 58(302):415–434, 1963.

[70] Sendhil Mullainathan and Jann Spiess. Machine learning: an applied econometric approach. *Journal of Economic Perspectives*, 31(2):87–106, 2017.

[71] Yetis Sazi Murat and Halim Ceylan. Use of artificial neural networks for transport energy demand modeling. *Energy policy*, 34(17):3165–3172, 2006.

[72] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective.* MIT Press, Cambridge, MA, 2012.

[73] Emi Nakamura. Inflation forecasting using a neural network. *Economics Letters*, 86(3):373–378, 2005.

[74] David M. W. Powers. Evaluation: From precision, recall and f-measure to roc, informedness, markedness & correlation. *Journal of Machine Learning Technologies*, 2(1):37–63, 2011.

[75] Sushil Punia, Sandeep Singh, and Manjeet Kadyan. Demand forecasting using random forest and artificial neural network. In *Proceedings of the International Conference on Intelligent Computing and Applications*, pages 219–227, 2019.

[76] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[77] John Quinlan. *C4. 5: Programs for machine learning.* Morgan Kaufmann Publishers, 1993.

[78] Felipe A. Ramos, Mario E. Garcia, and Roberto M. Batista. Adaboost in healthcare economics: A study of cost prediction for chronic diseases. *Health Economics Review*, 3(1):19, 2013.

[79] Apostolos N. Refenes, Afshin A. Azemi, and Andreas D. Zapranis. Stock performance modeling using neural networks: A comparative study with regression models. *Neural networks*, 7(2):375–388, 1994.

[80] Julian Rodriguez, Nancy L. Hwang, and Ritesh Singh. Market structure and competition in the digital economy: Using adaboost to model competition dynamics. *Journal of Industrial Economics*, 63(2):367–386, 2015.

[81] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in some of the brain. *Psychological review*, 65(6):386, 1958.

[82] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[83] Robert E. Schapire. The strength of weak learnability. *Machine Learning*, 5(2):197–227, 1990.

[84] Robert E Schapire and Yoav Freund. Boosting: Foundations and algorithms. *Kybernetes*, 2013.

[85] Robert E Schapire, Yoav Freund, Peter Bartlett, Wee Sun Lee, et al. Boosting the margin: A new explanation for the effectiveness of voting methods. *The annals of statistics*, 26(5):1651–1686, 1998.

[86] Lloyd S. Shapley. A value for n-person games. 2:307–318, 1953.

[87] Aaron Smalter Hall and Thomas R Cook. Macroeconomic indicator forecasting with deep neural networks. *Federal Reserve Bank of Kansas City Working Paper*, (17-11), 2017.

[88] Matthew Smith and Francisco Alvarez. Predicting firm-level bankruptcy in the spanish economy using extreme gradient boosting. *Computational Economics*, 59(1):263–295, 2022.

[89] Alex J. Smola and Bernhard Schölkopf. A tutorial on support vector regression. *Statistics and Computing*, 14(3):199–222, 2004.

[90] Hugo Storm, Kathy Baylis, and Thomas Heckelei. Machine learning in agricultural and applied economics. *European Review of Agricultural Economics*, 47(3):849–892, 2020.

[91] Erik Strumbelj and Igor Kononenko. An efficient explanation of individual classifications using game theory. *Journal of Machine Learning Research*, 11:1–18, 2010.

[92] Mukund Sundararajan and Amir Najmi. The many shapley values for model explanation. In *International Conference on Machine Learning (ICML)*, pages 9269–9278, 2020.

[93] Shaonan Tian, Yan Yu, and Hui Guo. Variable selection and corporate bankruptcy forecasts. *Journal of Banking & Finance*, 52:89–100, 2015.

[94] Yanfeng Tian, Yong Shi, and Xiaohui Liu. Recent advances on support vector machines research. *Technological and Economic Development of Economy*, 18(1):5–33, 2012.

[95] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288, 1996.

[96] Kais Tissaoui, Taha Zaghdoudi, Abdelaziz Hakimi, and Mariem Nsaibi. Do gas price and uncertainty indices forecast crude oil prices? fresh evidence through xgboost modeling. *Computational Economics*, 62(2):663–687, 2023.

[97] Greg Tkacz. Neural network forecasting of canadian gdp growth. *International Journal of Forecasting*, 17(1):57–69, 2001.

[98] Theodore B. Trafalis and Huseyin Ince. Support vector machine for regression and applications to financial forecasting. *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks (IJCNN 2000)*, 6:348–353, 2000.

[99] Wei Tsai, Chih-Ming Hsu, and Yi-Ting Chou. Fraud detection in credit card transactions using adaboost. *International Journal of Computer Applications in Technology*, 45(3):203–215, 2011.

[100] Leslie G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.

[101] Vladimir Vapnik. Statistical learning theory. *Wiley*, 1998.

[102] Vladimir N. Vapnik and Alexey Y. Chervonenkis. *Theory of Pattern Recognition.* Nauka, Moscow, 1974.

[103] Hal R. Varian. Big data: New tricks for econometrics. *Journal of Economic Perspectives*, 28(2):3–28, 2014.

[104] David Watson, Joshua O'Hara, Niek Tax, Richard Mudd, and Ido Guy. Explaining predictive uncertainty with information theoretic shapley values. *Advances in Neural Information Processing Systems*, 37, 2024.

[105] Jesse West and Mainak Bhattacharya. Intelligent financial fraud detection: A comprehensive review. *Computers & Security*, 57:47–66, 2016.

[106] Mochen Yang, Edward McFowland III, Gordon Burtch, and Gediminas Adomavicius. Achieving reliable causal inference with data-mined variables: A random forest approach to the measurement error problem. *arXiv preprint arXiv:2012.10790*, 2020.

[107] Mochen Yang, Edward McFowland III, Gordon Burtch, and Gediminas Adomavicius. Application of ai in credit risk scoring for small business enterprises. *arXiv preprint arXiv:2410.05330*, 2024.

[108] Lean Yu, Shouyang Wang, and Kin Keung Lai. Forecasting crude oil price with an emd-based neural network ensemble learning paradigm. *Energy Economics*, 30(5):2623–2635, 2008.

[109] Shipei Zeng and Deyu Rao. Random forests with economic roots: Explaining machine learning in hedonic imputation. *Computational Economics*, 2024.

[110] Yan Zhang and Lin Chen. A study on forecasting the default risk of bond based on xgboost algorithm and over-sampling method. *Theoretical economics letters*, 11(2):258–267, 2021.

[111] Ling Zhou, Jianwei Zhang, Bo Li, and Qiaozhen Liu. Bankruptcy prediction based on machine learning: A comprehensive survey. *Expert Systems with Applications*, 196:116601, 2022.

[112] Zhang Zhou, Zhi Zheng, and Li He. Boosting for demand forecasting. *Computers and Operations Research*, 38(3):789–799, 2011.

[113] Hui Zou and Trevor Hastie. Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 67(2):301–320, 2005.

[114] Erik Štrumbelj and Igor Kononenko. Explaining prediction models and individual predictions with feature contributions. *Knowledge and Information Systems*, 41(3):647–665, 2014.