Generating Actionable Interpretations from Ensembles of Decision Trees

Gabriele Tolomei and Fabrizio Silvestri

Abstract—Machine-learned models are often perceived as "black boxes": they are given inputs and hopefully produce desired outputs. There are many circumstances, however, where human-interpretability is crucial to understand *(i)* why a model outputs a certain prediction on a given instance, *(ii)* which adjustable features of that instance should be modified, and finally *(iii)* how to alter a prediction when the mutated instance is input back to the model.
In this paper, we present a technique that exploits the feedback loop originated from the internals of any ensemble of decision trees to offer recommendations for transforming a k-labelled predicted instance into a k'-labelled one (for any possible pair of class labels k, k'). Our proposed algorithm perturbs individual feature values of an instance, so as to change the original prediction output by the ensemble on the so-transformed instance. This is also achieved under two constraints: the *cost* and *tolerance* of transformation. Finally, we evaluate our approach on four distinct application domains: online advertising, healthcare, spam filtering, and handwritten digit recognition. Experiments confirm that our solution is able to suggest changes to feature values that help interpreting the rationale of model predictions, making it indeed useful in practice especially if implemented efficiently.

Index Terms—Machine learning interpretability, Actionable feature tweaking, Recommending feature changes, Altering model predictions, Ensemble of decision trees.

1 Introduction

An increasing number of companies and organizations rely on machine learning (ML) techniques to extract knowledge from the large volumes of data they collect every day, therefore to run their business more productively.

ML solutions are usually considered as "black boxes": they take some inputs and produce desired outputs. As long as ML models work properly, little attention is devoted to understand why they achieve such surprisingly predictive accuracy, or how robust they are when in presence of a malicious adversary [1]. Still, it is beneficial to have available techniques supporting humans in interpreting and "debugging" these models, particularly when they fail [2] or lead to some oddities.[1]

Excluding recent trends in ML like representation learning [3], [4], typically the initial effort when designing an ML solution consists in manually modelling the objects of a given domain of interest using human knowledge, i.e., feature engineering. This step, often time-consuming, aims to describe each object in the domain using an appropriate set of properties (*features*), which define a so-called *feature space*. For a given dataset, each object can be considered as a static point located in the feature space, since each feature value is deemed fixed; once a model is learned from the data, each prediction it makes on new objects is irreversible[2].

Let us suppose, for example, that we disagree with a prediction that a model returns for a given object, or that we would like to enforce switching such a prediction to a different outcome. Therefore, in this work we address the following research question: "*How can we understand what can be changed in the input feature vector in order to modify the prediction accordingly?*"

To better clarify this challenge with an example, consider an ML application in the healthcare domain, where patients (objects) are mapped to a vector of clinical indicators (features), such as age, blood pressure, daily carbohydrate absorbed, etc. Assume next that an ML model has been designed to accurately predict from these features whether a patient is at risk of a heart attack or not. If for a given patient our model predicts that there is a high risk of a heart attack it would be of great advantage for medical physicians to also have a tool that suggests the most appropriate clinical treatment by offering targeted adjustments to specific indicators (e.g., reducing the daily amount of carbohydrate). In other words, to recommend the clinical treatment to switch a patient from being of high risk to low risk.

In a past work of ours [5], we introduced an algorithm for *tweaking* input features to change the output predicted by an existing machine-learned model. The mechanism was originally designed to operate on top of any ensemble of bagged decision trees [6], yet limited to binary classification. This paper extends it to the most general case of multiclass classification task. We exploit specific characteristics of the model's internals to generate recommendations allowing the transformation of an instance whose (predicted) label is k into another instance whose label is predicted being k' by the same classifier (k and k' are possible class labels).

We describe the theoretical framework of the *Feature Tweaking Problem*, along with our solution and how to implement it efficiently. Our approach is then evaluated on four distinct application domains: online advertising, healthcare, spam filtering, and handwritten digit recognition. The first three are classical examples of binary classification tasks, whilst the latter refers to a well-known multiclass classification problem. More specifically, we use the first three domains to show how our algorithm can be used to automatically generate "interpretable" and "actionable"

[1] http://www.telegraph.co.uk/technology/2016/03/24/microsofts-teen-girl-ai-turns-into-a-hitler-loving-sex-robot-wit/

[2] The immutability of a model's prediction on an instance holds at least until the model is re-trained and updated.

suggestions to convert: a low quality advertisement into a high quality one, a diabetic patient into a healthy one, and a spam email into a non-spam one. Such insights may in turn be used to achieve different goals. In the case of online advertising, advertisers may improve their return on investment if they implement recommended changes to their ad campaigns. Medical physicians may subject their patients to more effective treatment protocols when supported by those suggestions. On the other hand, malicious entities having access to a model's internal structure might take advantage of our algorithm to carefully craft perturbed instances which induce an existing classifier to misclassification, i.e., to masquerade a spam message as non-spam. This kind of attack is the typical subject of interest of *adversarial learning* [7], which is a branch of machine learning (mostly, deep learning [8]) that studies how to design predictive models that are robust against manipulations of the input performed by an attacker whose aim is to coerce prediction errors [1], [9], [10], [11], [12]. Several approaches to adversarial learning treat ML models as black boxes; instead, we assume we can access the structure of the model we are considering. Furthermore, the focus of this work is on the *interpretability* and *actionability* of generated suggestions rather than their exposure to be used as adversarial examples, which we leave as future work.

We finally use the last domain (i.e., handwritten digit recognition) to assess the computational efficiency of our implementation based on space partitioning data structures.

Overall, with respect to our previous work [5], this paper contains the following additional contributions:

- We generalise our method to the case of multiclass classification problem (Section 2);
- We explain in more detail the algorithm we introduced, along with a thorough complexity analysis and the improvements we brought to make it more computationally-efficient when implemented in practice (Section 3);
- We further assess the validity of recommendations generated with our approach on two additional binary classification use cases, namely healthcare (Section 4.2) and spam filtering (Section 4.3);
- We measure the speedup obtained with our efficient implementation on a third additional use case, which represents an example of multiclass classification task, i.e., handwritten digit recognition (Section 5);
- We publicly release on GitHub[3] the source code of the implementation of the methodology proposed.

2 Problem Statement

In our previous work we focus on binary classification [5], whilst in this work we formulate our problem statement in the more general case of *multiclass* classification, i.e., K-ary classification ($K \geq 2$). Still, we consider ensembles of decision trees [6] as an effective solution to this problem, as also proved in many Kaggle competitions [13]. Additionally, we discuss how the internals of an existing ensemble of decision trees can be used to derive a feedback loop for recommending how a k-labelled predicted instance can be turned into a k'-labelled one, where both k and k' indicate one of the possible K class labels.

[3]https://github.com/gtolomei/ml-feature-tweaking

2.1 Notation

Let $\mathcal{X} \subseteq \mathbb{R}^n$ be an n-dimensional vector space of real-valued features. Any $\mathbf{x} \in \mathcal{X}$ is an n-dimensional feature vector, i.e., $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, representing an object in the vector space $\mathcal{X}$. Suppose that each $\mathbf{x}$ is associated with a K-ary *class label*, and let $\mathcal{Y} = \{1, \ldots, K\}$ be the set encoding all such possible class labels.

We assume there exists $f : \mathcal{X} \longmapsto \mathcal{Y}$ as an *unknown target* function that maps any feature vector to its corresponding class label. In addition, we let $\hat{f} \approx f$ which is learned from a labelled dataset of m i.i.d. instances $\mathcal{D} = \{(\mathbf{x_i}, y_i)\}_{i=1}^{m}$. More specifically, $\hat{f}$ is the estimate that best approximates f on $\mathcal{D}$, according to a specific *loss function* ℓ. Such a function measures the "cost" of prediction errors we would make if we replaced the true target f with the estimate $\hat{f}$. Eventually, learning $\hat{f}$ reduces to solving the following optimization problem (*empirical risk minimization*):

$$\hat{f} = \text{argmin}_{f^*} \ell(f^*, \mathcal{D}).$$

The flexibility vs. interpretability of the estimate $\hat{f}$ we learn depends on the assumptions we make on the family of functions (i.e., *hypothesis space*) which $\hat{f}$ has been picked from by the learning algorithm, and the loss function ℓ.

In this work, we focus on $\hat{f}$ represented as an *ensemble* of T *bootstrap aggregated* learners, i.e., *bagged decision trees*:

$$\hat{f} = \phi(\hat{h}_1, \ldots, \hat{h}_T)$$

where each $\hat{h}_t : \mathcal{X} \longmapsto \mathcal{Y}$ is a base decision tree classifier, and ϕ is the function responsible for combining the output of all individual base classifiers into a single prediction. Several combination rules can be used to implement ϕ [14]. Assuming such rules operate on class labels output by each base classifier, ϕ can reduce to a *majority voting* strategy, and the output of the ensemble on the generic input $\mathbf{x}$ be computed as follows:

$$\hat{f}(\mathbf{x}) = \arg\max_{y \in \mathcal{Y}} \sum_{t=1}^{T} \mathbf{1}_y(\hat{h}_t(\mathbf{x})) \tag{1}$$

where $\mathbf{1}_y(k)$ is an *indicator function* which evaluates to 1 iff $k = y$, 0 otherwise.

In the remainder of this work, we assume $\hat{f}$ is learned once for all, and never gets re-trained and updated; this is to guarantee that the prediction output by the model on an instance $\mathbf{x}$ never changes over time, i.e., $\hat{f}(\mathbf{x})$ is constant.

2.2 Enforcing Prediction Switch

The aim of this work is to identify how to transform an instance into *another* one, such that its original class label predicted by the ensemble turns into a different prediction. More formally, let $\mathbf{x} \in \mathcal{X}$ be a k-labelled *predicted* instance, such that $\hat{f}(\mathbf{x}) = k$. The task can now be defined as transforming the original input feature vector $\mathbf{x}$ into a new feature vector $\mathbf{x}'$ ($\mathbf{x} \rightsquigarrow \mathbf{x}'$), such that $\hat{f}(\mathbf{x}') = k' \neq k$. Moreover, we accomplish an optimised form of the problem by choosing $\mathbf{x}'$ as the best transformation among all the possible transformations $\mathbf{x}^*$, according to a *cost function* $\delta : \mathcal{X} \times \mathcal{X} \longmapsto \mathbb{R}$. This is defined as follows:

$$\mathbf{x}' = \arg\min_{\mathbf{x}^*} \left\{ \delta(\mathbf{x}, \mathbf{x}^*) \mid \hat{f}(\mathbf{x}) = k \wedge \hat{f}(\mathbf{x}^*) = k' \right\}$$

Intuitively, the cost function measures the "effort" of transforming $\mathbf{x}$ into $\mathbf{x}'$. For example, if $\delta(\mathbf{x}, \mathbf{x}') = ||\mathbf{x} - \mathbf{x}'||_0$ this will correspond to the number of features affected by the transformation. Alternatively, $\delta(\mathbf{x}, \mathbf{x}') = ||\mathbf{x} - \mathbf{x}'||_2$ will measure the Euclidean distance (also known as L^2-norm) between the original and the transformed vector.

More complex functions which takes into account the cost of transforming *each* individual feature can also be designed. This is to reflect all those situations – indeed very common in practice – where the value of some features may be relatively easy to change (e.g., increase the daily dosage of a drug), whilst for others it would rather be much harder, or even impossible (e.g., decreasing the weight or the age of an individual, respectively). Formally, this means introducing a set of functions $\{\delta_i\}_{i=1}^{n}$, where each $\delta_i : \mathbb{R}_i \times \mathbb{R}_i \longmapsto \mathbb{R}$ measures the cost of transforming the feature indexed by i, and $\mathbb{R}_i$ denotes its domain of values. Note that the definition above is still valid when the value of a feature x_i cannot be actually modified, as in such a case it will be $\delta_i(x_i, x_i') = +\infty$ for any $x_i' \neq x_i$. Overall, δ will be responsible for combining each individual δ_i into a single-valued cost, i.e., $\delta = \delta_1 \oplus \ldots \oplus \delta_n$, where $\oplus$ can be any aggregating function, such as the sum or the mean.

Finally, note that we do not make any assumptions on $f(\mathbf{x})$ and $f(\mathbf{x}')$, i.e., we are not demanding $f(\mathbf{x}) = k$ nor $f(\mathbf{x}') = k'$ since, in general, this can only be stated for training labelled instances. Still, if the learned model $\hat{f}$ is highly accurate we can also be quite confident that predicted labels very likely match with the true ones.

2.3 k-leaved Paths

Any root-to-leaf path of a single decision tree $\hat{h}_t$ in $\hat{f}$ can be interpreted as a cascade of *if-then-else* statements, where every branch node (i.e., non-leaf) is a boolean test on a specific feature value against a threshold. We restrict the tree decisions to be binary representations as any multiway decision can be represented in a binary form and there is little performance benefit in n-ary splits. An instance's feature value is then evaluated at each node to determine which branch to traverse. This is repeated until the leaves are reached, whereby one of the K classification labels (i.e., $k \in \{1, \ldots, K\}$) are defined and assigned.

In the following, we provide more formal definitions which will be used hereinafter in the paper.

Definition 1 (root-to-leaf path). Let $\hat{h}_t$ be a non-empty binary decision tree and $N_t \neq \emptyset$ denote the set of its nodes, where $r_t \in N_t$ is the root. Moreover, let $L_t \subseteq N_t$ be the set of all leaves of the tree (i.e., the set of all nodes with no children). Let us enumerate all the elements of L_t using a *pre-order depth-first traversal* of $\hat{h}_t$, so that we obtain a mapping $\lambda : L_t \mapsto \mathbb{N}$. Since $1 \leq |L_t| \leq \frac{|N_t|+1}{2}$, we let $\lambda : L_t \mapsto \{1, \ldots, \frac{|N_t|+1}{2}\}$. Thus, we define the sequence $p_{t,j} = (n_{t,1}, \ldots, n_{t,|p_{t,j}|})$ as the j-th root-to-leaf path of $\hat{h}_t$, such that:

- $n_{t,i} \in N_t,\ \forall i \in \{1, \ldots, |p_{t,j}|\}$;
- $n_{t,i+1}$ is the child of $n_{t,i},\ \forall i \in \{1, \ldots, |p_{t,j}| - 1\}$;
- $n_{t,1} = r_t$, i.e., the first node of the sequence is the root;
- $n_{t,|p_{t,j}|} \in L_t$ and $\lambda(n_{t,|p_{t,j}|}) = j$, i.e., the last node of the sequence is the j-th leaf.

Definition 2 (root-to-leaf path length). Given the j-th root-to-leaf path of $\hat{h}_t$ as $p_{t,j} = (n_{t,1}, \ldots, n_{t,|p_{t,j}|})$, we define its length as the number of its branch nodes, i.e., $|p_{t,j}| - 1$.

Note that the number of leaves of a tree is equal to the number of its root-to-leaf paths. Therefore, $P_t = \bigcup_{j=1}^{|L_t|} p_{t,j}$ is the set of all root-to-leaf paths of $\hat{h}_t$. Furthermore, the above two definitions are still valid when $\hat{h}_t$ has just one node and this is both the root and the only leaf, i.e., when $N_t = L_t = \{r_t\}$. Indeed, in this edge case there exists only one root-to-leaf path $p_{t,1} = (r_t)$, whose length is equal to $|p_{t,1}| - 1 = 0$.

In the following, we assume that the length of each path of a decision tree is at most n, i.e., $|p_{t,j}| - 1 \leq n$, which corresponds to n branch nodes (boolean conditions), one for each distinct feature.[4] This also means that each $\hat{h}_t$ is at most a depth-n binary tree.

It is worth noticing that, in general, we cannot ensure a bound to the depth of each tree of the ensemble. In practice though, we can specify such a bound at training time by capping the value of the hyperparameter of the model that regulates the maximum depth d of all trees to the number n of features, i.e., by setting $d = n$.

Definition 3 (k-leaved path). Let P_t be the set of all root-to-leaf paths of $\hat{h}_t$. A k-leaved path $p_t^k \in P_t$ is a root-to-leaf path that leads to a leaf node labelled as k.

Definition 4 (feature predicate). Let i be a feature identifier, and $\mathbb{R}_i \subseteq \mathbb{R}$ be the domain of values of i. A feature predicate is a function $\pi_i : \mathbb{R}_i \longmapsto \{\texttt{true}, \texttt{false}\}$. Each branch node of a decision tree encodes two possible feature predicates, i.e., either $\pi_i(x_i) = (x_i \leq \theta_i)$ or $\pi_i(x_i) = (x_i > \theta_i)$, with $\theta_i \in \mathbb{R}$.

Putting together all the definitions above, we characterise the encoding of each k-leaved path as follows.

Definition 5 (k-leaved path encoding). Let $p_{t,j}^k$ be the j-th root-to-leaf path of $\hat{h}_t$ ending up in a k-labelled leaf. We represent this with a boolean expression made of up to n clauses, as follows:

$$p_{t,j}^k = X_1 \wedge X_2 \wedge \ldots \wedge X_n \tag{2}$$

where each $X_i = \pi_i(x_i)$ is a boolean variable whose value is determined by the output of the predicate on the feature indexed by i.

Clearly, there exists only one assignment which satisfies a path, namely the one where *all* the clauses are satisfied ($X_i = \texttt{true},\ \forall i \in \{1, \ldots, n\}$). More generally, we provide the following definition.

Definition 6 (path-satisfactory instance). Let $p_{t,j}^k$ be the j-th k-leaved path of $\hat{h}_t$ encoded as specified in Definition 5. We denote by $\mathbf{x}_{t,j}^{\mathsf{SAT}}$ *any* input whose feature values satisfy all the predicates encoded by the path $p_{t,j}^k$, and we call it a *path-satisfactory instance* of $p_{t,j}^k$. As such, $\hat{h}_t(\mathbf{x}_{t,j}^{\mathsf{SAT}}) = k$.

[4] In general, there can be multiple boolean conditions associated with a single feature along the same path.

Of course, there can be several (possibly, infinitely many) path-satisfactory instances for the same path $p^k_{t,j}$. In the next section, we discuss how we select among those candidates.

So far, we have considered only a single k-leaved path of one decision tree $\hat{h}_t$; however, in general, the same tree may contain multiple k-leaved paths. We denote by $P^k_t \subseteq P_t$ the set of all the k-leaved paths in $\hat{h}_t$, and $P^k = \bigcup_{t=1}^{T} P^k_t$ the set of all the k-leaved paths in the ensemble $\hat{f}$. Note that we can encode P^k as a disjunctive normal form (DNF) boolean expression.

2.4 Tweaking Input Features

Given our input feature vector $\mathbf{x}$, we know from our hypothesis that $\hat{f}(\mathbf{x}) = k$. If the overall prediction output by the ensemble is obtained using a majority voting strategy as described in Equation 1, it follows that:

$$\hat{f}(\mathbf{x}) = k \iff k = \arg\max_{y \in \mathcal{Y}} \sum_{t=1}^{T} \mathbf{1}_y(\hat{h}_t(\mathbf{x}))$$

Therefore, if we denote by $T^{(\mathbf{x},k)} = \bigcup_{t=1}^{T} \{t \mid \hat{h}_t(\mathbf{x}) = k\}$ the set of indices of base decision trees which output k when input with $\mathbf{x}$, it must hold that:

$$|T^{(\mathbf{x},k)}| \geq |T^{(\mathbf{x},k')}| \quad \forall k' \neq k$$

Our goal is to tweak the original input feature vector $\mathbf{x}$ so as to adjust the prediction made by the ensemble from k to k'. To achieve this, we operate as follows. For each tree $\hat{h}_t$ of the ensemble, we consider the set $P^{k'}_t$ of all its k'-leaved paths. Then, with each $p^{k'}_{t,j} \in P^{k'}_t$ we associate a crafted path-satisfactory instance $\mathbf{x}^{\mathsf{SAT}}_{t,j} = \mathbf{x}'_{t,j} \in \mathcal{X}$ that *satisfies* that path – i.e., an instance whose adjusted feature values satisfies the boolean expression encoded in $p^{k'}_{t,j}$, therefore finishing on a k'-labelled leaf, so that $\hat{h}_t(\mathbf{x}'_{t,j}) = k'$.

Among all the possibly infinite instances satisfying $p^{k'}_{t,j}$, we restrict to $\mathbf{x}'_{t,j(\boldsymbol{\epsilon})}$ to be feature value changes with a "tolerance" bounded by $\boldsymbol{\epsilon}$: we call this the $\boldsymbol{\epsilon}$-satisfactory instance of $p^{k'}_{t,j}$. Without loss of generality, we let $\boldsymbol{\epsilon} \in \mathbb{R}^n_{>0}$, i.e., an n-dimensional vector of positive thresholds, one for each feature.

We consider $p^{k'}_{t,j}$ containing at most n boolean conditions, as specified by Equation 2. Therefore, for any fixed $\boldsymbol{\epsilon}$ we define $\mathbf{x}'_{t,j(\boldsymbol{\epsilon})}$ as follows:

$$\mathbf{x}'_{t,j(\boldsymbol{\epsilon})}[i] = \begin{cases} \theta_i - \epsilon_i & \text{if the } i\text{-th condition is } (x_i \leq \theta_i) \\ \theta_i + \epsilon_i & \text{if the } i\text{-th condition is } (x_i > \theta_i) \end{cases} \tag{3}$$

It is worth noticing that, in general, different features may have different domains, and therefore different scales. In case we want to operate with a single, global tolerance – i.e., with a scalar $\epsilon \in \mathbb{R}_{>0}$ instead of the vector $\boldsymbol{\epsilon}$ – we can standardise/normalise features in advance.

By applying Equation 3 to each $p^{k'}_{t,j} \in P^{k'}_t$, we obtain a set of $\boldsymbol{\epsilon}$-satisfactory transformations $\Gamma^{k'}_t = \bigcup_{j \in P^{k'}_t} \mathbf{x}'_{t,j(\boldsymbol{\epsilon})}$ associated with the t-th tree $\hat{h}_t$, and overall $\Gamma^{k'} = \bigcup_{t=1}^{T} \Gamma^{k'}_t$ the set of *all* the $\boldsymbol{\epsilon}$-satisfactory transformations derived from all the trees of the ensemble.

Interestingly, the set of $\boldsymbol{\epsilon}$-satisfactory transformations $\Gamma^{k'}$ does *not* depend on the input feature vector $\mathbf{x}$; in fact, $\Gamma^{k'}$ can be statically computed once for all as soon as we learn the ensemble $\hat{f}$, and therefore once the set $P^{k'}$ is known.

Anyway, not every transformation in $\Gamma^{k'}$ is valid to achieve our goal. Indeed, if we pick a transformation $\mathbf{x}' = \mathbf{x}'_{t,j(\boldsymbol{\epsilon})} \in \Gamma^{k'}$ induced by a single k'-leaved path of a specific tree $\hat{h}_t$ this may have an impact on other trees of the ensemble. As such, we cannot simply replace our original input $\mathbf{x}$ with $\mathbf{x}'$, since there might exist $l \in T^{(\mathbf{x},k')}$ whose corresponding tree by definition already provides the correct prediction when this is input with $\mathbf{x}$, $\hat{h}_l(\mathbf{x}) = k'$, yet changing $\mathbf{x}$ into $\mathbf{x}'$ leads to $\hat{h}_l(\mathbf{x}') = k \neq k'$. In other words, by changing $\mathbf{x}$ into $\mathbf{x}'$ we are only guaranteed that the prediction of the t-th base classifier is correctly fixed, i.e., from $\hat{h}_t(\mathbf{x}) = k$ to $\hat{h}_t(\mathbf{x}') = \hat{h}_t(\mathbf{x}'_{t,j(\boldsymbol{\epsilon})}) = k'$. The overall prediction for $\mathbf{x}'$ may or may not be fixed, whereby $\hat{f}(\mathbf{x}')$ may still output k, exactly as $\hat{f}(\mathbf{x})$ did.

If the change from $\mathbf{x}$ to $\mathbf{x}'$ also leads to $\hat{f}(\mathbf{x}') = k'$, then $\mathbf{x}'$ will be a *candidate* transformation for $\mathbf{x}$. More formally, we define the set of candidate transformations with respect to any target class as follows.

Definition 7 (k-labelled $\boldsymbol{\epsilon}$-satisfactory candidates). Let $\hat{f}$ be an ensemble of T base classifiers $\{\hat{h}_t\}_{t=1}^{T}$, $k \in \mathcal{Y}$ be a target class label and Γ^k be the set of possible $\boldsymbol{\epsilon}$-satisfactory transformations derived from the set P^k of all the k-leaved paths of the ensemble. Then:

$$\Gamma^k_c = \{\mathbf{x}' \mid \mathbf{x}' \in \Gamma^k \wedge \hat{f}(\mathbf{x}') = k\}$$

is the set of k-labelled, $\boldsymbol{\epsilon}$-satisfactory candidate transformations.

Our feature tweaking problem can thus be generally defined as follows.

Definition 8 (Feature Tweaking Problem). Let $\hat{f}$ be an ensemble of T base classifiers $\{\hat{h}_t\}_{t=1}^{T}$, $\mathbf{x}$ an instance such that $\hat{f}(\mathbf{x}) = k$, $k' \in \mathcal{Y}$, $k' \neq k$ a target class label, and $\delta : \mathcal{X} \times \mathcal{X} \longmapsto \mathbb{R}$ a *cost function*.
The feature tweaking problem aims to find another instance $\mathbf{x}'$ among all the possible k'-labelled, $\boldsymbol{\epsilon}$-satisfactory candidates $\Gamma^{k'}_c$, so that $\delta(\mathbf{x}, \mathbf{x}')$ is minimum:

$$\mathbf{x}' = \arg\min_{\mathbf{x}^* \in \Gamma^{k'}_c} \left\{ \delta(\mathbf{x}, \mathbf{x}^*) \right\} \tag{4}$$

In [15], it has already been proven that a problem similar to the one we define above is NP-hard, as it reduces to DNF-MAXSAT. Our version, in fact, introduces an additional constraint ($\boldsymbol{\epsilon}$) which further restricts the possible way features can be tweaked, and thus it is itself NP-hard.

The problem as described in Definition 8 is still valid for the base case when $T = 1$; there, in particular, $\Gamma^k_c = \Gamma^k$ (for all k). Indeed, in that scenario the ensemble is composed of a single base classifier ($\hat{f} = \hat{h}$) – i.e., the ensemble contains a single decision tree and tweaking its prediction also results in changing the overall prediction. Note that when there is only one decision tree, our problem can be solved optimally: we can enumerate all the k'-leaved paths, choose the one with the minimum cost, and check if the threshold of tolerance $\boldsymbol{\epsilon}$ is satisfied. Because base trees are interconnected through the features they share, simply enumerating k'-leaved paths does not work for an ensemble

of trees, since the output of a base tree may affect outputs of its sibling trees. This motivates why we need to further restrict the set of all ϵ-satisfactory transformations to the subset of candidates, as discussed in Definition 7. Still, it is worth noticing that there might be cases where the feature tweaking problem has no solution. For example, consider the case of a binary classification task where the majority of decision trees in the ensemble do not have *any* root-to-leaf path ending up in a leaf labelled with the target class. In such an extreme scenario, the set of candidate transformations will be empty as no perturbation of the original instance will ever enforce the ensemble to change its prediction. However, we believe that this is a very peculiar case as it would shape up a situation where the ensemble is seriously compromised and flawed in the first place (i.e., if the majority of decision trees have *all* their paths leading to leaves with the very same class label, the model will always predict the same class, no matter what the input is).

3 The Feature Tweaking Algorithm

In this section, we describe the intuition behind our algorithm to solve the problem defined in Equation 4. More precisely, we present the naïve solution introduced in our previous work [5], and a more efficient solution we have later designed, along with its pseudo-code implementation.

At query time, i.e., online, we are given an input feature vector $\mathbf{x}$ we want to tweak, a target class label $k' \in \mathcal{Y}$, and the set of k'-labelled ϵ-satisfactory candidates $\Gamma_c^{k'}$. Our problem thereby reduces to a *nearest neighbour search* using the function δ.

So far, we have not made any assumption on the cost function δ, i.e., we did not specify whether it is a distance metric or not. However, plausible definitions for it may be indeed distance metrics, such as: $\delta = \mathrm{L}^0$-norm (Hamming distance), $\delta = \mathrm{L}^1$-norm (Manhattan distance), or $\delta = \mathrm{L}^2$-norm (Euclidean distance), just to name a few.

Furthermore, we have already observed that the set of ϵ-satisfactory candidates Γ_c^k (for all $k \in \mathcal{Y}$) does not depend on the input feature vector $\mathbf{x}$ we may ultimately want to perturb, yet it can be computed offline from the k-leaved paths of the ensemble, once this is learned from data. We will see how this can be exploited to come up with a more efficient version of our original, naïve algorithm.

3.1 Naïve Solution

The simplest solution to the feature tweaking problem is proposed in our previous work [5], and consists of performing a linear search, namely computing the function δ from the query instance $\mathbf{x}$ to every other instance $\mathbf{x}' \in \Gamma_c^{k'}$, and keeping track of the "best so far". This solution has a running time complexity of $O(nm)$, where n is the dimensionality of $\mathcal{X}$ (i.e., the feature space) and $m = |\Gamma_c^{k'}|$ is the cardinality of the set of candidates.

It is worth noticing that m is related to n, as follows. The size of valid k'-labelled candidates are bound to the number of k'-leaved paths in the ensemble, i.e., $m = |\Gamma_c^{k'}| \leq |P^{k'}|$. In turn, $|P^{k'}|$ is bound to the sum of k'-leaved paths of all trees, i.e., $|P^{k'}| \leq \sum_{t=1}^{T} |P_t^{k'}|$. According to Definition 2, each $p_{t,j}^k$ of a single decision tree is *by design* at most a length-n path then $\hat{h}_t$ is a depth-n binary tree, whose number of leaves is therefore bounded to 2^n. As the total number of leaves coincides with the total number of paths, we can also state that $|P_t^{k'}| \leq 2^n, \ \forall k', t$. Overall, $m \leq \sum_{t=1}^{T} |P_t^{k'}| \leq 2^n T$. Assuming $2^n \gg T$[5], we can express the worst case time complexity of the naïve solution in terms of the number n of features, which is $O(2^n)$.

3.2 Space-Partitioning Solution

One of the novel contributions of this work is a solution to the feature tweaking problem which is more efficient than the one already proposed in [5] and described above. At the core of this new algorithm is the usage of a *space-partitioning* strategy to perform the nearest neighbour search. More precisely, our solution makes use of a *spatial index*, which arranges data (i.e., candidate ϵ-transformations) in a tree-like structure that allows discarding branches at once if they do not meet the specific search criteria. Possible examples of such indices are: R-trees [16], k-d trees [17], and ball trees [18]. They differ from each other by the way in which they recursively split data at each tree node: R-trees sort data into hyper-rectangles; k-d trees divide data into two halves around a median point; and ball trees partition data into hyper-spheres. The advantage of using one of the spatial data structures above is that this would generally allow us to lower the time complexity of the nearest neighbour search from linear (as in the case of the naïve solution) to logarithmic in the size of candidates, i.e., from $O(nm)$ to $O(n \log m)$[6]. Of course, this comes at the cost of processing data into a spatial index first, but this is a fair price to pay in exchange for quicker searches, since data changes are usually much less frequent than queries.

Our approach therefore consists of two steps: an offline step to build the spatial index from the set of all possible k-labelled ϵ-satisfactory candidates induced by the ensemble; and an online step which takes as input a feature vector (i.e., a query) and returns the best candidate transformation of it according to the cost function δ. In the following, we discuss the two steps above separately.

3.2.1 Offline step: building spatial index from candidates

This step takes as input a learned ensemble $\hat{f}$ and produces as output a set of spatial indices, i.e., one for each collection of k-labelled candidates $\Gamma_c^k, \forall k \in \mathcal{Y}$. The details of this step are described in Algorithm 1.

The algorithm starts examining each individual class label $k \in \{1, \ldots, K\}$ (outer loop, line 3); for each label, it computes the set of corresponding k-labelled candidates, which initially is empty (line 4). It therefore considers all the trees of the ensemble (inner loop, line 5), and for each tree it retrieves the set of its k-leaved paths P_t^k using the subprocedure GetKLeavedPaths (line 6). Then, for each k-leaved path it crafts the corresponding ϵ-satisfactory instance $\mathbf{x}'_{t,j(\epsilon)}$, i.e., an instance that satisfies that path according to Equation 3 and implemented by subprocedure

[5] This is indeed reasonable in practice, if we consider that with $n = 30$ features, $2^n = O(10^9)$, whilst large ensembles typically contain $T = O(10^3 \div 10^4)$ trees.

[6] The actual time complexity may depend on several factors, which in turn influence the choice of the spatial index to use.

Input:
▷ An estimate function $\hat{f} \approx f : \mathcal{X} \longmapsto \mathcal{Y}$, such that $\mathcal{X} \subseteq \mathbb{R}^n$ and $\mathcal{Y} = \{1, \ldots, K\}$. Moreover, $\hat{f}$ results from an ensemble of T depth-n decision trees, each one associated with a base estimate $\hat{h}_t,\ t = 1, \ldots, T$
▷ A threshold vector $\boldsymbol{\epsilon} \in \mathbb{R}^n_{>0}$
Output:
▷ A set $\mathcal{I} = \{I^1, \ldots, I^K\}$, where each I^k $(k = 1 \ldots K)$ is a spatial index built from the set of k-labelled $\boldsymbol{\epsilon}$-satisfactory candidates Γ^k_c

1: **procedure** BUILDSPATIALINDICES($\hat{f}, \boldsymbol{\epsilon}$)
2: $\mathcal{I} = \{\}$ *// Initialise the collection of spatial indices*
3: **for** $k = 1, \ldots, K$ **do**
4: $\Gamma^k_c = \emptyset$ *// Initialise the set of k-labelled candidates*
5: **for** $t = 1, \ldots, T$ **do**
6: $P^k_t \longleftarrow$ GETKLEAVEDPATHS($\hat{h}_t$, k)
7: **for all** $p^k_{t,j} \in P^k_t$ **do**
8: $\mathbf{x}'_{t,j(\boldsymbol{\epsilon})} \longleftarrow$ BUILDKLABELLEDINSTANCE($p^k_{t,j}, \boldsymbol{\epsilon}$)
9: **if** $\hat{f}(\mathbf{x}'_{t,j(\boldsymbol{\epsilon})}) == \hat{h}_t(\mathbf{x}'_{t,j(\boldsymbol{\epsilon})})$ **then**
10: $\Gamma^k_c = \Gamma^k_c \cup (\mathbf{x}'_{t,j(\boldsymbol{\epsilon})})$
11: **end if**
12: **end for**
13: **end for**
14: $I^k \longleftarrow$ BUILDSPATIALINDEX(Γ^k_c) *// Generate spatial index from data points in* Γ^k_c
15: $\mathcal{I}[k] \longleftarrow I^k$ *// Update the collection of spatial indices*
16: **end for**
17: **return** $\mathcal{I}$ *// Return all the spatial indices*
18: **end procedure**

Fig. 1: The procedure used to build the set of spatial indices from *all* the candidates Γ^k_c $(k \in \{1, \ldots, K\})$.

BUILDKLABELLEDINSTANCE. Before $\mathbf{x}'_{t,j(\boldsymbol{\epsilon})}$ can be added to the set of candidates Γ^k_c, the overall prediction of the ensemble on it must be equal to k (lines 9 and 10).

Once all the trees of the ensemble have been examined (for a class label k), the algorithm uses the subprocedure BUILDSPATIALINDEX for creating the spatial index data structure from the set of candidate data points in Γ^k_c (line 14), and consequently updates the collection of spatial indices (line 15). Note that, although very unlikely, Γ^k_c in general might be empty (i.e., there might be no valid candidates), as discussed at the end of Section 2.4; when this is the case, the corresponding spatial index I^k will degenerate to an empty data structure as well. Eventually, when all the class labels have been considered the final set $\mathcal{I}$ is returned.

The procedure described above examines *all* the paths of all the trees in the input ensemble. Therefore, the total number of steps is equal to $2^n T$, since each depth-n tree has 2^n paths and there are T trees in the ensemble. The subprocedure BUILDSPATIALINDEX, instead, can be accomplished in $O(nm \log m)$, where m is the size of candidates to be indexed (i.e., Γ^k_c). Overall, the running time complexity of Algorithm 1 is $O(2^n)$, again assuming 2^n dominates over T. In practice, however, the average path length of each tree may be significantly smaller than n. In fact, a well-known approach used in many ensemble learning settings is the *random subspace method* [19], which attempts to reduce the correlation between sibling trees by setting their maximum depth $d = O(\sqrt{n})$, thereby obtaining sub-exponential time complexity $2^{O(\sqrt{n})}$.

3.2.2 Online step: tweaking input query vector

The online step takes as input 4 key components:

- A feature vector $\mathbf{x}$ (query) that represents a k-labelled predicted instance, i.e., $\hat{f}(\mathbf{x}) = k$;
- A target class label, $k' \in \mathcal{Y}, k' \neq k$;
- A cost function δ measuring the "effort" required to transform the true k-labelled instance into a k'-labelled one (i.e., a distance function);
- The collection of spatial indices $\mathcal{I}$ built using the procedure described in Algorithm 1.

The result being the transformation $\mathbf{x}'$ of the original $\mathbf{x}$ that exhibits the minimum cost according to δ, such that $\hat{f}(\mathbf{x}') = k' \neq k = \hat{f}(\mathbf{x})$. The detailed description is presented in Algorithm 2.

Input:
▷ A feature vector $\mathbf{x}$ (query) representing a *true* k-labelled instance, such that $f(\mathbf{x}) = \hat{f}(\mathbf{x}) = k$
▷ A target class label $k' \in \mathcal{Y} = \{1, \ldots, K\},\ k' \neq k$
▷ A cost function δ
▷ A collection of spatial indices $\mathcal{I}$
Output:
▷ The optimal transformation $\mathbf{x}'$ with respect to δ, such that $\hat{f}(\mathbf{x}') = k'$

1: **procedure** TWEAKINGFEATURES($\mathbf{x}, k', \delta, \mathcal{I}$)
2: $\mathbf{x}' \longleftarrow$ NULL
3: $I^{k'} \longleftarrow \mathcal{I}[k']$ *// Retrieve the spatial index built from the set of candidate data points* $\Gamma^{k'}_c$
4: $\mathbf{x}' \longleftarrow$ NEARESTNEIGHBOURSEARCH($\mathbf{x}$, $I^{k'}$, δ)
5: **return** $\mathbf{x}'$ *// The optimal feature vector transformation*
6: **end procedure**

Fig. 2: The FEATURE TWEAKING ALGORITHM takes advantage of spatial indices to search for the nearest neighbour perturbation of the input $\mathbf{x}$ (i.e., $\mathbf{x}'$), efficiently.

The algorithm operates as follows. First of all, it retrieves the spatial index built from the candidate points whose label is the same of the target, i.e., k'-labelled candidates (line 2). Then, it simply delegates off to the subprocedure NEARESTNEIGHBOURSEARCH (line 3), which takes as input the query feature vector, the spatial index, and the cost function δ and returns the data point among those indexed by the spatial index that is the closest to the query $\mathbf{x}$. Finally, it returns $\mathbf{x}'$ as the nearest neighbour to $\mathbf{x}$ (if it exists), or NULL if the spatial index is empty.

The computational time complexity of Algorithm 2 coincides with that of NEARESTNEIGHBOURSEARCH (line 4), since the access to the k'-th spatial index (line 3) can be easily performed in $O(1)$. Although spatial indices do not guarantee good worst-case performance, they generally perform well on average [20], [21], and are able to lower the nearest neighbour search complexity to $O(n \log m)$, where m is the number of n-dimensional indexed data points. More recently, however, Priority R-tree has been proposed as worst-case optimal variant of R-tree [22]. In addition, the computation of nearest neighbours has proven efficient in combination with distance metrics based on any L^p-norm [23], [24]. However, for distances based on L^0-norm (Hamming distance) another solution specifically tailored to discrete metric spaces, i.e., BK-tree, works better [25].

4 Evaluating Recommended Tweaks

We demonstrate the utility of our method when applied to three use cases. First, in Section 4.1 we show how our algorithm can be used to solve a real-world business problem, i.e., to improve the quality of advertisements served by the Yahoo Gemini[7] ad network. This has been already thoroughly discussed in our previous work [5]; we invite the reader to refer to that paper for a more detailed description.

As novel contribution of this work, we further validate our method on two additional domains: healthcare and spam filtering (Section 4.2 and 4.3, respectively). We choose those as they both relate to highly impactful scenarios, with a clear application to real-life situations.

4.1 Online Advertising

The success of online advertising is highly sensitive to the quality of advertisements (ads, for short) that third-party ad networks deliver to users via web/app publishers.

Many factors can affect the quality of an ad: its *relevance*, i.e., whether the ad matches the user interest [26]; the *pre-click* experience, i.e., whether the ad annoys a user [27]; and finally the *post-click* experience, i.e., whether the *ad landing page*[8] meets the user click intent that brought them to the landing page [28]. We focus on the latter, the post-click experience, following from [28], [29].

We know from [28] that ad landing pages exhibiting long dwell times[9] promote a positive long-term post-click experience. On top of that, we can design an ad quality prediction model, namely a binary classifier that leverages ad features to effectively separates between *low* and *high* quality ads, i.e., ad landing pages whose dwell time is below or above a threshold τ, respectively.

As one of the biggest player in the online advertising business, Yahoo Gemini – i.e., the newly-integrated Yahoo's ad serving platform – has a varying distribution of quality with many ads being of low quality. Not serving them may not be an option when supply is exhausted. Therefore, another approach to positively shift the quality distribution of the ad inventory is to leverage the interpretability of the internal machinery of existing ad quality prediction models, so as to offer actionable *recommendations*. The intention of these programmatically-computed recommendations is to provide advertisers with guidance on how they can improve their ad quality at scale. Such a system yields value for all beneficiaries in the advertising ecosystem, ultimately culminating with a better user experience.

In the remaining of this section, we demonstrate that our feature tweaking algorithm introduced in Section 3 is able to achieve the task above by generating helpful suggestions on how to transform a low quality ad into a set of new "proposed" high quality ads.

4.1.1 Predicting Ad Quality

To apply our algorithm, we first need to learn a binary classifier that predicts whether an ad is of high quality or not, given a feature-based representation of each ad: textual content of the creative, number of images in the landing page, similarity between the creative and the landing page, etc. Ad feature engineering has been extensively investigated in our previous work, and we suggest the reader to refer to [5] for the complete list of 45 ad features extracted.

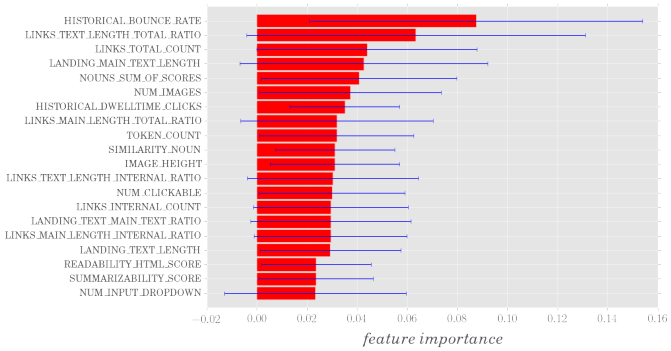


Fig. 3: Top-20 most important features of our RF model.

As our feature tweaking algorithm is designed to work on any ensemble of bagged decision trees, we train the following learning models to find our best estimate $\hat{f}$: Decision Trees (DT) [30] – which can be thought of as a special case of an ensemble with a single tree – and Random Forests (RF) [31].

We collect a random sample of 1,500 ads served by Yahoo Gemini on a mobile app during one month, which are labelled as low and high quality, using an estimate of the threshold of dwell time ($\tau \approx 62.5$ seconds).

This original dataset $\mathcal{D}$ is split into two datasets $\mathcal{D}_{\text{train}}$ and $\mathcal{D}_{\text{test}}$ using stratified random sampling. $\mathcal{D}_{\text{train}}$ is used for training the models and accounts for 80% of the total number of instances in $\mathcal{D}$, whilst $\mathcal{D}_{\text{test}}$ contains the remaining held-out portion used for evaluating the models. $\mathcal{D}_{\text{train}}$ is also used to perform model selection, which is achieved by tuning the hyperparameters specific for each.

With every combination of model and corresponding hyperparameters, we run a 10-fold cross validation to find the best settings for each model – i.e., the one with the best cross validation ROC AUC (RF = **0.93** and DT = 0.84). Each model is in turn re-trained on the whole $\mathcal{D}_{\text{train}}$ using the best hyperparameter setting. Finally, the overall best model is considered the one achieving the highest F_1 on the test set $\mathcal{D}_{\text{test}}$ (RF = **0.84** and DT = 0.75), which turns out to be RF.

4.1.2 Ad Feature Recommendations

We validate the recommendations generated with our approach, by applying our feature tweaking algorithm to our learned RF model. Any $\mathbf{x}'$ that results from a valid (i.e., positive) ϵ-transformation[10] of the original negative instance $\mathbf{x}$ encapsulates a set of directives on how to positively change the ad features. We compute the vector $\mathbf{r}$ resulting from the component-wise difference between $\mathbf{x}'$ and $\mathbf{x}$, which is $\mathbf{r}[i] = \mathbf{x}'[i] - \mathbf{x}[i]$. Then for each feature indexed by i, such that $\mathbf{r}[i] \neq 0$ (i.e., $\mathbf{x}'[i] \neq \mathbf{x}[i]$), this vector provides the *magnitude* and the *direction* of the changes that should be made on feature i. The magnitude denotes the absolute value of the change (i.e., $|\mathbf{x}'[i] - \mathbf{x}[i]|$), whilst the direction

[7] https://gemini.yahoo.com

[8] We refer to ad landing page as the web page of the advertiser that a user is redirected to after clicking on an ad.

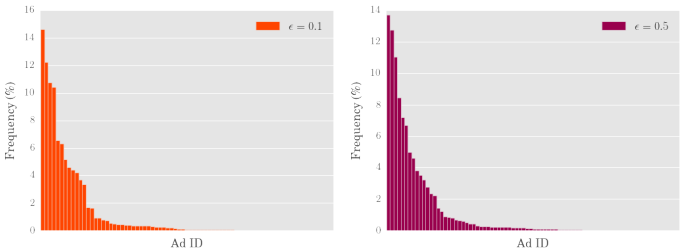
[9] The *dwell time* is the time spent by a user on a web page.

[10] Note that here ϵ is a scalar instead of a vector $\boldsymbol{\epsilon}$, as features have been standardised in advance like discussed in Section 2.4.

indicates whether this is an *increase* or a *decrease* of the original value of feature i (i.e., $\text{sgn}(\mathbf{x}'[i] - \mathbf{x}[i])$). Finally, to derive the final list of recommendations, we sort $\mathbf{r}$ according to the feature ranking, as shown in Figure 3.

Our approach depends on a *tweaking cost* (δ) associated with transforming a negative instance (low quality ad) into a positive instance (high quality ad), and a (global) *tweaking tolerance* (ϵ) used to change each individual ad feature. We first explore how ϵ impacts on the ad *coverage*, which is the percentage of ads for which our approach is able to provide recommendations. We experiment with five values of ϵ: 0.01, 0.05, 0.1, 0.5, and 1. These values can be thought of as multiples of a unit of standard deviation from each individual feature mean, assuming features are standardised using their z-scores, as discussed in Section 2.4. Ad coverage increases from 58.5% when $\epsilon = 0.01$ up to its highest value of 77.4% when $\epsilon = 0.5$; then, it starts decreasing as ϵ approaches to 1.

Although some low quality ads cannot be transformed, those that can are often associated with multiple transformations. Figure 4a and 4b show the distribution of ϵ-transformations across the set of ads, generated when $\epsilon = 0.1$ and $\epsilon = 0.5$, respectively[11]. Both distributions



(a) (b)

Fig. 4: Distribution of ad ϵ-transformations at $\epsilon = 0.1$ (a) and $\epsilon = 0.5$ (b).

are skewed, offering a high number of transformations proposed for few ads. Interestingly, the number of transformations is more evenly distributed across the ads when ϵ increases. This is in agreement with the finding above, where larger values of ϵ result in a higher coverage before decreasing again between $\epsilon = 0.5$ and 1.

To choose the most appropriate transformation for an ad, we experiment with several tweaking cost functions δ, each taking as input the original ($\mathbf{x}$) and the transformed ($\mathbf{x}'$) feature vectors:

- *tweaked_feature_rate*: proportion of features affected by the transformation of $\mathbf{x}$ into $\mathbf{x}'$ (range = [0, 1]);
- *euclidean_distance*: Euclidean distance (L^2-norm) between $\mathbf{x}$ and $\mathbf{x}'$ (range = $\bar{R}_{\geq 0}$);
- *cosine_distance*: 1 minus the cosine of the angle between $\mathbf{x}$ and $\mathbf{x}'$ (range = [0, 2]);
- *jaccard_distance*: one's complement of the Jaccard similarity between $\mathbf{x}$ and $\mathbf{x}'$ (range = [0, 1]);
- *pearson_correlation_distance*: 1 minus the Pearson's correlation between $\mathbf{x}$ and $\mathbf{x}'$ (range = [0, 2]).

[11]Similar behaviour is observed for all values of ϵ.

4.1.3 Assessing Recommendations

To evaluate our method, we asked an internal team of Yahoo's creative strategists (CS)[12] to validate the recommendations generated by our approach ($\epsilon = 0.5$, $\delta =$ *cosine_distance*) on a set of 100 low quality ad landing pages, i.e., the true negative instances in $\mathcal{D}_{\text{test}}$. Each CS was assigned a set of ad landing pages with the corresponding ϵ-transformations, and additional metadata useful for assessing the recommendations within each transformation. The same set of ad landing pages – and therefore the same list of recommendations – was assessed by two CSs, who were asked to rate each recommendation as *helpful*, *non-helpful*, or *non-actionable*. A recommendation is deemed *helpful* when it is likely to help the advertiser to improve the user experience of the ad, and *non-helpful* otherwise. A *non-actionable* recommendation is one that cannot be practically implemented. Whenever a disagreement occurred, a third CS was called to resolve the conflict.

Overall, 57.3% of all the generated recommendations are rated helpful with an inter-agreement rate of 60.4% and only 0.4% result in a non-actionable suggestion. We also look at the 42.3% non-helpful recommendations, and saw that about 25% can be considered "neutral"; that is, they would not hurt the user experience if discarded as well as not adding any positive value if implemented.

Non-helpful tweaks might occur due to two reasons. First, the learned model we leverage for generating feature recommendations – no matter how accurate it is – is not perfect. Therefore, a negatively-predicted instance that is transformed into a positively-predicted one does not necessarily mean it is *actually* positive (nor even that it was truly negative in the first place). Second, tuning the hyperparameters (δ and ϵ) of our algorithm affects the set of candidate transformations. Limiting non-helpful tweaks can be achieved by improving the accuracy of the learned model and choosing values of the hyperparameters so as to minimise prediction errors.

Moreover, when we further look into the non-actionable recommendations we see that these are related to the features ADULT_SCORE and NUM_INPUT_DROPDOWN. Our algorithm suggests to decrease the value of those features; however, the ad landing pages do not contain adult words nor drop-downs. Most likely, the ad copies and landing pages used to generate recommendations have changed before the CSs performed their assessment.

Finally, we measure the "helpfulness" of each feature recommendation as follows:

$$helpfulness(i) = \frac{|helpful(i)|}{|helpful(i)| + |\neg helpful(i)|}$$

This computes the relative frequency of recommendations for feature i as being described as helpful by the CS team. In Figure 5, we report the ranked list of features involved in the top-10 most helpfulness recommendations. A similar ranking will be obtained if we weight the helpfulness score on the basis of the *overall* relative recommendation frequency. The majority of the most helpful recommendations were features extracted from the HTML DOM structure and content

[12]Creative strategists work with advertisers' web masters on strategic choices to help them developing effective advertising messages.

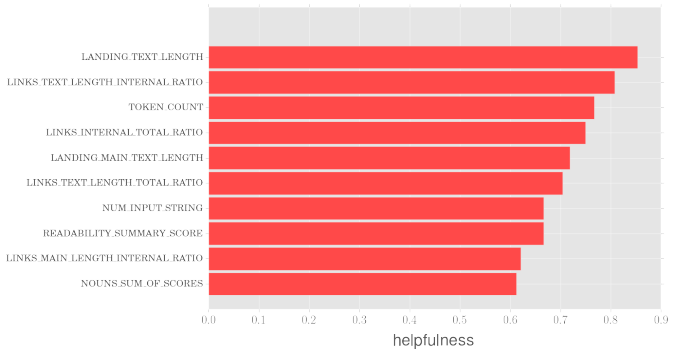


Fig. 5: Top-10 most helpful feature recommendations according to the helpfulness score.

of the ad landing page, indicating that high quality landing pages should exhibit a good balance between textual content and hyperlinks. Remarkably, those are also among the most predictive features according to our RF model (Figure 3).

4.2 Healthcare

We extend our set of experiments by first considering a public dataset of medical records available on Kaggle[13]. This dataset is originally from the National Institute of Diabetes and Digestive and Kidney Diseases. The initial reason for this dataset to be collected was to diagnostically predict whether or not a patient has diabetes, based on certain diagnostic measurements included in the medical records [32]. Several constraints were placed on the selection of these instances from a larger database. In particular, all patients here are females at least 21 years old of Pima Indian heritage (i.e., a group of Native Americans living in an area today known as central and southern Arizona).

The dataset $\mathcal{D}$ consists of 768 instances, each one represented by 8 medical predictor variables (i.e., features) and one target binary variable, which is whether the patient has diabetes (indicated by 1, or positive) or not (indicated by 0, or negative). There is a moderate class imbalance, with approximately 65% of instances being negative and the remaining 35% positive. The set of features (all numeric) are as follows:

- PREGNANCY: Number of times the patient has been pregnant.
- GLUCOSE: Plasma glucose concentration at 2 hours in an oral glucose tolerance test (mg/dl). For a 2-hour test with 75g intake, a value between 140 and 200 mg/dL (7.8 and 11.1 mmol/L) is called impaired glucose tolerance, and is known as "pre-diabetes"; it means the patient is at increased risk of developing diabetes over time. A glucose level of 200 mg/dL (11.1 mmol/L) or higher is used to diagnose diabetes.
- BLOOD_PRESSURE: Diastolic blood pressure (mmHg); if this is above 90 mmHg there is usually a high probability of diabetes, whereas if it is below 60 mmHg it means there is generally a lower probability of diabetes.
- SKIN_THICKNESS: Triceps skin fold thickness (mm) – i.e., a value used as proxy of body fat – which is usually around 23 mm in women. Higher figures may lead to obesity and, in turn, chances of diabetes may also increase.
- INSULIN: 2-hour serum insulin (mcIU/ml). Normal insulin level ranges between 16 and 166; any value above can be alarming.
- BMI: Body mass index computed as (weight in kg)/(height in m)^2. Normal BMI ranges between 18.5 and 25, whilst values between 25 and 30 indicate the patient is overweighted. A BMI of 30 or over falls within the obese range.
- DIABETES_PEDIGREE_FUNCTION: It provides information about diabetes history in relatives and genetic relationship of those relatives with the patient. Higher values of this feature indicate patient is more likely to have diabetes.
- AGE: Age of the patient, expressed in years.

In this use case, we want to show that our method is able to suggest which features should be perturbed (and how) in order to turn a patient with diabetes (true positive instance) into a healthy one (negatively predicted instance). In other words, we aim to demonstrate that our approach may help medical physicians taking educated clinical actions to reduce the risk of diabetes.

As for the first use case discussed earlier in Section 4.1 (online advertising), we start from learning an ensemble of bagged decision tree classifiers, which is able to predict if a patient has diabetes or not. To do so, we first split the original dataset $\mathcal{D}$ into two portions – which accounts for 90% and 10%, respectively – using stratified random sampling: a training set ($\mathcal{D}_{\text{train}}$) used for learning the predictive model[14], and a test set $\mathcal{D}_{\text{test}}$ to assess the set of transformed instances generated according to our proposed algorithm. It is worth remarking that, although the predictive model is learned considering all the signals above, our feature tweaking algorithm only operates on features that can be "easily" perturbed. More specifically, features like PREGNANCY, DIABETES_PEDIGREE_FUNCTION and AGE cannot be tweaked, as they are inherently historical and therefore their cost of transformation can be considered as $+\infty$ (see Section 2.2).

Our best-performing ensemble classifier is a RF, which able to reach about 77% accuracy using 100 trees. Out of 76 held-out test instances, 51 are negative and 25 positive. Those 25 positive instances are then transformed into a set of ϵ-perturbations, using $\epsilon = 0.1$ as (global) tolerance and $\delta = \textit{euclidean\_distance}$ as cost function.

Figure 6 shows the direction of feature tweaks as computed from the set of transformations above. Generally speaking, results confirm what one would expect using domain knowledge on diabetes. In particular, turning a positive instance into a negatively predicted one translates into reducing the values of 4 main features, namely GLUCOSE, BMI, SKIN_THICKNESS (maybe positively correlated with BMI), and BLOOD_PRESSURE. This might sound apparently

[13]https://www.kaggle.com/uciml/pima-indians-diabetes-database/data

[14]We do not report all the details about the learning stage, as this has been thoroughly addressed when we described the first use case and in [5]; moreover, it is not the main scope of this section.

straightforward, purely because we are already backed up by well-established findings concerning the relationship between those features and the disease (e.g., it is now well known that reducing BMI lowers the risk of diabetes). Chances are, though, that other possibly unexpected suggestions would arise if unconventional features were included as predictors. Still, there are few additional insights that may be worth discussing. For example, we are not able to provide recommendation on how to change the value of insulin which is valid in general (i.e., some instances may be suggested to increase it, whilst some others to decrease it). This is compliant with the fact that there exists multiple types of diabetes; in some cases, diabetes is caused by *insulinopenia* (i.e., deficit of insulin secreted by pancreas), whereas in some others cells are unable to absorb insulin to regulate the concentration of glucose, no matter how much it is secreted by pancreas. In the former case, it might have sense to increase the value of insulin, so as to compensate the deficit. In the latter case, instead, adding more insulin might not help (in fact, it might cause saturation), since the problem is on its absorption rather than its secretion.

Note that out of the features that are deemed unchangeable – and therefore not subject to any recommendation – the number of pregnancies is suggested to be reduced. Of course, this value cannot be modified on a patient; still, it might be a good advice for future inmates. Instead, historical feature capturing genetic relationships between the patient and her/his relatives does not even seem important.

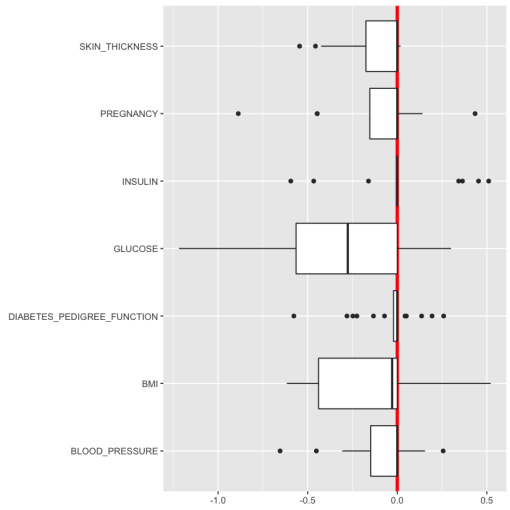


Fig. 6: Direction of feature tweaks for reducing the risk of diabetes.

4.3 Spam Filtering

The dataset used for this suite of experiments was collected at Hewlett-Packard Labs, and it is available for download from the UCI Machine Learning Repository[15] [33]. It contains 4,601 emails labelled as *spam* (denoted by 1, or positive) and *non-spam* (denoted by 0, or negative). Also this collection exhibits some class imbalance, since around 60% instances are non-spam and the remaining 40% spam.

[15]https://archive.ics.uci.edu/ml/datasets/spambase

In addition to the class label, each instance is represented by a set of 57 features, which indicate how frequently some specific words and characters occur in an email. More specifically, the following continuous features are defined:

- 48 real-valued features in the range $[0, 100]$ of type WORD_FREQ_`WORD`, each one measuring the percentage of words that match the term "`WORD`". A "`WORD`" is any string of alphanumeric characters bounded by non-alphanumeric characters or end-of-string (e.g., "`business`" or "`650`").
- 6 real-valued features in the range $[0, 100]$ of type CHAR_FREQ_`CHAR`, each one indicating the percentage of characters in the email that match "`CHAR`". The set of characters considered are: "`;`", "`(`", "`[`", "`!`", "`$`", and "`#`".
- 1 real-valued feature in the range $[1, \ldots, \infty]$ named AVG_CAPITAL_LENGTH as the average length of sequences of (contiguous) capital letters.
- 1 integer-valued feature in the range $[1, \ldots, \infty]$ named MAX_CAPITAL_LENGTH, which measures the length of the longest sequence of (contiguous) capital letters.
- 1 integer-valued feature in the range $[1, \ldots, \infty]$ named NUM_CAPITALS, which contains the total number of capital letters in the email.

Differently from the previous two use cases, where the main goals of using the feature tweaking algorithm were to *(i)* increase revenue (online advertising) and *(ii)* support medical physicians (helthcare), in this setting we aim to show that our method may help transform a truly spam email into a non-spam classified one. This is a typical example of *adversarial attack*, where a malicious entity is interested in enforcing an existing classifier to commit prediction errors by crafting ad hoc manipulation of the input. Such a kind of attack is the main subject of interest of *adversarial learning* [7], which is a branch of machine learning that studies how to design predictive models that are robust against artificial perturbations of the input [1], [9], [10], [11], [12]. It is worth remarking that our method requires knowing the internals of the model to perform the feature tweaking we describe, therefore it is quite different from the typical adversarial setting that considers the model as a black box.

Our original task consists of building an ensemble of bagged decision tree classifiers, which is able to distinguish between spam vs. non-spam messages. This step is done by first splitting the original dataset $\mathcal{D}$ into training and test set, which account for 90% ($\mathcal{D}_{\text{train}}$) and 10% ($\mathcal{D}_{\text{test}}$) of the total number of instances, respectively. We therefore use $\mathcal{D}_{\text{train}}$ for learning the model and $\mathcal{D}_{\text{test}}$ for generating input perturbations using our algorithm. The accuracy of the best-performing classifier is around 92% using an RF with 50 trees. Out of 460 held-out test instances, 301 are non-spam and 159 are spam. Eventually, we generate the set of ϵ-transformations from those spam instances with the same hyperparameter setting used for the healthcare scenario, i.e., $\epsilon = 0.1$ and $\delta = \textit{euclidean\_distance}$.

Figure 7 shows the direction of feature tweaks as computed from the set of transformations above, when considering the top-10 most important features.

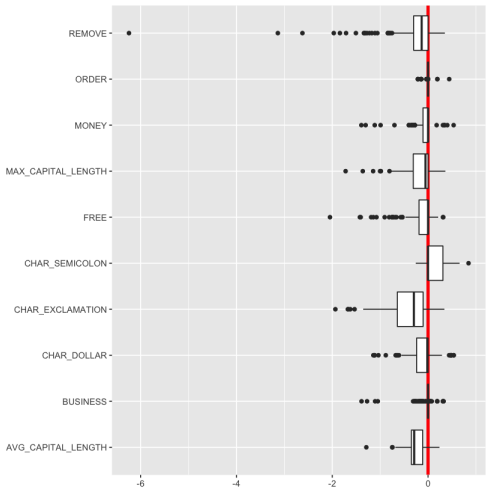


Fig. 7: Direction of feature tweaks for transforming a true spam message into a non-spam one.

From the box plot, it is quite clear that the vast majority of feature values need to be reduced whenever the goal is to transform a spam message into a non-spam one. This is mostly due to the way in which features have been engineered (i.e., they have been manually chosen so as to best reflect properties of spam emails). For example, we know from common experience that the presence of many exclamation marks ("`!`") is very often a good indicator of spammy content. Therefore, it should not be surprising our algorithm suggests to reduce the percentage of this symbol in the text. The same can be said for MAX_CAPITAL_LENGTH and AVG_CAPITAL_LENGTH. In fact, this confirms that our technique is able to capture which features encapsulate spam traits.

On the other hand, terms that might sound spammy, such as "`order`" or "`money`" do not seem to play a crucial role in masquerading spam emails. Interestingly, our algorithm recommends to increase the percentage of semicolon ("`;`") in the text. The rationale of this might be that semicolons more often appear in formal messages, since few people know how to use them properly. As a consequence of that, adding more semicolons to a spam message might induce the classifier to mistakenly label it as non-spam.

5 Evaluating Computational Efficiency

In this section, we compare the first, naïve implementation of our algorithm with the one proposed as a novel contribution of this work and discussed in Section 3.2. This leverages efficient space partitioning data structures for computing NearestNeighbourSearch rather than just performing a brute-force linear search over the set of candidate transformations. More specifically, we apply our approach to the well-known multiclass classification task of handwritten digit recognition. To this end, we use a sample $\mathcal{D}$ of 42,000 instances of the public MNIST dataset[16], which is available from Kaggle[17]. Each training example is made of a 28x28 pixel black and white image and its associated label, namely the digit it represents (i.e., one of the set $\{0, 1, \ldots, 9\}$). Each image is in turn represented by a 784-dimensional vector, where each dimension indicates the lightness or darkness of a pixel as an integer ranging between 0 and 255, inclusive.

We first split $\mathcal{D}$ using stratified random sampling into two portions: $\mathcal{D}_{\text{train}}$ and $\mathcal{D}_{\text{test}}$ accounting for 90% and 10% of the total number of instances, respectively. We thus train an RF model on $\mathcal{D}_{\text{train}}$ after the best number of trees (100) has been selected following a 5-fold cross validation run (average ROC AUC = 0.96). We generate all the candidate ϵ-transformations from all the k-leaved paths extracted from the trained model above, as specified by the BuildKLabelledInstance function defined in Figure1. Since there are 10 possible class labels (i.e., one for each digit) it turns out that $k \in \{0, 1, \ldots, 9\}$; overall we extracted a total of approximately 350,000 candidate ϵ-transformations, which are distributed over the 10 classes as shown in Figure 8.

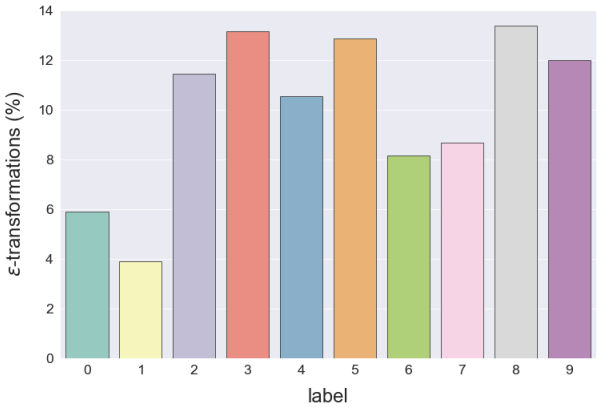


Fig. 8: Distribution of ϵ-transformations over labels.

We therefore build a set of space partitioning data structures, each one containing all the candidate ϵ-transformations associated with a specific class label. We repeat this operation using three distinct values of ϵ, namely 1, 5, and 10, which indicate the maximum "strength" of the change applicable to any pixel value in order to eventually satisfy a k-leaved path. More specifically, we use the implementations of k-d tree and ball tree provided by Python `scikit-learn`[18,19]. Note that this step, although compute-intensive, is done once for all.

To measure the speedup introduced with our new implementation, we proceed as follows. For a fixed value of ϵ, we extract a random sample of n instances from $\mathcal{D}_{\text{test}}$. Each sampled instance is an image with its associated label k indicating the digit it represents, which in turn is input to the TweakingFeatureAlgorithm described in Figure 2. This will return the optimal ϵ-transformation of the input query vector among the candidates previously computed, i.e., the top-1 closest to the original input instance w.r.t. a specific distance function δ and a target class label k', such that $k' \neq k$. It turns out that each input query vector can be transformed into 9 optimal ϵ-transformations, e.g., the input query vector for "`3`" can be transformed into 9 *new* instances representing the digits "`0`", "`1`", "`2`", "`4`", ..., "`9`".

[16]http://yann.lecun.com/exdb/mnist/
[17]https://www.kaggle.com/c/digit-recognizer/data
[18]https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KDTree.html
[19]https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.BallTree.html

In our experiments, we use $\delta = \textit{euclidean\_distance}$ and we further extend the list of ϵ-transformations retrieved by the algorithm to also include the top-5 and top-10 candidates for each input query vector and target class k'. Figure 9 shows

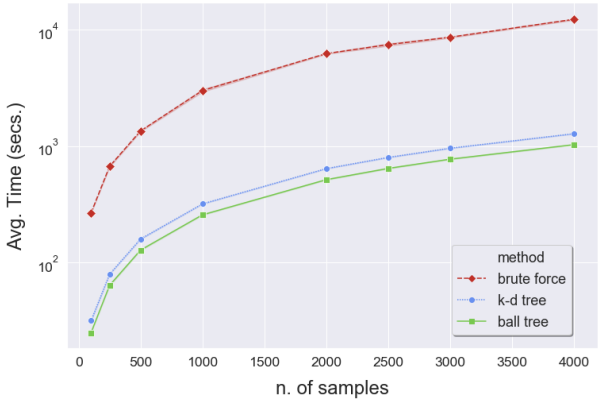


Fig. 9: Average execution time of NEARESTNEIGHBOURSEARCH for different query sample size.

the running time of our TWEAKINGFEATUREALGORITHM when NEARESTNEIGHBOURSEARCH is implemented using our first, naïve solution (i.e., brute-force linear scan), or a more efficient space partitioning data structure (i.e., k-d tree or ball tree). The plot shows how the average running time for computing top-1, top-5, and top-10 closest ϵ-transformations[20] varies when query sample size changes (i.e., $n = \{100, 250, 500, 1000, 2000, 2500, 3000, 4000\}$). Experiments were conducted on a 3,6 GHz Intel Core i7 processor with 16 GB of RAM. Two main considerations can be made out of this set of experiments. First, our two new implementations are both significantly more efficient than brute-force (i.e., ≈10x speedup), making them more convenient to deploy in practice. Second, no remarkable difference between the usage of k-d trees and ball trees are recorded, although the latter performs slightly yet consistently better.

We conclude this section by showing the distribution of the closest target labels for each original label, as follows. We consider all the $n = 4000$ query instances sampled from $\mathcal{D}_{\text{test}}$ as described above; for each of them, we keep track of its original label (i.e., which digit it represents) and the target label of the top-1 closest transformation (i.e., the digit of the nearest candidate transformation out of all possible ϵ-transformations). As it turns out, for each original label we can compute the distribution of target labels, as shown in Figure 10. For example, we may observe that the vast majority (i.e., around 40%) of top-1 closest transformations of a query vector representing the digit "1" are instances labeled as "7" (see the second plot on the first row of Figure 10). This somehow confirms what one would intuitively guess, as handwritten "1" and "7" may look indeed similar, and therefore the latter can be obtained from of a quite straightforward transformation of the former. Interestingly enough, this is not symmetrical; as a matter of fact, the closest transformations to query representing "7" are labelled as "9", which still seems reasonable.

[20]We only report the case of $\epsilon = 1$ as similar trend is observed for $\epsilon = 5$ and $\epsilon = 10$.

6 RELATED WORK

The research challenge addressed in this work is largely unexplored. Although machine learning has received a lot of attention in recent years, the focus has been mainly on the accuracy, efficiency, scalability, and robustness of the proposed various techniques.

Early works on extracting actionable knowledge from machine-learned models have focused on the development of interestingness metrics as proxy measures of knowledge actionability [34], [35]. Another line of research on actionable knowledge discovery concerns post-processing techniques. Liu et al. propose methods for pruning and summarizing learned rules, as well as matching rules by similarity [36], [37]. Cao et al. present domain-driven data mining; a paradigm shift from a research-centered discipline to a practical tool for actionable knowledge [38], [39].

Several works discuss post-processing techniques specifically tailored to decision trees [40], [41], [42], [43]. Yang et al. study the problem of proposing actions to maximise the expected profit for a group of input instances based on a single decision tree, and introduce a greedy algorithm to approximately solve such a problem [40]. This is significantly different from our work; in fact, our work is more related to the one presented by Cui et al. [15]. Here, the authors propose a method to support actionability for additive tree models (ATMs), which is to find the set of actions that can change the prediction of an input instance to a desired status with the minimum cost. The authors formulate the problem as an instance of integer linear programming (ILP) and solve it using existing techniques.

Similarly to Cui et al., we also consider transforming the prediction for a given instance output by an ensemble of trees, and we introduce an algorithm that finds the *exact* solution to the problem. Our work differs from theirs in several aspects: *(i)* we tackle the theoretical intractability (NP-hardness) of the problem by designing an algorithm that creates a feedback loop with the original model to build a set of candidate transformations without the need, in practice, to explore the entire exponential search space; *(ii)* we introduce another hyperparameter (ϵ) to govern the amount of change that can be sustained; *(iii)* we experiment with five concrete functions describing the cost of each transformation (δ); *(iv)* we leverage on the importance of each feature derived from the model to rank the final list of recommendations; *(v)* we focus on the actual recommendations generated, and how they impact in practice on three real use cases, if properly implemented.

More recent work on related topics are those of Ribeiro et al. [44], [45]. In particular, [44] presents LIME, a method that aims to explain the predictions of any classifier by learning an interpretable model that is specifically built around the predictions of interest. They frame this task as a submodular optimization problem, which the authors solved using a well-known greedy algorithm achieving performance guarantees. They test their algorithm on different models for text (e.g., random forests) and image classification (e.g., neural networks), and validate the utility of generated explanations both via simulated and human-assessed experiments.

Finally, a survey on methods for explaining black-box models has recently appeared in [46].

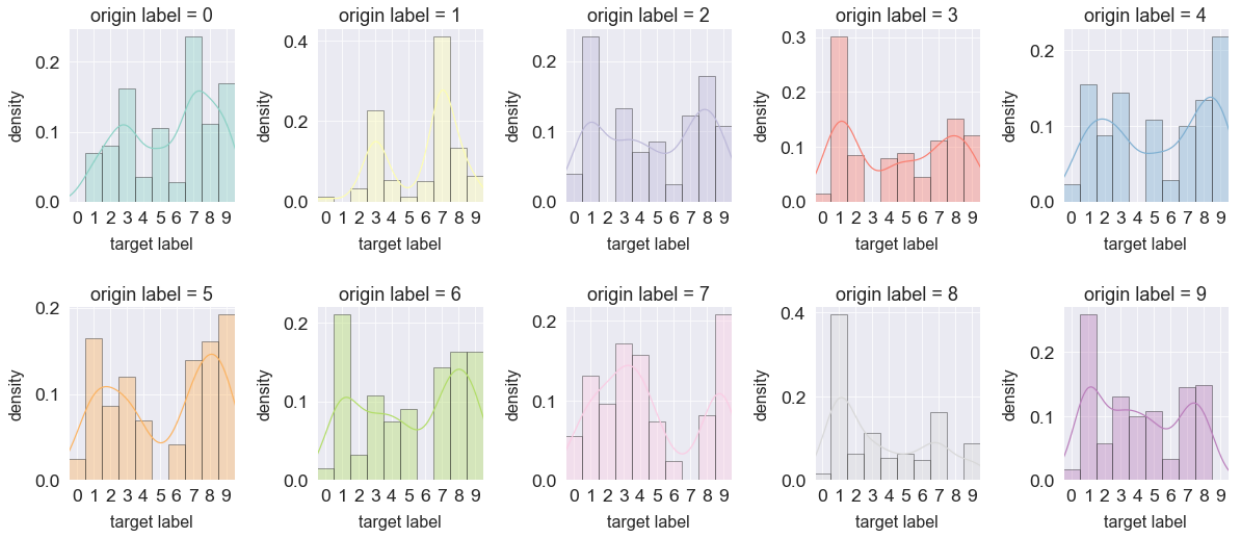


Fig. 10: Distribution of ϵ-transformations over target labels for each original label.

7 Conclusions

Machine-learned models are often designed to favour accuracy of prediction at the expense of human-interpretability. However, in some circumstances it is important to understand why the model returns a certain prediction on a given instance, and how such an instance could be transformed so that the model changes its original prediction.

We extend our discussion introduced in [5] by proposing an algorithm for *tweaking* input features to change the output predicted by an existing machine-learned model. This method is designed to operate on the general setting of multiclass classification, and exploits the feedback loop originated from the internals of any ensemble of bagged decision trees to generate recommendations for transforming a k-labelled predicted instance into a k'-labelled one (for any k, k' in the set of possible class labels).

The feasibility of our approach has been achieved in practice by: *(i)* setting an upper bound to the maximum number of changes affecting each instance (i.e., at most equivalent to the number of features), which can be controlled at training time; and *(ii)* making use of a spatial indexing data structure (e.g., k-d tree or ball tree) populated offline once for all when the model is learned, so as to reduce online computational complexity from exponential to at most quadratic in the number of features.

Finally, we demonstrate the applicability of our approach on four real-world use cases: online advertising, health-care, spam filtering, and handwritten digit recognition. Experiments confirm that our algorithm is able to suggest changes to feature values that help interpreting the rationale of model predictions, and would be indeed useful if implemented efficiently.

In future work, we plan to extend the approach presented in this work to other learning models, in particular to Gradient Boosted Decision Trees (GBDT) [47], and regression models. Another promising direction is that of formulating the solution within a reinforcement learning framework, enabling us to operate in the most generic setting possible and treating the model fully as a black box.

Acknowledgments

The authors would like to thank Huw Evans, Mahlon Chute, and all the Yahoo's internal team of creative strategists for their invaluable contributions in evaluating the quality of the method presented in this paper. In addition, a special mention goes to Satoshi Kato for publicly releasing his implementation of our framework as an R-package, along with a convenient visualization tool we used in this work.[21]

References

[1] L. Huang, A. D. Joseph, B. Nelson, B. I. Rubinstein, and J. D. Tygar, "Adversarial machine learning," in *AISec '11*. ACM, 2011, pp. 43–58.

[2] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. J. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," in *ICLR '14*, 2014.

[3] Y. Bengio, A. Courville, and P. Vincent, "Representation Learning: A Review and New Perspectives," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 8, pp. 1798–1828, 2013.

[4] Y. LeCun, Y. Bengio, and G. Hinton, "Deep Learning," *Nature*, vol. 521, pp. 436–444, May 2015.

[5] G. Tolomei, F. Silvestri, A. Haines, and M. Lalmas, "Interpretable Predictions of Tree-based Ensembles via Actionable Feature Tweaking," in *KDD '17*. ACM, 2017, pp. 465–474.

[6] L. Breiman, "Bagging predictors," *Machine Learning*, vol. 24, no. 2, pp. 123–140, 1996.

[7] D. Lowd and C. Meek, "Adversarial Learning," in *KDD '05*. ACM, 2005, pp. 641–647.

[8] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative Adversarial Nets," in *Advances in Neural Information Processing Systems*, 2014, pp. 2672–2680.

[9] D. Lowd and C. Meek, "Good Word Attacks on Statistical Spam Filters," in *CEAS*, vol. 2005, 2005.

[10] M. Barreno, B. Nelson, A. D. Joseph, and J. Tygar, "The Security of Machine Learning," *Machine Learning*, vol. 81, no. 2, pp. 121–148, 2010.

[11] B. Biggio, G. Fumera, and F. Roli, "Security evaluation of pattern classifiers under attack," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 4, pp. 984–996, 2014.

[12] N. Papernot, P. D. McDaniel, A. Sinha, and M. P. Wellman, "Towards the science of security and privacy in machine learning," *CoRR*, vol. abs/1611.03814, 2016. [Online]. Available: http://arxiv.org/abs/1611.03814

[21] https://github.com/katokohaku/featureTweakR

[13] F. Chollet, *Deep Learning with Python*, 1st ed. Greenwich, CT, USA: Manning Publications Co., 2017.
[14] L. Xu, A. Krzyzak, and C. Y. Suen, "Methods of Combining Multiple Classifiers and their Applications to Handwriting Recognition," *IEEE Transanctions on Systems, Man, and Cybernetics*, vol. 22, no. 3, pp. 418–435, 1992.
[15] Z. Cui, W. Chen, Y. He, and Y. Chen, "Optimal Action Extraction for Random Forests and Boosted Trees," in *KDD '15*. ACM, 2015, pp. 179–188.
[16] A. Guttman, "R-trees: A Dynamic Index Structure for Spatial Searching," in *SIGMOD '84*. ACM, 1984, pp. 47–57.
[17] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communincations of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.
[18] S. M. Omohundrol and S. M. Omohundro, "Five balltree construction algorithms," 1989.
[19] T. K. Ho, "The Random Subspace Method for Constructing Decision Forests," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 8, pp. 832–844, 1998.
[20] J. H. Friedman, J. L. Bentley, and R. A. Finkel, "An algorithm for finding best matches in logarithmic expected time," *ACM Transactions on Mathematical Software*, vol. 3, no. 3, pp. 209–226, 1977.
[21] S. Hwang, K. Kwon, S. K. Cha, and B. S. Lee, "Performance Evaluation of Main-Memory R-tree Variants," in *International Symposium on Spatial and Temporal Databases*. Springer, 2003, pp. 10–27.
[22] L. Arge, M. D. Berg, H. Haverkort, and K. Yi, "The Priority R-tree: A Practically Efficient and Worst-case Optimal R-tree," *ACM Transactions on Algorithms*, vol. 4, no. 1, pp. 9:1–9:30, Mar. 2008.
[23] T. Brinkhoff, H.-P. Kriegel, and B. Seeger, "Efficient processing of spatial joins using r-trees," in *SIGMOD '93*. ACM, 1993, pp. 237–246.
[24] C. Böhm and F. Krebs, "Supporting kdd applications by the k-nearest neighbor join," in *Database and Expert Systems Applications*. Springer Berlin Heidelberg, 2003, pp. 504–516.
[25] W. A. Burkhard and R. M. Keller, "Some approaches to best-match file searching," *Communications of the ACM*, vol. 16, no. 4, pp. 230–236, Apr. 1973.
[26] H. Raghavan and D. Hillard, "A Relevance Model Based Filter for Improving Ad Quality," in *SIGIR '09*. ACM, 2009, pp. 762–763.
[27] K. Zhou, M. Redi, A. Haines, and M. Lalmas, "Predicting Pre-click Quality for Native Advertisements," in *WWW '16*. International World Wide Web Conferences Steering Committee, 2016, pp. 299–310.
[28] M. Lalmas, J. Lehmann, G. Shaked, F. Silvestri, and G. Tolomei, "Promoting Positive Post-Click Experience for In-Stream Yahoo Gemini Users," in *KDD '15*. ACM, 2015, pp. 1929–1938.
[29] N. Barbieri, F. Silvestri, and M. Lalmas, "Improving Post-Click User Engagement on Native Ads via Survival Analysis," in *WWW '16*. International World Wide Web Conferences Steering Committee, 2016, pp. 761–770.
[30] J. R. Quinlan, "Induction of Decision Trees," *Machine Learning*, vol. 1, no. 1, pp. 81–106, Mar. 1986.
[31] L. Breiman, "Random Forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, Oct. 2001.
[32] J. W. Smith, J. Everhart, W. Dickson, W. Knowler, and R. Johannes, "Using the ADAP Learning Algorithm to Forecast the Onset of Diabetes Mellitus," in *Proceedings of the Annual Symposium on Computer Application in Medical Care*. American Medical Informatics Association, 1988, pp. 261–265.
[33] D. Dheeru and E. Karra Taniskidou, "UCI Machine Learning Repository," 2017. [Online]. Available: http://archive.ics.uci.edu/ml
[34] R. J. Hilderman and H. J. Hamilton, "Applying Objective Interestingness Measures in Data Mining Systems," in *PKDD '00*. Springer Berlin Heidelberg, 2000, pp. 432–439.
[35] L. Cao, D. Luo, and C. Zhang, "Knowledge Actionability: Satisfying Technical and Business Interestingness," *International Journal of Business Intelligence and Data Mining*, vol. 2, no. 4, pp. 496–514, Dec. 2007.
[36] B. Liu and W. Hsu, "Post-analysis of Learned Rules," in *AAAI '96*. AAAI Press, 1996, pp. 828–834.
[37] B. Liu, W. Hsu, and Y. Ma, "Pruning and Summarizing the Discovered Associations," in *KDD '99*. ACM, 1999, pp. 125–134.
[38] L. Cao and C. Zhang, "Domain-Driven Actionable Knowledge Discovery in the Real World," in *PAKDD '06*. Springer-Verlag, 2006, pp. 821–830.
[39] L. Cao, H. Zhang, E. Park, D. Luo, Y. Zhao, and C. Zhang, "Flexible Frameworks for Actionable Knowledge Discovery," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 9, pp. 1299–1312, Sep. 2013.
[40] Q. Yang, J. Yin, C. X. Ling, and T. Chen, "Postprocessing Decision Trees to Extract Actionable Knowledge," in *ICDM '03*. IEEE Computer Society, 2003, pp. 685–688.
[41] K. Masud and M. R. Rashedur, "Decision Tree and Naïve Bayes Algorithm for Classification and Generation of Actionable Knowledge for Direct Marketing," *Journal of Software Engineering and Applications*, vol. 6, no. 4, pp. 196–206, 2013.
[42] Q. Yang, J. Yin, C. Ling, and R. Pan, "Extracting Actionable Knowledge from Decision Trees," *IEEE Transactions on Knowledge and Data Engineering*, vol. 19, no. 1, pp. 43–56, Jan. 2007.
[43] J. Du, Y. Hu, C. X. Ling, M. Fan, and M. Liu, "Efficient Action Extraction with Many-to-Many Relationship between Actions and Features," in *Proceedings of the International Workshop on Logic, Rationality and Interaction*. Springer Berlin Heidelberg, 2011, pp. 384–385.
[44] M. T. Ribeiro, S. Singh, and C. Guestrin, ""Why Should I Trust You?": Explaining the Predictions of Any Classifier," in *KDD '16*. ACM, 2016, pp. 1135–1144.
[45] M. T. Ribeiro, S. Singh, and C. Guestrin, "Model-Agnostic Interpretability of Machine Learning," in *2016 ICML Workshop on Human Interpretability in Machine Learning*, 2016, pp. 91–95.
[46] R. Guidotti, A. Monreale, S. Ruggieri, F. Turini, F. Giannotti, and D. Pedreschi, "A survey of methods for explaining black box models," *ACM Computing Surveys*, vol. 51, no. 5, pp. 93:1–93:42, 2019.
[47] J. H. Friedman, "Stochastic Gradient Boosting," *Computational Statistics and Data Analysis*, 2002.

Gabriele Tolomei is Assistant Professor at University of Padova, Italy. From 2014 to 2017, he has been a Research Scientist at Yahoo Research in London, UK. He received his Ph.D. in Computer Science from University of Venezia Ca' Foscari, Italy, in 2011. His main research interests include Web Search, Machine Learning, and Computational Advertising. On those topics, he authored around 30 papers, which appeared in topmost international peer-reviewed journals and conferences. Moreover, he filed 4 patents with the US Patent and Trademark Office.

Fabrizio Silvestri is a Research Scientist at Facebook AI London, UK, in the Integrity team. His interests are in Web Search, and in particular his specialization is building systems to better interpret queries from users' search traces. Prior to Facebook, Fabrizio was a Principal Scientist at Yahoo Research, where he has worked on sponsored search and native advertising within the Yahoo Gemini project. Fabrizio holds a Ph.D. in Computer Science from the University of Pisa, Italy, where he studied problems related to Web Information Retrieval with particular focus on efficiency issues like caching, collection partitioning, and distributed IR in general. He authored around 140 research papers, which are published in topmost international peer-reviewed journals and conferences.