Explaining White-box Classifications to Data Scientists (technical report)

Daniel Deutch Nave Frost

Abstract

A main component of many Data Science applications is the invocation of Machine Learning (ML) classifiers. The typical complexity of these classification models makes it difficult to understand the reason for a result, and consequently to assess its trustworthiness and detect errors. We propose a simple generic approach for explaining classifications, by identifying relevant parts of the input whose perturbation would be significant in effecting the classification. Our solution requires white-box access to the model internals and a specification of constraints that define which perturbations are "reasonable" in the application domain; both are typically available to the data scientist. We have implemented the approach for prominent ML models such as Random Forests and Neural Networks. We demonstrate, through use cases and experiments, the effectiveness of our solution, and in particular of its novel use of constraints.

1 Introduction

Data Science heavily relies on the use of Machine Learning classifiers to derive insights and predictions in multiple domains. As these models get more and more sophisticated, the quality of prediction generally increases; but the downside is hampering understandability: why did we get a result? what would change it? Such questions that are often investigated in the context of database queries, generally remain unanswered when a complex Machine Learning model is used. This in turn is harmful to the applications quality and to the trust level in their results.

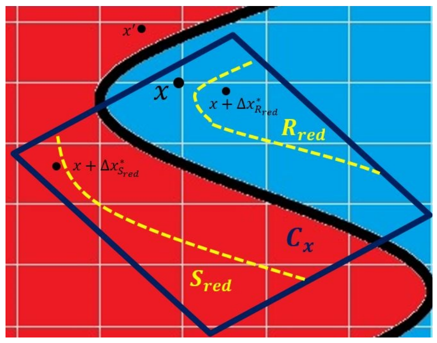
A particular application domain for which we will demonstrate the presentation of explanations is that of *image classification*, namely the process of associating images with labels that describe their contents. Typically multiple labels are assigned to each image, and each is given a confidence score. Explaining the reason for the assigned labels can potentially reveal recurring errors of the classification model, and thus guide its improvement, e.g. by further training on images of a particular flavour.

A major conceptual challenge is how do we define explanations? Our approach is to explain output instances by focusing on minimal hypothetical perturbations to the input (e.g. features or pixels) that significantly change the model confidence in a particular label of interest. Intuitively, the input parts that are modified through such a perturbation were the most influential for the model original decision. Both positive and negative contributions, i.e. ones that significantly increase or decrease the model confidence in a particular decision (label), are informative in this respect – they both indicate the impact of the input part on the label.

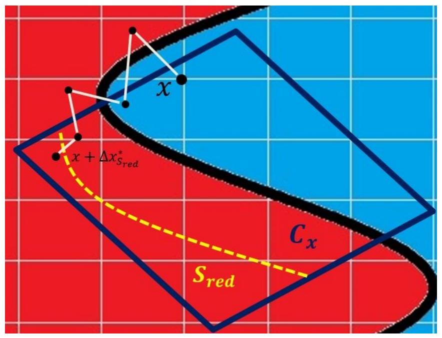
A conceptual problem with this approach is that an impactful modification is not necessarily informative for presentation as an explanation. In particular, there is a flourish of research on *adversarial Machine Learning* [13, 5, 46, 21, 39], demonstrating that minor, unnoticeable changes spread throughout the input (in particular, an image) can in practice "fool" the model, leading to a significant change of the labelling decision. Such unnoticeable perturbation to (almost) every pixel of the image is clearly uninformative for explanations.

To this end, our approach is to impose *constraints* over the allowed perturbation, and then aim at finding optimal perturbations out of those that satisfy the constraints. The constraints intuitively capture the usefulness of a perturbation in the context of explanations. For instance, it may bound the number of pixels or super-pixels (sets of pixels that are "semantically" related) that may be modified, thus leading the perturbation to be focused on a small number of pixels/areas which in turn can be transformed into a meaningful explanation (see Example 3.4, explained in detail in Section 3). For other classification tasks (beyond image classification) the constraints may bound the domain of (modified) features, impose functionality constraints between multiple features, etc. Designing constraints requires some domain knowledge, but we demonstrate that even reasonably simple constraints can lead to satisfactory explanations in practical settings. We propose a formalism for constraints – linear inequalities over the original and modified values – and demonstrate that it allows specification of realistic and useful constraints. We further discuss extensions beyond this class.

The input to our algorithm is then (1) a "white-box" ML model instance, such as the trees of a Random Forest [25, 26] or the instance of a given Neural Network [36, 33, 20], (2) an input whose classification we wish to explain, (3) a classification class of interest and a required confidence level, and (4) a specification of constraints. The algorithm's goal, at a high level, is demonstrated in Figure 1a. x is the input point, currently classified as "blue". The classification class of interest is "red", and we either wish to explain what would change the decision to be red, and be sufficiently confident at that – i.e. push the classification beyond the "S_{red}" line, or conversely, to further reduce the confidence in "red", i.e. find a modification to be classified beyond the "R_{red}" line. We further have constraints on the allowed modifications; these are captured by the dark blue box. Our goal is to find a closest point to x that is classified as "red" with the required confidence, and is also inside the "allowed" area. So x', which may be the closest point to x having the required "red" confidence, does not qualify since it does not satisfy the constraints. Instead the result we wish to obtain is the point denoted as $x + \Delta x^*_{S_{red}}$. Similarly if we wish to (sufficiently) decrease the confidence in "red", then the result should be $x + \Delta x^*_{R_{red}}$.



(a) Optimal changes example



(b) Optimal changes path

Figure 1: Optimal changes

To achieve that, the algorithm operates iteratively, as illustrated in Figure 1b. Each iteration consists of two steps, as follows.

Perturb The first step is perturbation, where we look for minimal changes to the input so that the output is changed in the "right" direction (ignoring the constraints). The implementation of this step depends on the type of the ML model, and performing it optimally is already NP-hard for e.g. Random Forests and Neural Networks; still, we provide effective heuristics for these models. For Random Forests, the algorithm iterates over the decision trees one by one. For each tree we compute a "candidate" perturbation that will lead the decision to one of the relevant leaves. Eventually, the algorithm selects the candidate agreed upon by the majority of trees. For Neural Networks we rely on each of the input features gradient, that is obtained by back-propagation through the network. The perturbation will then shift each feature value according to its gradient.

Project As demonstrated in Figure 1b, the perturbation step may lead to a point that violates the constraints. If this is the case, we *project* it back to the constrained area, i.e. find a closest point that does satisfy the constraints. The projection step is oblivious to the model and depends only on the constraints. For constraints that are expressed as linear inequalities (which, as mentioned above, is a reasonably expressive formalism), the projection step can be effectively implemented through Quadratic Programming. We also briefly discuss the implementation of this step for non-linear, convex constraints.

Each application of *perturb* intuitively "pushes" the point in the right direction in terms of its tag. The application of *project* then returns the point to the constrained area. If the projection step had not made "too much harm" in terms of the model decision, then we have obtained a "legal" point that is closer to the tag we wish to achieve, and we can continue iterating to improve it. The algorithm is not guaranteed to converge in general, but our experimental study indicates its effectiveness in practice.

We have implemented the algorithm with concrete perturbation and projection functions for multiple ML models (Linear model, Random Forests and Neural networks), and multiple types of constraints. We have applied it to state-of-the-art ML solutions (including in particular Google's `Inception-v3` [44, 45]). We demonstrate the effectiveness of our solution in multiple ways: first, we show examples of the explanations that are obtained through our implementation (see Figure 8), and show two particular use cases where we have used explanations for concrete insights on how to improve existing models. Then, we provide a user study comparing our solution both to a simple baseline of computing explanations based solely on minimal changes (as in adversarial ML), and to the state-of-the-art tool for ML instance explanation, `LIME` [41] (see discussion of related work in Section 2). Finally, we analyze computational aspects of our solution, comparing it to the optimal one where applicable.

The rest of this paper is organized as follows. Section 2 overviews related work. Section 3 formally defines the problem and provides examples. Section 4 introduces the general framework and algorithm, and in Sections 5 and 6 we provide concrete implementations for the perturbation and projection functions. Section 7 shows use cases and results of our implementation, and Section 8 describes the results of our experimental evaluation. We conclude with a discussion of future work in Section 9.

2 Related Work

We overview related work in three areas: works that aim at explaining ML models and results; explanations in database contexts; and adversarial Machine Learning.

Explanations for ML The need for explanations in the context of Machine Learning, notably rising from the complexity of employed models, is well recognized in the ML community. Consequently there is a large body of work on the problem, usually under the umbrella of *interpretable Machine Learning* [47, 15, 29, 40]. A prominent approach in this respect is to attempt at *explaining the model as a whole*, e.g. by approximating its specification through an instance of a simpler-to-understand (interpretable) model. For example, [27, 28] computes, for a given ML model, a reasonably small set of logical rules that should intuitively capture a similar reasoning to that of the model. An inherent limitation of this approach is that such simple rules would generally fail to capture all the nuances of the model decisions. A second approach, that we follow here, is not to explain the model as a whole but rather the reasoning underlying individual classifications. Notably, `LIME` [41] is one of the first solutions following this approach. Unlike our work, [41] assumes neither access to the internals of the model instance (i.e. the model is a black box) nor constraints. At a high-level, their approach is to randomly choose points in space that are in the proximity of the point whose prediction they wish to explain, and probe the model instance for predictions at these points. Then, they train a simple interpretable model (e.g. a linear model or a decision tree), using the decisions of the original predictor as training data. Finally, for such simple models it is easy to identify the most influencing features of the input, and these are used for explanations.

There are several aspects distinguishing our work from that of `LIME` and similar works [43, 30]. Our solution requires additional input: white-box rather than black-box access to the ML model instance, and the specification of constraints. These additional pieces of information are typically available to the data scientist or can be designed by her with a reasonable effort. Our main contribution is in demonstrating that we can leverage this domain knowledge to significantly improve the quality of explanations. *In particular, the use of constraints for explanations has, to our knowledge, not been studied in the context of ML.* It is consistent with conventional wisdom in data management (see discussion below), and as such our approach may pave the way for holistic explanations that account for the full data science cycle. A further distinguishing factor of our approach in this respect is that it is deterministic and purely algorithmic (e.g. `LIME` uses randomization and Machine Learning to compute the explanations). See further discussion in Section 9.

Provenance for DB Queries There is a large body of work on provenance and explanations for database queries (see e.g. [23, 8, 42, 10, 12]), as well as some recent work on provenance for Data Mining [19]. Specifically, the approach of explaining query results based on minimal changes in the input database, and related notions of causality and responsibility, has been studied in e.g. [38, 37, 48]. Explanations for non-answers have been studied in multiple lines of work, e.g. by identifying responsible query operators [9] or relevant input tuples (e.g. [4]). Our use of constraints is somewhat related to the work on repairs (e.g. [2, 17, 3, 18]) which typically aims at minimal changes to a “dirty” database that satisfy the given constraints. In a sense, we “transfer” the approach of constrained-based minimal changes to the setting of explanations for ML. As mentioned above, beyond the direct contribution of improving explanations of the ML model classification, this consistency of approaches is advantageous for future integration of explanation solutions.

Adversarial Machine Learning In the context of ML, minimal modifications in the input that change the model output have been extensively studied as means of "fooling" the ML model (see e.g. [22, 46]). Surprisingly, it was shown that carefully placed, minor changes to the input can cause the model to output a different class with high confidence. Adversarial algorithms have been in particular designed for Neural Networks, Decision Trees and, among many others (see e.g. [39, 1, 32]). Even though the changes are specifically tailored to the model instance, they look to the human eye as plain noise. This is a highly favourable feature for adversarial examples, which is the goal of these works, but (as we show) it makes the solution unsuitable for explanations. Our more refined model that restricts the modified inputs to satisfy some pre-specified constraints allows to identify "areas" that have the most impact on the model output, and use these for explanations.

3 Problem Statement

We formally define a generic formulation of the problem. Let d be the dimension of inputs to the Machine Learning model and let $\mathcal{L}$ be its domain of classification labels. The input to the problem is then:

- A Machine Learning model instance M, which may be expressed as a function $M : \mathbb{R}^d \times \mathcal{L} \mapsto [0, 1]$, such that $\forall x \in \mathbb{R}^d$, M satisfies that $\sum_{l \in \mathcal{L}} M(x, l) = 1$, i.e. it assigns a probability to each pair of possible input and a class label[1]. We will consider particular types of ML models below.
- An vector $x \in \mathbb{R}^d$, which is the input to M whose classification we wish to explain.
- A label $l \in \mathcal{L}$ of interest and required confidence levels in $[0, 1]$.
- A constraint function, mapping vectors in $\mathbb{R}^d$ to sub-spaces of $\mathbb{R}^d$. For a given input $x \in \mathbb{R}^d$, the set $C_x \subseteq \mathbb{R}^d$ denotes the allowed modifications of x.

Intuitively, we are interested in minimal changes to the input vector such that the model will be either more or less confident in a given label. The problem is further parametrized by the level of confidence that we are looking for. As explained in the Introduction, imposing constraints on the allowed modifications is necessary to avoid "adversarial" changes that do not serve as meaningful explanations.

We next formalize the problem; recall that $M(y, l)$ is the confidence assigned by M to labeling the input vector y using the label l.

Definition 3.1 *Given a model M, input x, desired label l and a set of constraints C_x the* ***label supporting modifications*** *set defined as:*

$$S_l = \{\Delta x \mid (x + \Delta x) \in C_x \textit{ and } M(x + \Delta x, l) \geq \beta_1\} \tag{1}$$

[1]In practice, many labels may be associated a confidence value 0, and the model may output a ranked list of labels ordered by confidence.

Similarly the ***label refuting modifications*** *set defined as:*

$$R_l = \{\Delta x \mid \ (x + \Delta x) \in C_x \textit{ and } M(x + \Delta x, l) \leq \beta_2\} \tag{2}$$

Where β_1 and β_2 are constant values representing confidence thresholds.

We then define the **optimal label supporting\refuting modification** as the smallest change in the respective set, as follows.

Definition 3.2 (Problem Statement) *Given M, x, l and C_x as above, we define the optimal supporting modifications as $\Delta x^*_{S_l} = \arg\min_{\Delta x \in S_l} \|\Delta x\|$ and the optimal refuting modifications as $\Delta x^*_{R_l} = \arg\min_{\Delta x \in R_l} \|\Delta x\|$. Our problem is to find optimal supporting and refuting modifications.*

Input Specification Our problem statement is generic in that it may be applied to any ML model and any constraints. This leads to the question of how is the input to the problem specified. For the ML model, we require white-box access. This means that instantiations of our framework will, for example, take as input the decision trees of a Random Forest, or the full net structure of a Neural Net. As for constraints, we will in particular consider ones that are specified as *linear inequalities* over the original and modified input values. Examples include constrains indicating that only specific features can be changed, or restricting the total number of features that may be changed. In the domain of image classification we can limit the number of pixels we allow to change from the original picture; a more sophisticated constraint can first process the image to identify different areas in the image (e.g. super-pixels) and prevent modification of too many areas. These constraints may all be expressed through linear inequalities, which are relatively simple to write. We discuss other types of constraints in the sequel.

We next demonstrate the problem and the introduced notions through a simple example.

Example 3.3 *Consider a simple linear model for loan applications, where the model takes into account two binary parameters, income and debt (so an input vector is $x = [x_{income}, x_{debt}]$). For each loan request, the model will return a score based on the following formula:*

$$M(x, Approve) = 0.5 + \frac{x_{income}}{2} - \frac{x_{debt}}{2},$$

consequently $M(x, Deny) = 1 - M(x, Approve)$. In this simple example we assume that modifications are constrained simply by the domain of variable values, so $C_x = \{0, 1\}^2$ for every x.

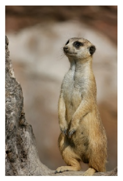
Further assume that loans are approved only if they have model score of 1 *and consider an applicant with $x = [1, 1]$ who applies for a loan. Her model score is* 0.5*, and the application is consequently denied. Naturally, the applicant may be interested in understanding the denial reason. In particular, she would like to know what changes will* support *the loan approval (i.e. increase the model* *score to $\beta_1 = 1$). In this case, changing her profile to $x = [1, 0]$ (setting debt to be 0) will lead to the application approval. Note that changing her profile to $x = [2, 1]$ will have the same score, but it contradicts C_x (and indeed, this is uninformative). Based on the supporting changes, we have identified that the debt parameter has a major influence on the model result. We may further look for changes that* refute *the loan approval (i.e. decrease the model score to $\beta_2 = 0$). This may be achieved by modifying the income value such that $x = [0, 1]$. Combined, we observe that both income and debt parameters have a significant influence on the model decision.*

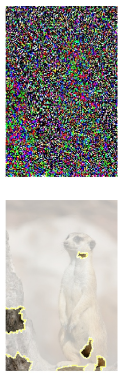
The following example shows explanations that we compute for decisions made by *image recognition* model. The explanations are based on both supporting and refuting modifications. To pictorially present them, we identify the different areas (super-pixels) in the image. For each super-pixel, we measure the $L2$ distance of the pixel modifications within it, and normalize by division in the number of its pixels. The top-10 areas that contained the most modifications are highlighted. We will also use this example to demonstrate the importance of constraints.

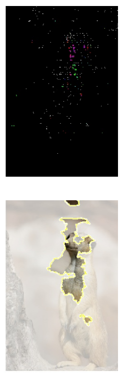
Example 3.4 *Figure 2a presents an image that was (correctly) labeled by the* `Inception-v3` *[45] model as a Meerkat, with 82% confidence. We look for small changes to pixels that will impact the model decision, and we set our target to reduce its certainty that the image contain a Meerkat to be below 15%. Intuitively, pixels whose change would reduce the confidence in the label are ones that are likely to have had significant influence on the original decision. We show the results for three different cases of constraints, as explained below. For each case, we show in the top row all pixels that have changed: each pixel is coloured by its delta in each of the RGB components. In the bottom row we highlight the parts of the image that contained the most changes as explained above. Those parts are the intuitively the "reason" that the model tagged the image as a Meerkat.*

In Figure 2b we impose no constraints at all. Observe, as shown in the top row, that this has resulted in changing all of the image pixels. Furthermore, presenting the areas with most changes is also uninformative in this case, as we found little correlation between the model decision and the areas of significant modifications. This is consistent with adversarial ML: many pixels are (slightly) modified, throughout the picture; the result does not lend itself towards useful explanations.

In contrast, in Figure 2c we impose a simple constraint: we allow modification of only 1% of the pixels. This leads our framework to look for the most indicative pixels, and indeed the areas of major changes are now better focused: they are mostly around the Meerkat's head and torso, with a few exceptions. Furthermore, more carefully designed constraints lead to better explanations. In Figure 2d we show the results obtained for constraints that restrict the modification of superpixels. Namely, we had first preprocessed the image and identified the superpixels; then the constraints allow to modify only pixels that reside in up *to* 10 *superpixels. Observe that the resulting explanation is better focused and more informative.*


(a) Original image


(b) Adversarial


(c) Limited pixels

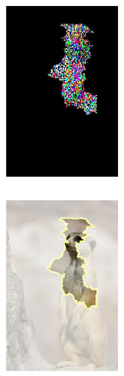

(d) Limited superpixels

Figure 2: Adversarial vs. Constraints

4 The General Case

We have formally defined the problem of finding optimal label supporting and refuting modifications and illustrated its application to explanations. We have defined the problem in general terms; in this section we propose a general algorithmic framework for its solution.

We start by observing that in general, one cannot achieve optimal solutions, unless $P = NP$. This holds (by a simple reduction from CNF satisfiability) for the commonly used models of Random Forests and Neural Networks, and even in the absence of constraints:

Proposition 4.1 *Finding optimal label supporting and refuting modifications is NP-Hard for both Random Forest and Neural Networks, and even if* $C_x = \mathcal{R}^d$ *for every* x.

We point out that for simpler models, such as decision trees and linear models, and in the absence of constraints, the problem is tractable; see the following section. Also see Section 6 for a discussion of tractable classes of constraints.

We next present our generic framework, shown in Algorithm 1. The algorithm operates in an iterative manner, at each step it applies two main steps, each of which requires a specialized implementation. In each iteration, it first aims at finding a small perturbation over the point in a direction that will lead the model to classify it in the desired label (Line 3). However, the modified point is not guaranteed to obey the constrains given by C_x, and so we project the modified point into C_x (Line 4). This will ensure that after each step, and in particular the final solution, will be legitimate and that the changes are meaningful, as illustrated in the previous section. Refuting modifications are found in a symmetric manner: we look for perturbations that shift the model decision in the opposite direction, and use a stopping criteria when the confidence is below the desired threshold.

Algorithm 1: Black Box - Supporting changes

```
input  : ML model $M$, point $x$, label $l$, constraints $C_x$
output: Minimal legitimate changes to $x$ according to $C_x$ so it will be
        tagged as $l$ according to $M$ with confidence higher than $\beta$

1 $x' = x$;
2 while $M.confidence(x', l) > \beta$ do
3     $x' + \Delta x' = M.perturb(x', l)$;
4     $x' = project(x' + \Delta x', C_x)$;
5 return $x'$;
```

Example 4.2 *Reconsider the illustration in Figure 1b. Point x was classified to be Blue, but we are interested in finding a point that is close to x, classified as Red and satisfies the constraints. At each iteration we find a small perturbation that "pushes" the point to be Red, however in case the point does not satisfy the constraints it will be projected into the constrained space specified by C_x. This is done repeatedly, until we obtain a point labeled as Red with high enough confidence, inside C_x and relatively close to x.*

We note that the convergence of the algorithm is not guaranteed in general, but our experimental study (Section 8) indicates that the algorithm typically converges after a small number of iterations.

In the next two sections we will discuss in depth, and provide concrete implementations for, the two main parts of the algorithm, i.e. *perturb* and *project*:

Perturb is a step that depends on the model structure, aiming to find a small change to x' in the desired direction. This can be achieved by examination of the model and the point, and identification of the features with the highest influence on the model decision. Since this is a white-box solution we can inspect the model instance, guiding our search of perturbations. In section 5 we describe how to find those perturbation for different types of models.

Project is a step that depends on the constraints. In case the perturbation shifted the point to be illegitimate based on C_x this step will project the point to satisfy the constraints. In section 6 we describe how to implement the projection step, based on different types of constraints.

5 Perturbation Implementation

We present, in a high-level, three concrete ML models to which we will apply our framework: Linear Models, Random Forests, and Neural Networks. For each model, we present a dedicated implementation of the Perturb function.

5.1 Linear Models

Linear models are common and simple tools for learning a linear relationship between the input space X and the outcome Y. The model decision is based on a linear function $h([x_1, ..., x_d]) = b + c_1 \cdot x_1 + ... + c_d \cdot x_d$, whose value is then tested to decide the label assigned to $x = [x_1, ..., x_d]$. We will consider the case where one of two labels is assigned, based on whether or not the value of h exceeds some threshold T; the confidence of the model in its decision is defined as $|h(x) - T|$. In this case, finding an optimal perturbation with respect to a given input is straightforward, as follows.

Proposition 5.1 *Let*

$$h([x_1, ..., x_d]) = b + c_1 x_1 + ... + c_d x_d,$$

and let x be an input vector such that $h(x) = \alpha$. For any $\beta > \alpha$ the following holds:

$$\underset{\{x' | h(x') = \beta\}}{\arg\min} \|x' - x\| = [x_1 + \epsilon c_1, ..., x_d + \epsilon c_d],$$

where $\epsilon = \frac{\beta - \alpha}{\sum c_i^2}$

Proof 5.2 *Let $x' = [x_1 + a_1, ..., x_d + a_d]$ be a vector such that $h(x') = \beta$. This means that:*

$$h(x') - h(x) = \sum_i c_i a_i = \beta - \alpha.$$

According to the geometric definition of dot product:

$$\sum_i c_i a_i = \|[c_1, ..., c_d]\| \cdot \|[a_1, ..., a_d]\| \cos(\theta) = \beta - \alpha,$$

where θ is the angle between the vectors $[c_1, ..., c_d]$ and $[a_1, ..., a_d]$. Hence, we can derive that:

$$\|[a_1, ..., a_d]\| = \frac{\beta - \alpha}{\cos(\theta)\|[c_1, ..., c_d]\|}.$$

Since we wish to minimize $\|x' - x\| = \|[a_1, ..., a_d]\|$ we have that $\cos(\theta) = 1$. This means that $\theta = 0$ which implies that $[a_1, ..., a_d]$ is obtained by a linear function of $[c_1, ..., c_d]$, i.e. for every i $a_i = \epsilon c_i$. Thus we conclude that:

$$x' = [x_1 + \frac{\beta - \alpha}{\Sigma c_i^2} c_1, ..., x_d + \frac{\beta - \alpha}{\Sigma c_i^2} c_d].$$

Linear models are generally considered “interpretable”, in the sense that identifying the most relevant attributes can be done by simply choosing those that appear with the largest absolute value of their coefficient. The above result thus serves mainly as a “sanity check” for our approach. Indeed, observe that our pertubation function is consistent with this conventional wisdom: the obtained modification value is in fact independent of x and indeed the magnitude of each attribute modification is only based on its coefficient.

5.2 Random Forests

A Random Forest is an ensemble learning method [25, 26], used for many ML tasks. It consists of a set of decision trees that were trained to classify the input space; its output for a given input instance is the mean decision of the forest trees. When it consists of many trees, it is hard to understand the reason for the final model decision, as it depends on the interaction between the different trees. This interaction will also be the main challenge for our solution.

We start by designing an algorithm for a single tree in the model, then consider the full forest.

Decision Trees Algorithm 2 implements the Perturb function for a single decision tree. Given a decision tree, an input vector x and a label l, the algorithm returns a modified vector x', such that x' was obtained from x with minimal number of changes, and the tree assigns the label l to x'. The algorithm iterates over the tree leaves that contain the label l (Lines 2, 3). For each of these leaves, it “climbs” upon the path from the leaf to the root (While loop in Line 6), and gradually modifies the vector according to the conditions associated with nodes along this path (Line 7). After the inner loop terminates, the updated vector x' is guaranteed to be labelled by the tree as l, as it satisfies all the conditions along the path from the tree root to a leaf labelled l. In Line 9, x' is added to the set of candidate perturbation results. After iterating over all relevant leaves and collecting their candidates, all that remains is to return the candidate with minimal distance from x (Line 10).

Example 5.3 *Consider the decision tree illustrated in Figure 3, and the input vector $x = [2, 3, 6]$. The decision tree will label x as 0, since is classification will lead to the highlighted leaf. Let us consider the minimal changes to x such that the tree will classify the modified vector as 2. Algorithm 2 will traverse all of the leaves labeled as 2 and will examine the minimal changes required to result in each of them. For each leaf we will traverse the path up to the tree root (these paths are highlighted in bold in Figure 3), and will update the vector according to the accumulated conditions. After traversing all of these paths, for each leaf labelled in 2 we will have one candidate vector that will lead to it. For this example, the candidates are $[2, \underline{6}, 6]$, $[\underline{4}, 3, 6]$ and $[\underline{4}, \underline{5}, 6]$, ordered by the order of relevant leaves (underlined coordinates represent changes to the initial input vector). Comparing these vector to the initial input $x = [2, 3, 6]$ will reveal that*

Algorithm 2: Decision Tree - Perturb

```
input  : Decision Tree $T$, point $x$, label $l$, constraints $C_x$
output: Minimal changes to $x$ so it will be tagged as $l$ according to $T$
1  $Xs = \{\}$;
2  foreach $leaf \in T.leaves$ do
3      if $leaf.label() = l$ then
4          $x' = x$;
5          $parent = leaf.parent()$;
6          while $parent \neq Null$ do
7              $x' = Update(x', parent.condition())$;
8              $parent = parent.parent()$;
9          $Xs.add(x')$;
10 return $\arg\min_{x' \in Xs} \|x - x'\|$;
```

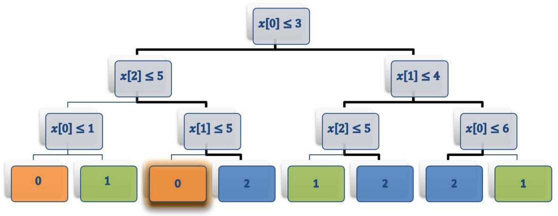


Figure 3: Decision tree

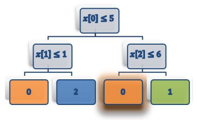
the required minimal perturbation changes $x[0]$ *to* 4*. Thus the algorithm will return* $x' = [4, 3, 6]$.

For a single decision tree, we achieve optimal perturbations:

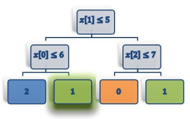
Proposition 5.4 *Algorithm 2 guarantees to return optimal perturbation, i.e.*

$$\arg\min_{\{x' | T(x') = l\}} \|x - x'\|.$$

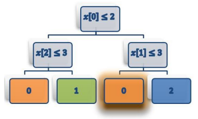
Proof 5.5 *Let* $x^* = \arg\min_{\{x'|T(x')=l\}} \|x - x'\|$ *be the optimal perturbation. Since* $T(x^*) = l$ *we know that it leads to some* $leaf_{x^*}$ *whose label is* l*. Algorithm 2 found some candidate* x'*, based only on updates of features along the path from the tree root to* $leaf_{x^*}$ *(denote this path by* p*). Now, assume that there exists a feature* i *such that* $|x[i] - x^*[i]| < |x[i] - x'[i]|$*. Observe that* i *must appear in the path* p*, since* x' *only differs from* x *for features appearing in* p*. However, for features that appear in* p*, Algorithm 2 performed the minimal update in order to satisfy the condition required for cascading along* p*. Hence, since* x^* *must also satisfy this condition, this contradicts the fact that* $|x[i] - x^*[i]| < |x[i] - x'[i]|$*. Thus there is no feature* i *such that* $|x[i] - x^*[i]| < |x[i] - x'[i]|$*, which means that* $\|x - x'\| \leq \|x - x^*\|$*, and since Algorithm 2 has* x' *as a candidate for selection (in its Line 10), it is guaranteed to return the optimal perturbation.*



(a) Tree 1



(b) Tree 2



(c) Tree 3

Figure 4: Random Forest

Random Forest Given algorithm 2 we construct a solution (Algorithm 3) for an ensemble of decision trees. For each tree in the forest, we obtain a "proposal" for a small change in x such that the tree classifies it in the appropriate label. Now we consider the application of the proposed perturbation to x, and store the results (Lines 2–3). For each such result (Line 4), we re-evaluate all trees in the forest, and count the number of trees returning l. We pick the perturbation that maximizes this number.

Note that this is a greedy approach, and the optimality of the obtained perturbation is not guaranteed, as expected from Proposition 4.1. The effectiveness of this approach is shown experimentally in Section 8.

Algorithm 3: Random Forest - Perturb

```
input  : Forest $F$, point $x$, label $l$, constraints $C_x$
output: Minimal changes to $x$ so it will be tagged as $l$ according to $F$
1 $Xs = \{\}$;
2 foreach $T \in F.trees$ do
3     $Xs.add(T.Perturb(x, l))$;
4 return $\arg\max_{x' \in Xs} |\{T \in F.trees : T(x') = l\}|$;
```

Example 5.6 *Consider the random forest consisting of the* 3 *trees depicted in Figure 4. Further consider an input vector* $x = [7, 2, 2]$*, leading the decision to the highlighted leaves labeled as* 0*,* 1*, and* 0*. Suppose we are interested in the minimal changes to* x *such that the model will classify it as* 2*. For each of the trees, we will execute Algorithm 2 that will return candidate vectors labeled as* 2 *by the corresponding tree. In this case, the candidates will be* $[\underline{5}, 2, 2]$*,* $[\underline{6}, 2, 2]$*,* $[7, \underline{4}, 2]$*. Each of these candidates will be evaluated on the remaining trees, and the candidate that is agreed upon by the majority of the trees will be chosen. In this case,* $[5, 2, 2]$ *will be classified as* 2 *by both 4a and 4a, while each of* $[6, 2, 2]$ *and* $[7, 3, 2]$ *will each be classified as* 2 *by a single tree. Thus,* $x' = [5, 2, 2]$ *will be returned.*

Returning to Algorithm 1, its iterative process will continue and once again the Perturb function of the Random Forest model will be invoked (assuming constraints are satisfied). Again, for each tree we will examine what changes to $[5, 2, 2]$ *will lead to the label* 2. *For both Tree 1 and Tree 2, no change is required, while for Tree 3 we will obtain the candidate* $x' = [5, \underline{4}, 2]$ *which is agreed upon by all the trees in the forest. Eventually* $[5, 4, 2]$ *will be returned, and it is tagged by* 2 *with confidence of* 100%.

5.3 Neural Network

Neural networks are composed from neurons organized in multiple layers. The input of each neuron is the output of neurons from the previous layer. Given its input $x_1, ..., x_n$ it performs a linear computation, i.e returns $b + w_1 x_1 + ...w_n x_n$ followed by a nonlinear activation. Combined, all the neurons in the network can learn complex and non-linear functions. It is typically extremely hard to interpret the cause for a neural net decision as it requires us to understand the interactions between many neurons in the network. However, there are known techniques to measure the effect of each individual feature. In particular, using backpropagation, we can fix the values of all except for one feature x_i, and examine the effect of changes in x_i over the output of the network (for these particular values of the other features). We denote the obtained value (called the gradient of the x_i with respect to the network n) by $\frac{\partial n}{\partial x_i}$. We then perturb the input by changing each of its features in a direction that should adjust the output slightly towards the desired direction as follows: $x' = (x_1 + \epsilon \frac{\partial n}{\partial x_1}, ..., x_d + \epsilon \frac{\partial n}{\partial x_d})$, where ϵ controls the magnitude of the change. Due to the highly non-linear structure of neural networks, the gradient of each feature changes rapidly, hence small ϵ values should be used.

We note that finding a small perturbation in a desired direction has been extensively studied in the context of adversarial examples [22]. It is known that the perturbation themselves are not indicative, and can not be interpretable or give us indication on informative changes to the original vector. However, as we will show, by integrating the perturbation with projection over a constrained space, we can understand which are the meaningful modifications to be presented as explanations.

6 Project Implementation

Recall that algorithm 1 includes two repeating steps, namely perturb and project. The perturb step outputs a modification $x' + \Delta x'$ to the previous point x', but it does not account for the constraints C_x, and so it is possible that $x' + \Delta x' \notin C_x$. In the project step we wish to apply minimal changes to $x' + \Delta x'$ so that the result will satisfy the constraints. Formally, for a point $x' \notin C_x$ we aim for $project(x', C_x)$ to return $\arg\min_{x'' \in C_x} \|x'' - x'\|$, i.e. return the closest point in C_x.

We explain multiple possible implementations of the projection step, for different types of constraints; then we discuss an optimization that performs perturbation and projection in a single step where possible.

6.1 Implementing Project

We start by the case of linear constraints, then discuss extensions to other types.

Linear Constraints Many practical constraints may be captured through linear inequalities. In this case, the projection step amounts to solving a quadratic programming problem. Formally, we say that C_x defines *linear constraints* if $C_x = \{v \in \mathbb{R}^d \mid A_x v \leq b\}$, where $A_x \in \mathbb{R}^{m \times d}$ is a matrix, derived from the original input x, defining the parameters for m inequality equations; $b \in \mathbb{R}^m$ defines the thresholds for the inequality equations. Given linear constraints C_x, and a point $x' \notin C_x$, we then have that $project(x', C_x) = x^*$, where x^* is the solution to the following quadratic programming problem:

$$\begin{aligned} \text{Minimize} \quad & \|x^* - x'\| \\ \text{Subject to} \quad & A_x x^* \leq b \end{aligned} \tag{3}$$

Based on Hilbert projection theorem, the problem is guaranteed to have a unique solution, but finding the solution for general quadratic programming is NP-hard. However, for our case, where we are interested in minimizing $\|x^* - x'\|$ the problem is much simpler, and the ellipsoid method [24] solves the problem in polynomial time.

Example 6.1 *Let us revisit example 5.3, in which got as an input* $x = [2, 3, 6]$ *which was modified by the perturb step to be* $x' = [4, 3, 6]$. *Assume additionally that the constraints require that the change in* $x[0]$ *will be smaller or equal to the change in* $x[1]$, *i.e.* $C_x = \{v \in \mathbb{R}^3 \mid (v[0] - 2) - (v[1] - 3) \leq 0\}$. *Simplifying the equation, it can be represented as* $C_x = \{v \in \mathbb{R}^3 \mid A_x v \leq b\}$ *where* $A_x = [1, -1, 0]$ *and* $b = [-1]$. *The perturbed vector* x' *does not satisfy the constraints, as* $4 - 3 \not\leq -1$. *Thus we will project it over* C_x *so that the changes will be legitimate. Finding the projection result amounts to solving the following optimization problem:*

$$\begin{aligned} \textit{Minimize} \quad & \sqrt{(x^*[0] - 4)^2 + (x^*[1] - 3)^2 + (x^*[2] - 6)^2} \\ \textit{Subject to} \quad & x^*[0] - x^*[1] \leq -1 \end{aligned} \tag{4}$$

A solution to this problem is $x^* = [3, 4, 6]$; *note that* $x^* \in C_x$ *since* $3 - 4 \leq -1$, *i.e.* x^* *is obtained by legitimate modifications to the original point* x. *However, the projected point is no longer classified as* 2 *by the forest, and so a second round of perturb and project will occur. Only after the second iteration a result in* C_x *that is classified as* 2 *will be obtained.*

Convex non-linear constraints Many natural constraints may be formulated as linear ones, including all of our examples thus far. Still, there are natural non-linear constraints such as ones requiring that the modifications are all in a particular circle around a given point. Non-linear but convex constraints can be represented as a set of linear equalities (based on A_x) and convex inequalities (based on $f_{x,i}$):

$$A_x y = b, \quad f_{x,i}(y) \leq 0,\ i = 1, ..., m,$$

and finding the projection of x' on C_x solving the convex non-linear optimization problem (see [6]):

$$\begin{aligned} \text{Minimize} \quad & \|x^* - x'\| \\ \text{Subject to} \quad & A_x x^* = b \\ f_{x,i}(x^*) \leq 0,\ & i = 1, ..., m \end{aligned} \tag{5}$$

For convex non-linear optimization problem, project still guaranteed to have a unique solution. The optimization problem has been studied thoroughly and multiple efficient approximation algorithms have been proposed (see [7]). Naturally, for particular constraints, one may design specialized projection functions and avoid solving the above optimization problem. For example, if we know in advance that C_x is the Unit Ball then we have that $project(x', C_x) = \frac{x'}{\max\{1, \|x'\|\}}$. Naturally, constraints may in principle also be non-convex; such domain-tailored projection functions will be required in such cases.

6.2 Optimization

A natural optimization to consider is to avoid the separation between the perturbation and projection steps, and instead already in the perturbation step make sure that the result satisfies the constraints. We next discuss this optimization for the case of linear constraints and for the different ML models.

Random Forests We perform the optimization for each individual tree of the Random Forest in Algorithm 4. The idea is to encode in a single quadratic program both the perturbation and projection steps. The algorithm operates in a similar manner to Algorithm 2, with the difference being that for each path, instead of applying immediate updates to the point, we store the conditions that need to be satisfied (Line 7). For each leaf (with label l), the accumulated conditions, together with the constraints, form a quadratic programming instance. We solve it (Line 9), and obtain for each leaf its locally-optimal perturbation that satisfies the constraints. The solutions obtained for the different leaves are accumulated (Line 10), and eventually (Line 11) we choose the overall best solution (i.e. one with minimal distance).

Example 6.2 *Again consider the decision tree illustrated in Figure 3, and the setting from Examples 5.3 and 6.1. The input $x = [2, 3, 6]$ is classified as 0, and the goal is to find $x^* \in C_x = \{v \in \mathbb{R}^3 \mid v[0] - v[1] \leq -1\}$ labelled as 2, such that $\|x - x^*\|$ is minimal. For each 2-labeled leaf, Algorithm 4 will accumulate the conditions in the path from it up to the tree root. Eventually, a quadratic*

Algorithm 4: Decision Tree - Perturb & Project

```
input  : Decision Tree $T$, point $x$, label $l$, constraints $C_x$
output: Minimal legitimate changes to $x$ according to $C_x$ so it will be
        tagged as $l$ according to $T$
1  $Xs = \{\}$;
2  foreach $leaf \in T.leaves$ do
3      if $leaf.label() = l$ then
4          $Conditions = \{\}$;
5          $parent = leaf.parent()$;
6          while $parent \neq Null$ do
7              $Conditions.add(parent.condition())$;
8              $parent = parent.parent()$;
9          $x' = solve(x, Conditions \cup C_x)$;
10         $Xs.add(x')$;
11 return $\arg\min_{x' \in Xs} \|x - x'\|$;
```

programming problem based on both C_x and the accumulated conditions will be solved. For example, for the rightmost leaf labeled as 2, the following conditions will be accumulated: $\{x^*[0] \leq 6 \wedge x^*[1] \geq 5 \wedge x^*[0] \geq 4\}$ (in case the tree path "goes right", we use the condition that $feature \geq threshold+1$). The following optimization problem will then be solved:

$$\begin{aligned} \textit{Minimize} \quad & \sqrt{(x^*[0]-2)^2 + (x^*[1]-3)^2 + (x^*[2]-6)^2} \\ \textit{Subject to} \quad & x^*[0] - x^*[1] \leq -1 \\ & x^*[0] \geq 4 \\ & x^*[1] \geq 5 \\ & x^*[0] \leq 6 \end{aligned} \tag{6}$$

Solving this minimization problem results in $[\underline{4}, \underline{5}, 6]$. Note that combining the perturbation and projection into a single operation has resulted in this case in obtaining a result already in the first iteration, unlike the non-optimized version that required 2 iterations.

The optimized algorithm can then be used for Random Forests, by simply applying it to each of the forest trees. Namely, the only change to Algorithm 3 is in Line 3, which now uses the optimized version for its decision trees. This guarantees that the perturbation computed for the entire Random Forest will also satisfy the constraints, as it leads to a result returned by one of its tress.

For other models, such as Neural Networks, the implementation of such optimization remains an open problem. The difficulty is that the optimization requires understanding of interactions between different features in order to find a perturbation that satisfies the constraints. For the case of random forests, we were able to find a change in the input so that the constraints continue to hold, since we could infer from the model structure the entire group of features that need to be modified. In contrast, for Neural Networks, we can infer the impact of each individual feature by computing the gradient, but the effect of a change of multiple features remains unknown.

7 Use-Cases

We have now presented our approach and algorithms for explaining the results of ML models. In this section we demonstrate the use of our solution to derive insights for commonly used models and real-life data.

Example 7.1 *We have trained a simple Random Forest with* 50 *trees at maximal depth of* 6 *over the* `MNIST` *dataset [34]. The model had mediocre performance of* 89% *accuracy. Figure 5 presents in its upper row an image of a handwritten* 4 *that was erroneously classified by the model as* 9 *with* 39% *confidence. At first, it is unclear what caused the model to believe the image is a handwritten* 9*. Two examples for explanations of the "*4*" decision are shown in the middle row. On the left, there were no constraints on the allowed modifications, and we can see that* 32 *pixels have changed (blue indicates addition of pixels, and red indicates deletions). Additionally, we can see the modified digit and the updated model decision (classified correctly as* 4 *with* 60% *confidence). Some of the changes are not even close to the digit, and do not serve as a convincing explanation. On the right we present the result of the minimal sequence of pixel deletions (the constraint is that only deletions are allowed) that lead to the tag "*4*". We can see that* 10 *out of the* 13 *deletions were of pixels in the top part of the digit, providing some insight on model decision.*

Further insight is gained by looking at all *images of* 4 *in the dataset that were tagged as* 9 *by our model. We have executed our algorithm (allowing only deletions) for each of these images individually, and then computed the average pixel-by-pixel value for the modifications (presented on the left of the bottom row). Now we can clearly see a trend: deleting pixels in the top area of the digit is generally most effective in leading the model to tag these images as* 4*. This means that the existence of these pixels, and in other words the "closed top" of the* 4 *digit is a characteristic explanation for its misclassification as* 9*. The correctness of this explanation can be further verified by looking at the average of* 4 *images that were tagged correctly by the random forest model (on the right of the bottom row), and observing that in those, the upper part of the digit is indeed open. On the other hand the top is close in the average of* 4 *images that were misclassified as* 9 *(middle of the bottom row).*

The next use-case shows insights we have derived with respect to a state-of-the-art Neural Network model, namely Google `Inception-v3` [45].

Example 7.2 *We have used a collection of flower images from* `ImageNet` *[31] and classified them using Google's* `Inception-v3` *model. We observed that a large portion of the images were classified by the model as a Bee in its top-5 labels. Note that none of these images contain a bee, and so the Bee tag*

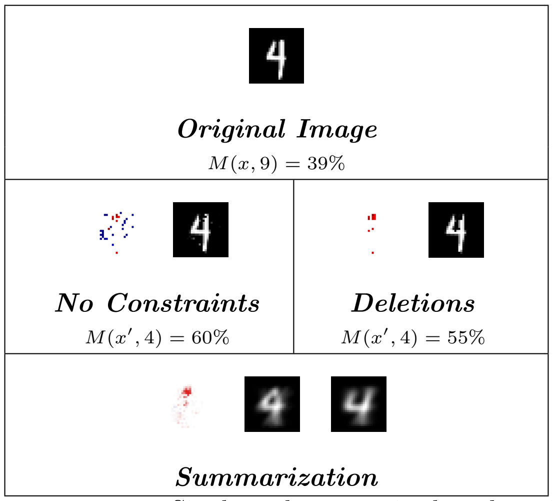


Figure 5: `MNIST` - Single and summarized explanations

is incorrect. For those images we ran our algorithm using a constraint that restricts modification of more than 10 *super-pixels, aiming at explaining the Bee label. Figure 6a presents* 4 *images of flowers with their Bee explanation; observe that our explanations are generally focused on the center of the flower. Looking at the* `ImageNet` *dataset, that was used to train the model, we see that many Bee-tagged images include a bee on top of a flower. Indeed, when running our algorithm to explain the tagging of these images (see Figure 6b), it correctly identifies the bees in the flower center. Our explanations indicate that the model could be improved, e.g. by adding more training data including images of bees without flowers.*

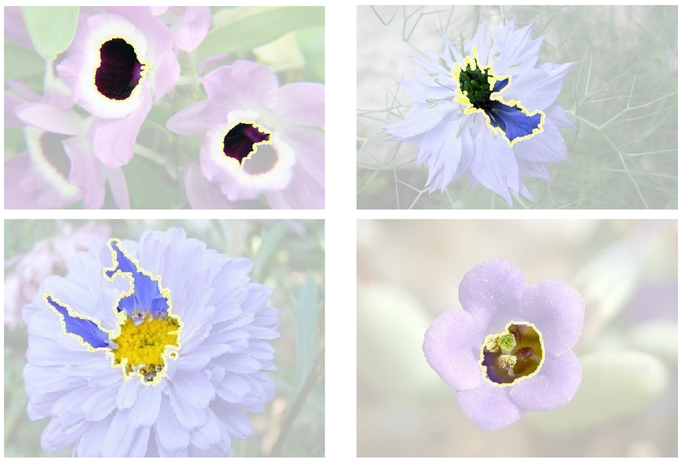
8 Experiments

We have conducted two kinds of experiments: a user study to assess the quality of presented explanations, and automated experiments to evaluate computational facets (the source code is available in [16]).

8.1 User Study

To evaluate the quality of our explanations we have compared them to state of the art technique for ML explanations, i.e. `LIME` [41], for the task of explaining results obtained by `Inception-v3` [45]. Our study included 30 images from `ImageNet` [14]; images were drawn randomly from the cases that were accurately identified by the model. For each of those images, 4 different explanations generated by the following techniques were shown:

- **LIME** - Explanation of top 10 features generated by `LIME` after training an explanation model using 1000 examples. We have used the default


(a) Flowers with Bee explanations

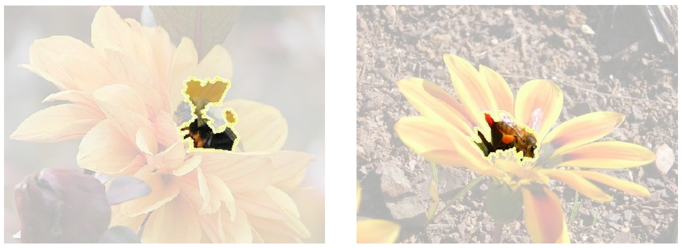

(b) Bees explanation

Figure 6: `Inception-v3` - Bee summarization

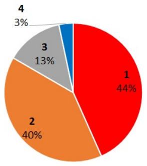
parameters of the algorithm as given in LIME's open-source package[35] for image explanations.

- **Adversarial** - Adversarial examples were generated by running our algorithm without constraints.
- **Limited Pixels** - Our algorithm, with a constraint limiting the change to only 0.1% of the image pixels.
- **Limited Superpixels** - Our algorithm allowing to change only pixels in 10 of the image super-pixels.

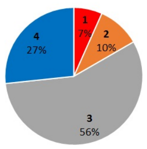
For the last 3 techniques we generated both supporting and refuting explanations, by tuning β to be $\frac{m(x)+5}{6}$ and $\frac{m(x)}{6}$ respectively (where $m(x)$ is the initial model confidence in the explained label). In cases the algorithm had not achieved the required β in 30 iterations, we halted and used the obtained result. The presented explanation is the average between both supporting and refuting explanations.

The user study was conducted using `CrowdFlower` platform. Since we aimed at examining the usefulness of explanations to data scientists, we have not released the task to anonymous users but rather manually recruited data scientists for the evaluation, asking them to rank the quality of each explanation from 1 (*Very Bad*) to 5 (*Very Good*). To avoid a potential bias, the users were not informed of which of the explanations were generated by which system. The task, as given to the users, is available in [11].

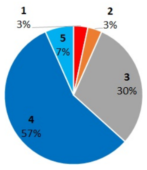
Each image was evaluated by 8–15 different data scientists and we averaged their ranking for each of the techniques. The results are shown in Figures 7a, 7b, 7c and 7d. Clearly, trying to explain the model using adversarial examples leads to very poor results, as 44% of the images got *Very Bad* rating using this approach. For `LIME`, much better results have been achieved where for 56% of the images, the explanations were given the neutral rating of 3. Our results indicate that by leveraging knowledge on the problem domain and imposing constraints, we can significantly improve user satisfaction. In particular, using *Limited Pixels* and *Superpixels* constraints, the explanations for 64% and 73% of the cases (respectively) have been given a positive rating (4 or 5).



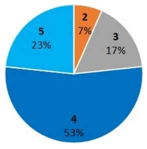
(a) Adversarial



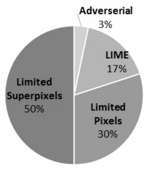
(b) `LIME`



(c) Limited Pixels



(d) Limited Superpixels



(e) Best Rating

Figure 7: User-Study Results

We also report for each image which technique got the highest average rating. This measurement shows the same trend: for 15 out of the 30 images, the *Limited Superpixels* explanations received the highest average rating. This was the case for *Limited Pixels* for 9 images, and explanations computed by `LIME` have received the highest score for 5 images. The adversarial (no constraints) solution has received the highest score for a single image.

In figure 8 we illustrate multiple explanations obtained by the *Limited Su-*

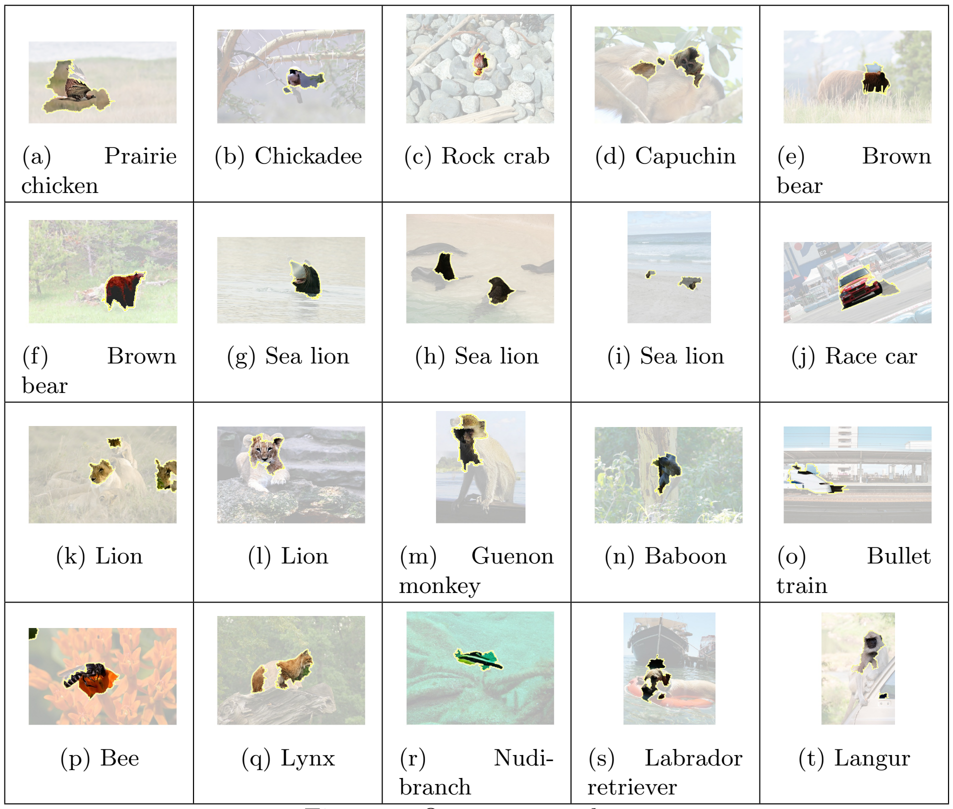

(a) Prairie chicken (b) Chickadee (c) Rock crab (d) Capuchin (e) Brown bear (f) Brown bear (g) Sea lion (h) Sea lion (i) Sea lion (j) Race car (k) Lion (l) Lion (m) Guenon monkey (n) Baboon (o) Bullet train (p) Bee (q) Lynx (r) Nudibranch (s) Labrador retriever (t) Langur

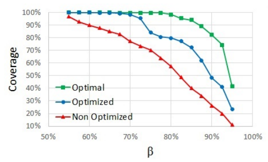
Figure 8: Output examples

perpixels execution over the images from the user study. Observe that indeed, the explanations mostly focus on areas that contain meaningful information for the model prediction.

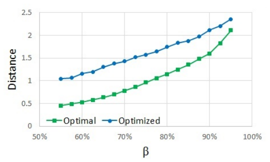
8.2 Automated Experiments

We have further studied computational aspects of our algorithms: how often does the algorithm achieve the given confidence? how many iterations are required for that? how far is the achieved result from the input? How does this compare with the optimal result (when available)? What is the effect of our optimization in this respect?

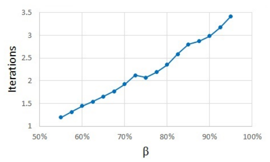
Random Forest We have generated a synthetic dataset of $1K$ data points using `sklearn` *make_classification* method having 2 classes with 5 features (3 informative and 2 redundant) with $class\_sep = 0.5$. Over this dataset we have trained a Random Forest classifier with 50 trees having maximal depth of 5, the classifier reached 88% accuracy over a test set. Using our domain knowledge on the dataset creation, which contains only 3 informative features, we have



(a) RF Coverage



(b) RF Distance



(c) RF Iterations

Figure 9: Random Forest Experiments

devised a constraint that allows to change up to 2 of the input features. Then, we looked at 300 cases that the model misclassified. Since the input space is of low dimension, we can compute the optimal change using a brute force solution, and then compared three solutions:

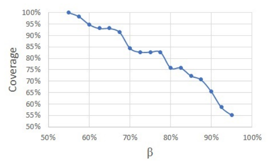
- **Optimal** - Exact brute-force solution to the NP-Hard problem of finding the optimal label supporting change.
- **Optimized** - The result obtained by the optimization for Random Forest model, combining the perturb and project into a single step.
- **Non-Optimized** - The result obtained by the non-optimized version of the Random Forest model, where the perturb and project are two different steps.

First, we evaluated for each confidence level, the algorithms coverage – the percentage out of the 300 cases where we were able to reach a given β, limited by 30 iterations. In 9a we observe the improvement that the optimized version achieves over the non-optimized one. The highest difference was for $\beta = 85\%$ where the optimized returned a result for 72% of the cases and the non optimized version was able to answer only 40% of the cases. Additionally, when both algorithms found an answer they were (with the exception of 2 cases) the same one. Thus, the remaining figures include only the optimized version.

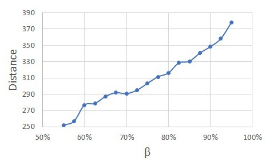
In Figure 9b we compare the $L2$ distance, from the input vector, of the results returned by optimized algorithm and those of the optimal solution. Results are presented only for the cases where the optimized algorithm was able to return a valid solution. We observe that for small β, the difference is greater than for large β values. This is because for small β values there are many more candidates that satisfy the constraints and cause the model to change its decision to the appropriate confidence level. On the other hand for high β values there are only few possible candidates, and whenever the optimized algorithm returns an answer it is very close to the optimal answer.

Last, in figure 9c we plot the average number of iterations incurred by our optimized algorithm until returning an answer, as a function of β. Again, results are shown only for the cases where the algorithm has succeeded in returning valid results. As expected, requiring higher confidence results in more iterations made by the algorithm. On average, the optimized algorithm has required less than 4 iterations for all confidence levels. Note the small drop in iteration number for

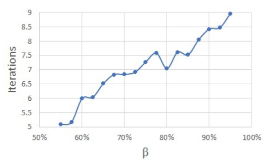
$\beta = 75\%$ that is explained by the large drop in coverage at the same confidence level. Since the average number of iterations is measured only on successful cases, many cases that required more iterations for $\beta < 75\%$ were removed from the examination at that point.



(a) NN Coverage



(b) NN Distance



(c) NN Iterations

Figure 10: NN Experiments

Neural Networks In this experiment, we evaluated the performance of our algorithm for the Neural Network model. Again we used `Inception-v3` model over 60 images from `ImageNet`. Observing in the user study that the superpixel constraint yields the best explanations, we continued our evaluation using this constraint. The images were chosen based on the model prediction: we took images where the model was uncertain in regards to the prediction and looked at its top-3 predictions. For each prediction we applied our algorithm with the *Limited Superpixel* constraint and evaluated its performance (the optimal solution was unavailable due to the complexity of the problem).

In figure 10 we show that the performance trends are similar to those observed in our Random Forest experiment. As we require higher confidence, the coverage decreases down to 55% with $\beta = 0.95$. The $l2$ distance increases as a function of β but remains below 400. The number of iterations increases in an almost linear fashion.

9 Conclusion and Limitations

We have proposed a novel approach to explaining individual classifications of Machine Learning models. Our explanations are based on repeated perturbation and projection of the input, the former to "push" it to the label of interest, and the latter to enforce that the modified input "makes sense" in the context of the problem domain. We demonstrate the effectiveness of our approch for multiple prominent ML models and real-life data.

The main limitation of our approach is that it requires domain knowledge, unlike the black-box approach of `LIME`. We thus view our solution as complementary to that of `LIME`, and envision it to be used primarily by data scientists. In this context, our solution has the advantage of being purely algorithmic (rather than applying Machine Learning itself), and thus "compatible" with explanation efforts for data management (see Section 2). In future work we intend to further explore this connection. In particular, we will study applications of our technique in cases where the constraints originate from the Database.

References

[1] V. Behzadan and A. Munir. Vulnerability of deep reinforcement learning to policy induction attacks. In *International Conference on Machine Learning and Data Mining in Pattern Recognition*, pages 262–275, 2017.

[2] L. Bertossi. Database repairing and consistent query answering. *Synthesis Lectures on Data Management*, 3(5):1–121, 2011.

[3] L. Bertossi, S. Kolahi, and L. V. Lakshmanan. Data cleaning and query answering with matching dependencies and matching functions. *Theory of Computing Systems*, 52(3):441–482, 2013.

[4] N. Bidoit, M. Herschel, and A. Tzompanaki. Efficient computation of polynomial explanations of why-not questions. In *CIKM*, pages 713–722. ACM, 2015.

[5] B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. Šrndić, P. Laskov, G. Giacinto, and F. Roli. Evasion attacks against machine learning at test time. In *Joint European conference on machine learning and knowledge discovery in databases*, pages 387–402, 2013.

[6] S. Boyd and L. Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[7] S. Bubeck et al. Convex optimization: Algorithms and complexity. *Foundations and Trends® in Machine Learning*, 2015.

[8] P. Buneman, S. Khanna, and W. Tan. Why and where: A characterization of data provenance. In *ICDT*, 2001.

[9] A. Chapman and H. V. Jagadish. Why not? In *SIGMOD*, pages 523–534, 2009.

[10] J. Cheney, L. Chiticariu, and W. C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 2009.

[11] User study on crowdflower. `https://tinyurl.com/yaw8rwoz`.

[12] Y. Cui, J. Widom, and J. L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.*, 2000.

[13] N. Dalvi, P. Domingos, S. Sanghai, D. Verma, et al. Adversarial classification. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 99–108, 2004.

[14] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[15] F. Doshi-Velez and B. Kim. Towards a rigorous science of interpretable machine learning. 2017.

[16] `https://github.com/navefr/Explaining_White_box_Classifications_to_Data_Scientists`.

[17] W. Fan and F. Geerts. Foundations of data quality management. *Synthesis Lectures on Data Management*, 4(5):1–217, 2012.

[18] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. The llunatic data-cleaning framework. *Proceedings of the VLDB Endowment*, 2013.

[19] B. Glavic, J. Siddique, P. Andritsos, and R. J. Miller. Provenance for data mining. In *Proceedings of the 5th USENIX Workshop on the Theory and Practice of Provenance*, 2013.

[20] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[21] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[22] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and Harnessing Adversarial Examples. *ArXiv e-prints*, 2014.

[23] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, 2007.

[24] M. Grötschel, L. Lovász, and A. Schrijver. *Geometric algorithms and combinatorial optimization*, volume 2. Springer Science & Business Media, 2012.

[25] T. K. Ho. Random decision forests. In *Document analysis and recognition, 1995., proceedings of the third international conference on*, pages 278–282, 1995.

[26] T. K. Ho. The random subspace method for constructing decision forests. *IEEE transactions on pattern analysis and machine intelligence*, 20(8):832–844, 1998.

[27] J. Huysmans, B. Baesens, and J. Vanthienen. Using rule extraction to improve the comprehensibility of predictive models. 2006.

[28] J. Huysmans, K. Dejaeger, C. Mues, J. Vanthienen, and B. Baesens. An empirical evaluation of the comprehensibility of decision table, tree and rule based predictive models. *Decision Support Systems*, 51:141–154, 2011.

[29] B. Kim. *Interactive and interpretable machine learning models for human machine collaboration.* PhD thesis, Massachusetts Institute of Technology, 2015.

[30] P. W. Koh and P. Liang. Understanding black-box predictions via influence functions. *arXiv preprint arXiv:1703.04730*, 2017.

[31] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[32] A. Kurakin, I. Goodfellow, and S. Bengio. Adversarial examples in the physical world. 2016.

[33] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[34] Y. LeCun, C. Cortes, and C. Burges. Mnist handwritten digit database. *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist*, 2, 2010.

[35] `https://github.com/marcotcr/lime`.

[36] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[37] A. Meliou, W. Gatterbauer, J. Y. Halpern, C. Koch, K. F. Moore, and D. Suciu. Causality in databases. *IEEE Data Eng. Bull.*, 2010.

[38] A. Meliou, W. Gatterbauer, K. F. Moore, and D. Suciu. The complexity of causality and responsibility for query answers and non-answers. *Proceedings of the VLDB Endowment*, 2010.

[39] N. Papernot, P. McDaniel, and I. Goodfellow. Transferability in machine learning: from phenomena to black-box attacks using adversarial samples. *arXiv preprint arXiv:1605.07277*, 2016.

[40] F. Poursabzi-Sangdeh, D. G. Goldstein, J. M. Hofman, J. W. Vaughan, and H. Wallach. Manipulating and measuring model interpretability. In *NIPS 2017 Transparent and Interpretable Machine Learning in Safety Critical Environments Workshop*, 2017.

[41] M. T. Ribeiro, S. Singh, and C. Guestrin. Why should i trust you?: Explaining the predictions of any classifier. pages 1135–1144, 2016.

[42] A. D. Sarma, M. Theobald, and J. Widom. Exploiting lineage for confidence computation in uncertain and probabilistic databases. In *ICDE*, 2008.

[43] K. Simonyan, A. Vedaldi, and A. Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. 2013.

[44] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, et al. Going deeper with convolutions. 2015.

[45] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2818–2826, 2016.

[46] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. J. Goodfellow, and R. Fergus. Intriguing properties of neural networks. *CoRR*, 2013.

[47] A. Vellido, J. D. Martín-Guerrero, and P. J. Lisboa. Making machine learning models interpretable. In *ESANN*, volume 12, pages 163–172, 2012.

[48] E. Wu and S. Madden. Scorpion: Explaining away outliers in aggregate queries. *Proceedings of the VLDB Endowment*, 2013.