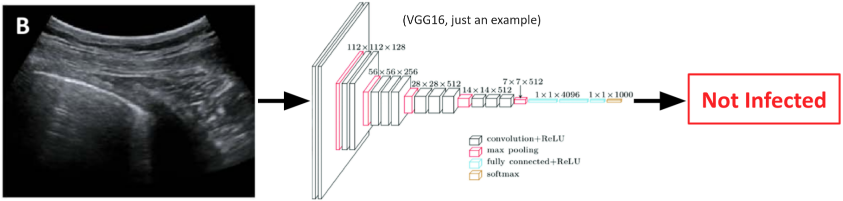
Explaining Black-box Models

Stefano Teso

Advanced Topics in Machine Learning and Optimization 22/23

Preliminaries

You need to be checked for COVID-19. The doctor takes a scan of your lungs and uses a state-of-the-art deep neural network to automatically compute a diagnosis. The model thinks that you are not infected.



Question: Would you trust the model's prediction?

As progress in AI is made – and hype grows – people are finding more and more ways of integrating machine learning models into applications.

- These include plenty of **high-stakes applications**:
  - Medical Diagnosis
  - Crime (e.g., predicting recidivism in convicts)
  - Credit Scoring (e.g., approving loan requests)
  - Surveillance (e.g., face recognition, profiling)
  - Hiring (e.g., ranking/filtering candidates)
  - ...

- Regulations from EU and other countries actually establish the **right to explanation**:

Example: *you apply for a* 50, 000 *eur loan. Unfortunately, your bank rejects your application. You have a right to know why it was rejected: was it your credit history or your age/gender/ethnicity?*

See https://en.wikipedia.org/wiki/Right_to_explanation

- Misbehaving models running unchecked might cause **all** sorts of trouble.
- (Although humans are not necessarily better and/or fairer than machines [Lin et al., 2020])

How can we check that models learned from data behave as expected?

As progress in AI is made – and hype grows – people are finding more and more ways of integrating machine learning models into applications.

- These include plenty of **high-stakes applications**:
  - Medical Diagnosis
  - Crime (e.g., predicting recidivism in convicts)
  - Credit Scoring (e.g., approving loan requests)
  - Surveillance (e.g., face recognition, profiling)
  - Hiring (e.g., ranking/filtering candidates)
  - . . .

- Regulations from EU and other countries actually establish the **right to explanation**:

Example: *you apply for a* 50,000 *eur loan. Unfortunately, your bank rejects your application. You have a right to know why it was rejected: was it your credit history or your age/gender/ethnicity?*

See https://en.wikipedia.org/wiki/Right_to_explanation

- Misbehaving models running unchecked might cause **all** sorts of trouble.
- (Although humans are not necessarily better and/or fairer than machines [Lin et al., 2020])

How can we check that models learned from data behave as expected?

As progress in AI is made – and hype grows – people are finding more and more ways of integrating machine learning models into applications.

■ These include plenty of **high-stakes applications**:

- Medical Diagnosis
- Crime (e.g., predicting recidivism in convicts)
- Credit Scoring (e.g., approving loan requests)
- Surveillance (e.g., face recognition, profiling)
- Hiring (e.g., ranking/filtering candidates)
- ...

■ Regulations from EU and other countries actually establish the **right to explanation**:

Example: *you apply for a* 50, 000 *eur loan. Unfortunately, your bank rejects your application. You have a right to know why it was rejected: was it your credit history or your age/gender/ethnicity?*

See https://en.wikipedia.org/wiki/Right_to_explanation

■ Misbehaving models running unchecked might cause **all** sorts of trouble.

■ (Although humans are not necessarily better and/or fairer than machines [Lin et al., 2020])

How can we check that models learned from data behave as expected?

As progress in AI is made – and hype grows – people are finding more and more ways of integrating machine learning models into applications.

- These include plenty of **high-stakes applications**:
  - Medical Diagnosis
  - Crime (e.g., predicting recidivism in convicts)
  - Credit Scoring (e.g., approving loan requests)
  - Surveillance (e.g., face recognition, profiling)
  - Hiring (e.g., ranking/filtering candidates)
  - . . .

- Regulations from EU and other countries actually establish the **right to explanation**:

Example: *you apply for a* 50, 000 *eur loan. Unfortunately, your bank rejects your application. You have a right to know why it was rejected: was it your credit history or your age/gender/ethnicity?*

See https://en.wikipedia.org/wiki/Right_to_explanation

- Misbehaving models running unchecked might cause **all** sorts of trouble.
- (Although humans are not necessarily better and/or fairer than machines [Lin et al., 2020])

How can we check that models learned from data behave as expected?

As progress in AI is made – and hype grows – people are finding more and more ways of integrating machine learning models into applications.

- These include plenty of **high-stakes applications**:
  - Medical Diagnosis
  - Crime (e.g., predicting recidivism in convicts)
  - Credit Scoring (e.g., approving loan requests)
  - Surveillance (e.g., face recognition, profiling)
  - Hiring (e.g., ranking/filtering candidates)
  - . . .

- Regulations from EU and other countries actually establish the **right to explanation**:

Example: *you apply for a* 50,000 *eur loan. Unfortunately, your bank rejects your application. You have a right to know why it was rejected: was it your credit history or your age/gender/ethnicity?*

See https://en.wikipedia.org/wiki/Right_to_explanation

- Misbehaving models running unchecked might cause **all** sorts of trouble.
- (Although humans are not necessarily better and/or fairer than machines [Lin et al., 2020])

How can we check that models learned from data behave as expected?

The “Clever Hans” Phenomenon

The models pick up (subtle) features of the training data that happen to **correlate** with the desired label, but are **not causally related** to it.

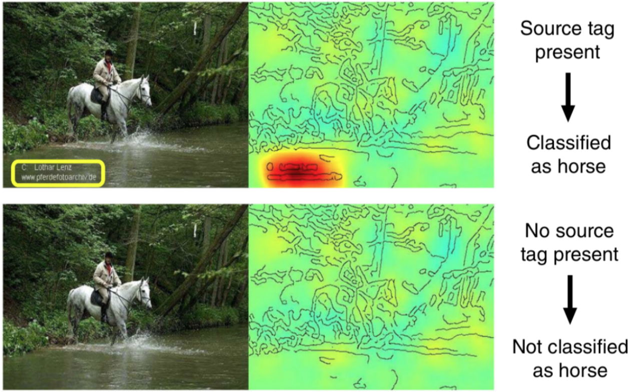
Confounders

If **watermarks** that correlate with the class “horse” appear in the **training set**:

- The model learns to rely on them to achieve low training loss
- Butits predictions are useless if the confounder is not present

If they also appear in the **test data**, evaluation does not spot them

Horse-picture from Pascal VOC data set



Source tag present → Classified as horse

No source tag present → Not classified as horse

Credit [Lapuschkin et al., 2019]

Who is Clever Hans?

“*Clever Hans was a horse that was claimed to have performed arithmetic and other intellectual tasks.*”

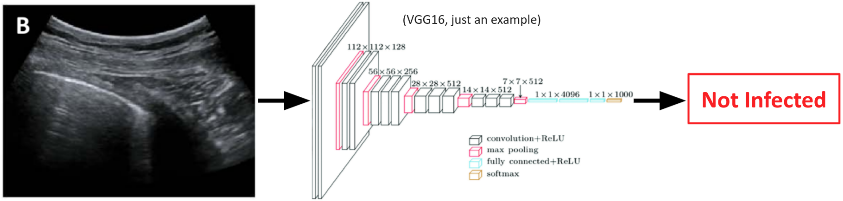
“*After a formal investigation in 1907, psychologist Oskar Pfungst demonstrated that the horse was not actually performing these mental tasks, but was watching the reactions of his trainer.*”

Hans managed to picked up on **confounders**

(This is actually quite an impressive feat for a horse!)

Credit: `en.wikipedia.org/wiki/Clever_Hans`

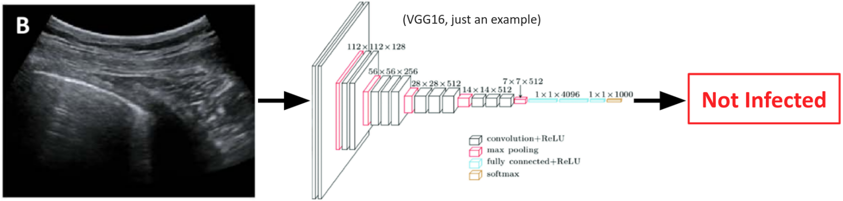
You need to be checked for COVID-19. The doctor takes a scan of your lungs and uses a state-of-the-art deep neural network to automatically compute a diagnosis. The model thinks that you are not infected.



Question: Would you trust the model's prediction?

Presumably, you'll want to know **whether the model exhibits C-H behavior first** ;-)

You need to be checked for COVID-19. The doctor takes a scan of your lungs and uses a state-of-the-art deep neural network to automatically compute a diagnosis. The model thinks that you are not infected.



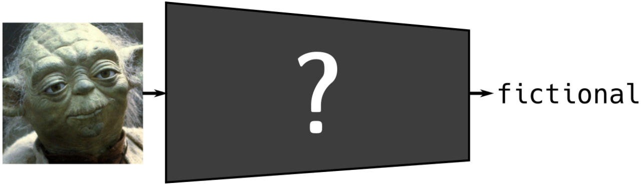
(VGG16, just an example)

Not Infected

Question: Would you trust the model's prediction?

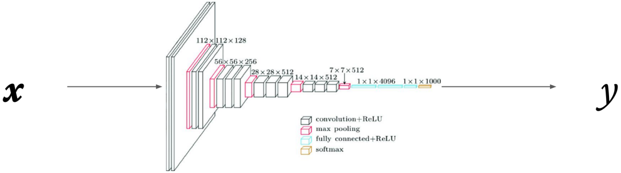
Presumably, you'll want to know **whether the model exhibits C-H behavior first** ;-)

A **black-box** classifier $f : \mathbb{R}^d \to [c]$ (where $[c]$ is a shorthand for $\{1, \ldots, c\}$) should look like this:



Examples: neural networks, non-linear support vector machines, random forests, ...

However, this is **not quite true**. A CNN $f : \mathbb{R}^d \to [c]$ looks like this:



It is **not** quite a black box, is it?

True: the functional form and parameters are **known**. However, it is hard to [Lipton, 2018]:

- Break down the computation into an **interpretable sequence of simple steps**.
- Allocate **responsibility of decisions** to individual weights, inputs, features, examples, . . .

This is necessary to answer “why” questions and spot C-H behavior.

Not all classifiers are black-box!

Example: Linear Model to Identify Ripe Papayas

- Does a **papaya** x taste good? (Here $\mathbb{1}(\mathit{cond})$ is 1 if *cond* is true and 0 otherwise.)

- Consider a **linear classifier**:

$$\begin{aligned} f(\mathrm{x}) = \mathrm{sign}\big(& 1.3 \cdot \mathbb{1}(\mathrm{x}\text{ pulp is orange}) + \\ & 0.7 \cdot \mathbb{1}(\mathrm{x}\text{ skin is yellow}) + \\ & \mathbf{0} \cdot \mathbb{1}(\mathrm{x}\text{ is round}) + \\ & -0.5 \cdot \mathbb{1}(\mathrm{x}\text{ skin is green}) + \\ & -2.3 \cdot \mathbb{1}(\mathrm{x}\text{ is moldy}) \big) \end{aligned}$$

Figure 1: A bunch of papaya fruits.

- It is easy to read off what attributes are “for” and “against” x being tasty **for the model**. This is possible because f implicitly encodes independence assumptions, e.g., that the shape of x is unrelated to its color.[1]

[1] When **explaining** a decision made by the model, **it is irrelevant whether these assumptions match how reality works**: we are explaining the model’s reasoning process, or equivalently its interpretation of how reality works, not reality itself!

- A **linear model** has the form:

$$f(\mathbf{x}) = \mathrm{sign}(\underbrace{\sum_i w_i x_i + w_0}_{\text{“score” of } \mathbf{x}})$$

In a **sparse linear model** $\mathbf{w} \in \mathbb{R}^d$ contains few non-zero entries [Tibshirani, 1996, Ustun and Rudin, 2016]

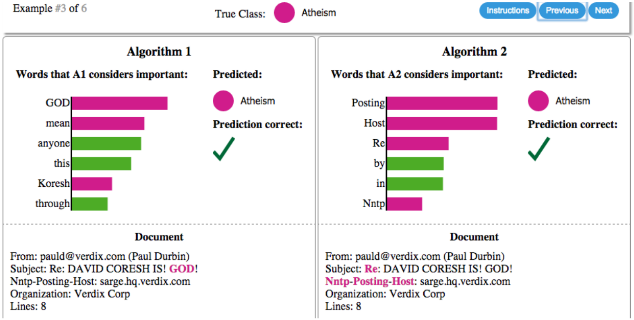
- This model assumes **conditional independence** among inputs: changing one does not change the others. This makes it “easy” to **attribute responsibility** to inputs by looking at their weights:[2]

  - $w_i > 0 \implies x_i$ correlates with, aka “votes for”, the positive class
  - $w_i < 0 \implies x_i$ anti-correlates with, aka “votes against”, the positive class
  - $w_i \approx 0 \implies x_i$ is irrelevant: changing it does not affect the outcome

- What matters is the weight of attribute i *relative* to all the other weights, so typically people normalize the weights s.t. they range in $[-1, 1]$.

[2] This is intuitively appealing but not “causal”. For instance, flipping a binary input x_i with a positive weight $w_i > 0$ is not guaranteed to change a negative prediction into a positive one. So intuitively x_i ought to be irrelevant. More on this later.

Example: Newsgroup Posts



Document

From: pauld@verdix.com (Paul Durbin)
Subject: Re: DAVID CORESH IS! **GOD**!
Nntp-Posting-Host: sarge.hq.verdix.com
Organization: Verdix Corp
Lines: 8

Document

From: pauld@verdix.com (Paul Durbin)
Subject: **Re**: DAVID CORESH IS! GOD!
Nntp-Posting-Host: sarge.hq.verdix.com
Organization: Verdix Corp
Lines: 8

Figure 2: Explaining individual predictions of competing classifiers trying to determine if a document is about "Christianity" or "Atheism". The bar chart represents the importance given to the **most relevant words**, also highlighted in the text. Color indicates which class the word contributes to (green for "Christianity", magenta for "Atheism".) [Ribeiro et al., 2016]

Caveats

- If many non-zero weights, it may be difficult to simulate the model's reasoning in your head.

Example

What if your linear model includes 1000 different weights, 80% of which are non-zero?

- The weights learned by the model depend on the available attributes. In other words, it's best not to make *absolute* judgments based on an arbitrary selection of attributes.

Example

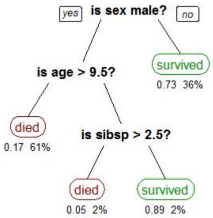
The importance of the attribute $\mathbb{1}(\mathbf{x}$ skin is yellow) depends on what the other attributes are!

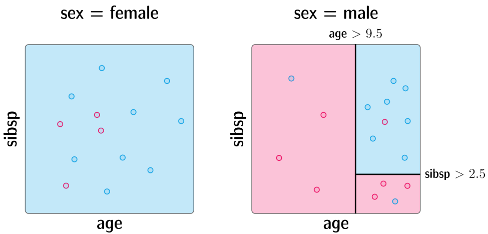
- If the other attributes include extra information like ruggedness or softness, color may become less important a factor.
- If they do not, then color may be the only important factor.

Hence, it is hard to tell how important color is in absolute terms.

Decision trees (DTs) encode sequences of conditions that partition the input space into geometrically simple regions.

Example: a DT for the Titanic survivors dataset is shown on the left. The variables include `age`, `sex`, passenger class (`class`), and number of siblings onboard (`sibsp`).





The decision surface of the DT is shown on the right for the two cases `sex` = `female` and `sex` = `male`. Nodes are conditions that sequentially **split** the input space into halves. Leaves correspond to rectangular regions of this space. (Training examples are represented as colored dots.)

■ Decision trees (DTs) that are **shallow** and rely on **interpretable variables** are transparent

Example

□ Given a prediction $y = f(x)$, it is easy to understand why such decision was taken by looking at **which nodes were traversed** during the inference procedure.

For instance, did $\mathtt{x} = (\mathtt{age} = 9, \mathtt{sex} = \mathtt{male}, \mathtt{sibsp} = 4)$ die because it had too few siblings on board (according to the model)?

□ The decision in each node only involves **exactly one interpretable variable** (e.g., age) and is quite easy to understand.

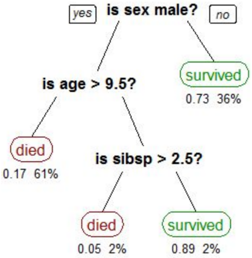


Figure 3: A shallow decision tree.

Note: this kind of models are called **simulatable** because they is easy to simulate in your own head.

Caveats

What if the data is **very complex**?

This will lead to a DT that is:

- **Wide**: it has a million small, very local leaves.
- **Deep**: in high dimensions, each of these leaves will have a large number of decision (i.e., sides) attached to it.

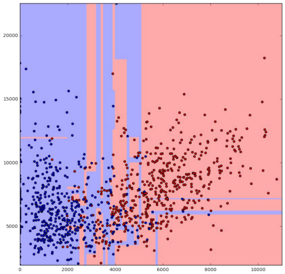
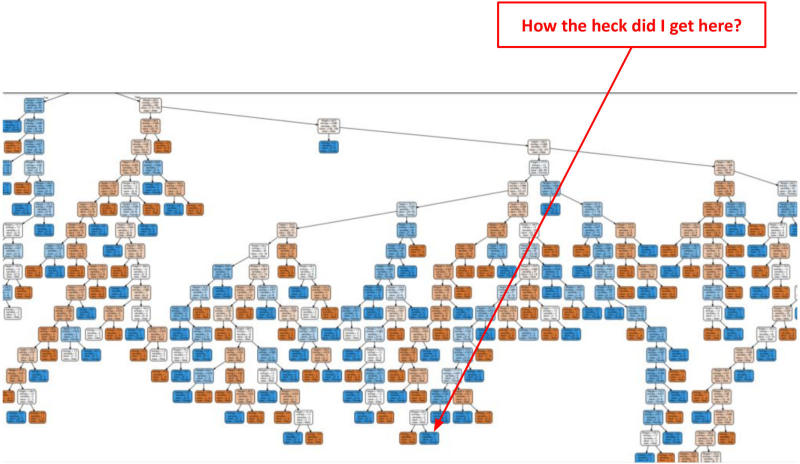


Figure 4: A shallow decision tree.

This makes the resulting tree much harder to simulate in your head & to understand in general

Caveats

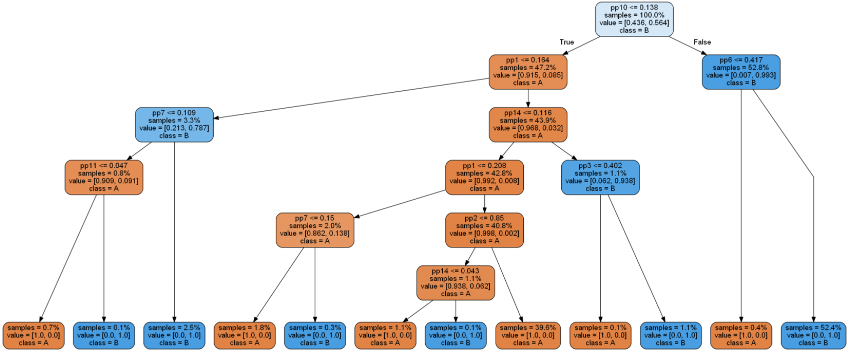
What if a transparent model is **really large**?



How the heck did I get here?

Caveats

What if a transparent model relies on **uninterpretable features**?



What the heck is **pp14**? (Credits: [Lipinski et al., 2020])

Topics in XAI

Factual explanations answer the question "why did model f output prediction y_0 for input x_0?"

- ...in terms of what **inputs** (e.g., pixels in an image) are responsible.
- ...in terms of what **high-level concepts** (e.g., objects in an image) are responsible.
- ...in terms of what **training examples** are responsible.

Counterfactual explanations answer the question "why did I get outcome y_0 instead of (a more desirable) outcome y_1?"

- ...in terms of what **inputs** should be changed to achieve the alternative outcome.

Global & Regional explanations answer "why" questions for more than a single decision.

- ...often in terms of simple rules, e.g., "if a papaya (*any* papaya!) is red then it does not taste good".

Topics in XAI

Factual explanations answer the question "why did model f output prediction y_0 for input x_0?"

- . . . in terms of what **inputs** (e.g., pixels in an image) are responsible.
- . . . in terms of what **high-level concepts** (e.g., objects in an image) are responsible.
- . . . in terms of what **training examples** are responsible.

Counterfactual explanations answer the question "why did I get outcome y_0 instead of (a more desirable) outcome y_1?"

- . . . in terms of what **inputs** should be changed to achieve the alternative outcome.

Global & Regional explanations answer "why" questions for more than a single decision.

- . . . often in terms of simple rules, e.g., "if a papaya (*any* papaya!) is red then it does not taste good".

Topics in XAI

Factual explanations answer the question "why did model f output prediction y_0 for input x_0?"

- ... in terms of what **inputs** (e.g., pixels in an image) are responsible.
- ... in terms of what **high-level concepts** (e.g., objects in an image) are responsible.
- ... in terms of what **training examples** are responsible.

Counterfactual explanations answer the question "why did I get outcome y_0 instead of (a more desirable) outcome y_1?"

- ... in terms of what **inputs** should be changed to achieve the alternative outcome.

Global & Regional explanations answer "why" questions for more than a single decision.

- ... often in terms of simple rules, e.g., "if a papaya (*any* papaya!) is red then it does not taste good".

Take-away

- White-box models are **no silver bullet**:
  - Transparent $\neq$ easy to understand: the model might be too complex or rely on black-box pieces
  - White-box models do **not** achieve SotA performance in many important applications (such as document classification!), while black-box models do.

Given their widespread use, it makses sense to **develop techniques for explaining black-box models**.

- This is what the rest of the slides are about ;-)

Note: another option is to develop "gray-box" models that combine white-box and black-box elements in a way that makes the model interpretable enough without giving up on performance even in demanding applications [Rudin, 2019]. This is still a few weeks away though.

Outline

Preliminaries

What is an explanation?

Attribute-level explanations

Example-level Explanations

Counterfactual Explanations

What is an explanation?

Explanations are studied in epistemology & philosophy of science. There are many **incompatible** but **complementary** schools of though:

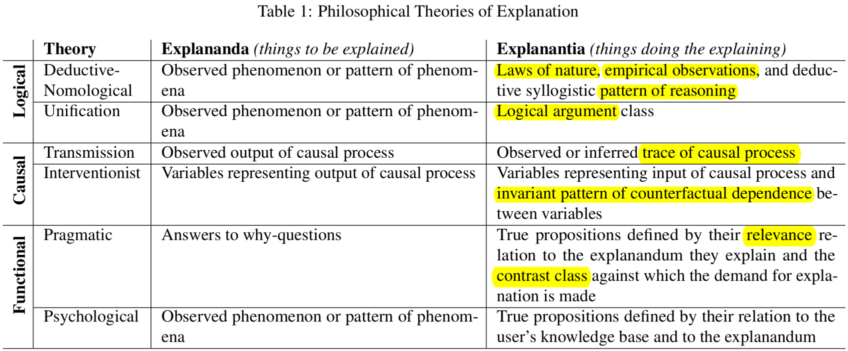


Table 1: Philosophical Theories of Explanation

	Theory	**Explananda** *(things to be explained)*	**Explanantia** *(things doing the explaining)*
Logical	Deductive-Nomological	Observed phenomenon or pattern of phenomena	Laws of nature, empirical observations, and deductive syllogistic pattern of reasoning
	Unification	Observed phenomenon or pattern of phenomena	Logical argument class
Causal	Transmission	Observed output of causal process	Observed or inferred trace of causal process
	Interventionist	Variables representing output of causal process	Variables representing input of causal process and invariant pattern of counterfactual dependence between variables
Functional	Pragmatic	Answers to why-questions	True propositions defined by their relevance relation to the explanandum they explain and the contrast class against which the demand for explanation is made
	Psychological	Observed phenomenon or pattern of phenomena	True propositions defined by their relation to the user's knowledge base and to the explanandum

Biased towards explanations *in science*. Most work focus on "**interventionist**" accounts.

In the **deductive-nomological** account, the explanation for a fact involves a combination of:

- Laws of nature
- Empirical observations
- A chain of deductive (*aka* logical) steps

Example

“*Why is the shadow 2m long?*”

“*Because the* **sun** *is at this position, and nuclear fusion emits photons, and photons get absorbed by the flagpole, and the geometry of space is such and such. Hence the cast shadow is 2m long*”

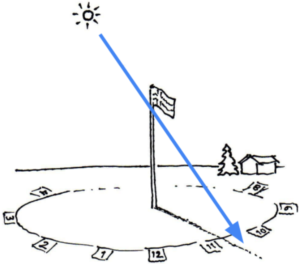


Figure 5: a flagpole and the Sun.

This is verbose but quite **intuitive**.

Problem: purely logical explanations do not take the **direction of causation** into account:

Example

“*Why is the sun at such and such position?*”

“*Because the* **shadow** *is at this position, and nuclear fusion emits photons, and photons get absorbed by the flagpole, and the geometry of space is such and such. Hence the sun is at this position.*”

This is a perfectly **valid** deductive-nomological explanation, but intuitively we **cannot accept** the shadow’s position to be a valid explanation for the sun’s motion!

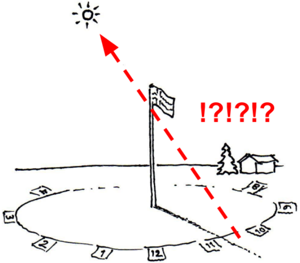


Figure 6: a flagpole and the Sun.

Interventions [Pearl, 2009]

Consider a **room with a thermostat**. Normally, the room's temperature and the value displayed by the thermostat are the same. Which value "causes" the other?

This can change if we **intervene** on the system:

- Changing **the room's temperature** (by, e.g., opening a window) *does* change the temperature displayed by the thermostat.
- Changing **the temperature displayed by the thermostat** (by, e.g., rewiring the circuits) *does not* change the temperature in the room!

In other words, interventions help to assess the **directionality of causation** – and they are exactly what was missing in the flagpole example.

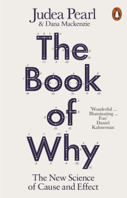
■ This is what people do in science and debugging: knocking out genes in mices or fixing the value of some variables in programs to compare the original and altered systems. Interventions are key to understand **how a mechanism works**.

Take-away

- **No unique definition** of explanation, even in philosophy
- Explaining machine learning models is still an open research question
- Non-causal accounts can be *incompatible* with our intuition of what makes a good explanation
- We will stick to explanations that have a somewhat **interventional** flavour

---

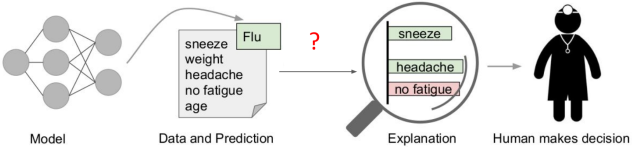
Note: causality is a fascinating topic. If you are interested, a good non-technical introduction is given by "The Book of Why" [Pearl and Mackenzie, 2018].


Judea Pearl
& Dana Mackenzie
The Book of Why
'Wonderful ... Illuminating ... Fun' Daniel Kahneman
The New Science of Cause and Effect

Attribute-level explanations

Attributions

Fix classifier $f : \mathbb{R}^d \to [c]$ and a decision $f(\mathbf{x}_0) = y_0$. What elements of $\mathbf{x}_0$ are **responsible** for this outcome?



Credit [Ribeiro et al., 2016]

Fix classifier $f : \mathbb{R}^d \to [c]$ and a decision $f(\mathbf{x}_0) = y_0$. What elements of $\mathbf{x}_0$ are **responsible** for this outcome?

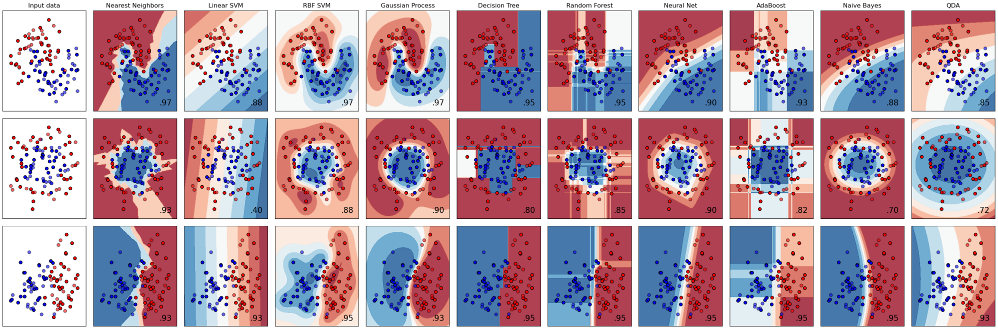
Recall that it is **easy** to answer this question for white-box models.

Idea:

1. Convert f to a white-box model g.
2. Extract an attribution map from g.

Seems easy enough. Does it always make sense?

All classifiers, including black-box ones, can be viewed as **decision surfaces**:



This view **abstracts away** unimportant details.

Model Translation

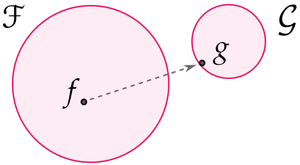
Given a classifier $f \in \mathcal{F}$ (e.g., a neural net), find a white-box classifier $g \in \mathcal{G}$ (e.g., a shallow decision tree) that approximates its predictions.

Translation can be viewed as a **projection** from $\mathcal{F}$ to $\mathcal{G}$:

$$\operatorname*{argmin}_{g \in \mathcal{G}} \quad d(f, g)$$

for an appropriate distance between functions $d(\cdot, \cdot)$.

Depending on the functional form of $\mathcal{F}$ and $\mathcal{G}$, computing the projection may be hard.



Model Translation

Given a classifier $f \in \mathcal{F}$ (e.g., a neural net), find a white-box classifier $g \in \mathcal{G}$ (e.g., a shallow decision tree) that approximates its predictions.

Strategy:

1. Sample a (large) set of instances $\{\mathrm{x}_1, \ldots, \mathrm{x}_m\}$, e.g., take random documents from the internet or replace words and sentences in your target document
2. Obtain a prediction $y_i := f(\mathrm{x}_i)$ for $i = 1, \ldots, m$ for each of your samples
3. Fit $g \in \mathcal{G}$ on the the **synthetic data set** $S = \{(\mathrm{x}_i, y_i) : i = 1, \ldots, m\}$

In other words, model translation can be implemented as **learning** using a synthetic dataset labeled using the model to be translated.

The trained white-box model g will have a decision surface *similar* to that of f, hence it can be used to answer "why" questions in its place.[3]

[3] This assumes that the explanation only includes relevance information about the observed, input variables. If the explanation also includes latent variables (e.g., whether concepts captured by hidden layers are present or not), then the white-box model must also match the output of the black-box for those variables.

Model Translation

Given a classifier $f \in \mathcal{F}$ (e.g., a neural net), find a white-box classifier $g \in \mathcal{G}$ (e.g., a shallow decision tree) that approximates its predictions.

Strategy:

1. Sample a (large) set of instances $\{\mathrm{x}_1, \ldots, \mathrm{x}_m\}$, e.g., take random documents from the internet or replace words and sentences in your target document
2. Obtain a prediction $y_i := f(\mathrm{x}_i)$ for $i = 1, \ldots, m$ for each of your samples
3. Fit $g \in \mathcal{G}$ on the the **synthetic data set** $S = \{(\mathrm{x}_i, y_i) : i = 1, \ldots, m\}$

In other words, model translation can be implemented as **learning** using a synthetic dataset labeled using the model to be translated.

The trained white-box model g will have a decision surface *similar* to that of f, hence it can be used to answer "why" questions in its place.[3]

[3] This assumes that the explanation only includes relevance information about the observed, input variables. If the explanation also includes latent variables (e.g., whether concepts captured by hidden layers are present or not), then the white-box model must also match the output of the black-box for those variables.

Model Translation

Given a classifier $f \in \mathcal{F}$ (e.g., a neural net), find a white-box classifier $g \in \mathcal{G}$ (e.g., a shallow decision tree) that approximates its predictions.

Strategy:

1. Sample a (large) set of instances $\{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$, e.g., take random documents from the internet or replace words and sentences in your target document
2. Obtain a prediction $y_i := f(\mathbf{x}_i)$ for $i = 1, \ldots, m$ for each of your samples
3. Fit $g \in \mathcal{G}$ on the the **synthetic data set** $S = \{(\mathbf{x}_i, y_i) : i = 1, \ldots, m\}$

In other words, model translation can be implemented as **learning** using a synthetic dataset labeled using the model to be translated.

The trained white-box model g will have a decision surface *similar* to that of f, hence it can be used to answer "why" questions in its place.[3]

[3] This assumes that the explanation only includes relevance information about the observed, input variables. If the explanation also includes latent variables (e.g., whether concepts captured by hidden layers are present or not), then the white-box model must also match the output of the black-box for those variables.

Model Translation

Given a classifier $f \in \mathcal{F}$ (e.g., a neural net), find a white-box classifier $g \in \mathcal{G}$ (e.g., a shallow decision tree) that approximates its predictions.

Strategy:

1. Sample a (large) set of instances $\{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$, e.g., take random documents from the internet or replace words and sentences in your target document
2. Obtain a prediction $y_i := f(\mathbf{x}_i)$ for $i = 1, \ldots, m$ for each of your samples
3. Fit $g \in \mathcal{G}$ on the the **synthetic data set** $S = \{(\mathbf{x}_i, y_i) : i = 1, \ldots, m\}$

In other words, model translation can be implemented as **learning** using a synthetic dataset labeled using the model to be translated.

The trained white-box model g will have a decision surface *similar* to that of f, hence it can be used to answer "why" questions in its place.[3]

[3] This assumes that the explanation only includes relevance information about the observed, input variables. If the explanation also includes latent variables (e.g., whether concepts captured by hidden layers are present or not), then the white-box model must also match the output of the black-box for those variables.

Model Translation

Given a classifier $f \in \mathcal{F}$ (e.g., a neural net), find a white-box classifier $g \in \mathcal{G}$ (e.g., a shallow decision tree) that approximates its predictions.

Strategy:

1. Sample a (large) set of instances $\{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$, e.g., take random documents from the internet or replace words and sentences in your target document
2. Obtain a prediction $y_i := f(\mathbf{x}_i)$ for $i = 1, \ldots, m$ for each of your samples
3. Fit $g \in \mathcal{G}$ on the the **synthetic data set** $S = \{(\mathbf{x}_i, y_i) : i = 1, \ldots, m\}$

In other words, model translation can be implemented as **learning** using a synthetic dataset labeled using the model to be translated.

The trained white-box model g will have a decision surface *similar* to that of f, hence it can be used to answer "why" questions in its place.[3]

[3] This assumes that the explanation only includes relevance information about the observed, input variables. If the explanation also includes latent variables (e.g., whether concepts captured by hidden layers are present or not), then the white-box model must also match the output of the black-box for those variables.

Model Translation

Given a classifier $f \in \mathcal{F}$ (e.g., a neural net), find a white-box classifier $g \in \mathcal{G}$ (e.g., a shallow decision tree) that approximates its predictions.

Strategy:

1. Sample a (large) set of instances $\{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$, e.g., take random documents from the internet or replace words and sentences in your target document
2. Obtain a prediction $y_i := f(\mathbf{x}_i)$ for $i = 1, \ldots, m$ for each of your samples
3. Fit $g \in \mathcal{G}$ on the the **synthetic data set** $S = \{(\mathbf{x}_i, y_i) : i = 1, \ldots, m\}$

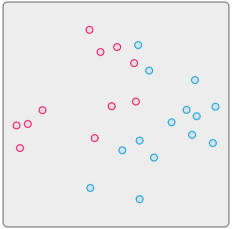
In other words, model translation can be implemented as **learning** using a synthetic dataset labeled using the model to be translated.

The trained white-box model g will have a decision surface *similar* to that of f, hence it can be used to answer "why" questions in its place.[3]

[3] This assumes that the explanation only includes relevance information about the observed, input variables. If the explanation also includes latent variables (e.g., whether concepts captured by hidden layers are present or not), then the white-box model must also match the output of the black-box for those variables.

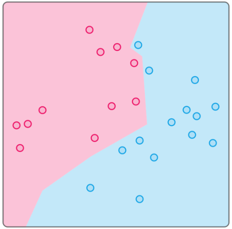
Model Translation Step by Step

1. You are given a handful of training points, e.g., labeled documents. Color indicates the label.



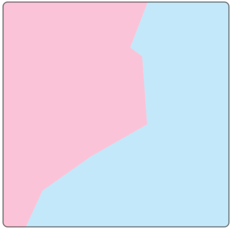
Model Translation Step by Step

1. You are given a handful of training points, e.g., labeled documents. Color indicates the label.
2. You train a black-box model on the training data.



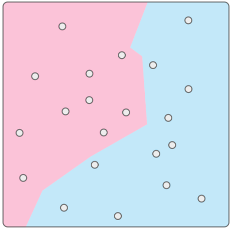
Model Translation Step by Step

1. You are given a handful of training points, e.g., labeled documents.
Color indicates the label.
2. You train a black-box model on the training data.
3. How to obtain a white-box model that mimics its predictions?



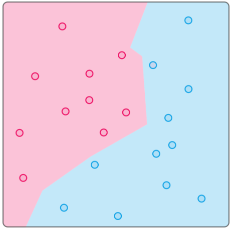
Model Translation Step by Step

1. You are given a handful of training points, e.g., labeled documents. Color indicates the label.
2. You train a black-box model on the training data.
3. How to obtain a white-box model that mimics its predictions?
4. Sample a large set of synthetic inputs - they have no label yet!



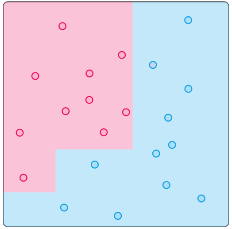
Model Translation Step by Step

1. You are given a handful of training points, e.g., labeled documents. Color indicates the label.
2. You train a black-box model on the training data.
3. How to obtain a white-box model that mimics its predictions?
4. Sample a large set of synthetic inputs - they have no label yet!
5. Use the black-box model's prediction to label these synthetic inputs.



Model Translation Step by Step

1. You are given a handful of training points, e.g., labeled documents. Color indicates the label.
2. You train a black-box model on the training data.
3. How to obtain a white-box model that mimics its predictions?
4. Sample a large set of synthetic inputs - they have no label yet!
5. Use the black-box model's prediction to label these synthetic inputs.
6. Use the synthetic examples to train a white-box model - *and voilà*!



- **How large should the synthetic data set S be?**
  - Start with a small S
  - Grow S and retrain until $d(f, g) \leq \tau$, with τ controllable threshold.

- **There may be multiple $g \in \mathcal{G}$ with the same distance to f / accuracy on S**
  - Example: two different decision trees g that both “look like” f.
  - Troublesome if they have different structure and give different explanations!
  - Sometimes it is enough to grow S so to remove alternatives.

- **How complex should g be allowed to be?**
  - If g is too simple, it may not capture f's decision surface faithfully enough
  - Making g too complex may break interpretability (and require enormous amounts of synthetic data)
  - **There may be no middle ground!**

How large should the synthetic data set S be?

- Start with a small S
- Grow S and retrain until $d(f, g) \leq \tau$, with τ controllable threshold.

There may be multiple $g \in \mathcal{G}$ with the same distance to f / accuracy on S

- Example: two different decision trees g that both "look like" f.
- Troublesome if they have different structure and give different explanations!
- Sometimes it is enough to grow S so to remove alternatives.

How complex should g be allowed to be?

- If g is too simple, it may not capture f's decision surface faithfully enough
- Making g too complex may break interpretability (and require enormous amounts of synthetic data)
- **There may be no middle ground!**

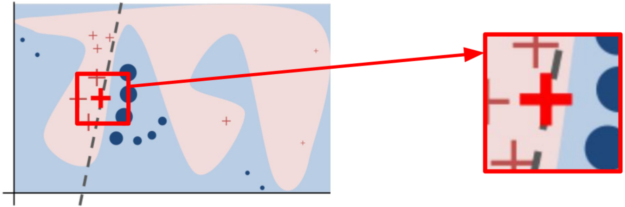
- **How large should the synthetic data set S be?**
  - Start with a small S
  - Grow S and retrain until $d(f, g) \leq \tau$, with τ controllable threshold.

- **There may be multiple $g \in \mathcal{G}$ with the same distance to f / accuracy on S**
  - Example: two different decision trees g that both "look like" f.
  - Troublesome if they have different structure and give different explanations!
  - Sometimes it is enough to grow S so to remove alternatives.

- **How complex should g be allowed to be?**
  - If g is too simple, it may not capture f's decision surface faithfully enough
  - Making g too complex may break interpretability (and require enormous amounts of synthetic data)
  - **There may be no middle ground!**

Local Interpretable Model-agnostic Explanations (LIME)

Idea: rather than translating *all* of f, only translate the neighborhood of $f(\mathrm{x}_0)$



- Those parts of the model that do not contribute to the decision surface around $f(\mathrm{x}_0)$ are **irrelevant** and do not influence the prediction nor the explanation.
- Even if the model is extremely complex, **locally it can be much simpler** (it is almost linear in this example) meaning that it will be much easier to fit it with an interpretable white-box model!

Credit [Ribeiro et al., 2016]

LIME

Given a classifier $f \in \mathcal{F}$ and a point x_0, find a white-box classifier $g_0 \in \mathcal{G}$ that approximates the predictions of f **in the neighborhood of x_0**.

Algorithm:

- Sample a set of instances $\{\mathrm{x}_1, \ldots, \mathrm{x}_m\}$ from an "appropriate" distribution [**same as before**]
- Label all samples using f, obtaining $y_i = f(\mathrm{x}_i)$ for all $i \in [m]$ [**same as before**]
- Fit $g_0 \in \mathcal{G}$ by solving the **weighted learning problem**:

$$g_0 := \operatorname*{argmin}_{g \in \mathcal{G}} \frac{1}{m} \sum_{i \in [m]} k(\mathrm{x}_0, \mathrm{x}_i) L(g_0(\mathrm{x}_i), y_i)$$

- Each example (x_i, y_i) is **weighted by its similarity to x** using a kernel k, e.g., a Gaussian kernel:

$$k(\mathrm{x}_0, \mathrm{x}_i) = \exp(-\gamma \cdot \|\mathrm{x}_0 - \mathrm{x}_i\|^2)$$

The *closer* to x_0, the more *important* getting the label of x_i right is.

Remark: notice that the kernel upscales (exponentially) all points closer than a threshold and downscales (exponentially) all points farther than the threshold.

LIME

Given a classifier $f \in \mathcal{F}$ and a point x_0, find a white-box classifier $g_0 \in \mathcal{G}$ that approximates the predictions of f **in the neighborhood of x_0**.

Algorithm:

- Sample a set of instances $\{\mathrm{x}_1, \ldots, \mathrm{x}_m\}$ from an "appropriate" distribution [**same as before**]
- Label all samples using f, obtaining $y_i = f(\mathrm{x}_i)$ for all $i \in [m]$ [**same as before**]
- Fit $g_0 \in \mathcal{G}$ by solving the **weighted learning problem**:

$$g_0 := \underset{g \in \mathcal{G}}{\operatorname{argmin}} \frac{1}{m} \sum_{i \in [m]} k(\mathrm{x}_0, \mathrm{x}_i) L(g_0(\mathrm{x}_i), y_i)$$

- Each example (x_i, y_i) is **weighted by its similarity to x** using a kernel k, e.g., a Gaussian kernel:

$$k(\mathrm{x}_0, \mathrm{x}_i) = \exp(-\gamma \cdot \|\mathrm{x}_0 - \mathrm{x}_i\|^2)$$

The *closer* to x_0, the more *important* getting the label of x_i right is.

Remark: notice that the kernel upscales (exponentially) all points closer than a threshold and downscales (exponentially) all points farther than the threshold.

LIME

Given a classifier $f \in \mathcal{F}$ and a point x_0, find a white-box classifier $g_0 \in \mathcal{G}$ that approximates the predictions of f **in the neighborhood of x_0**.

Algorithm:

- Sample a set of instances $\{\mathrm{x}_1, \ldots, \mathrm{x}_m\}$ from an "appropriate" distribution [**same as before**]
- Label all samples using f, obtaining $y_i = f(\mathrm{x}_i)$ for all $i \in [m]$ [**same as before**]
- Fit $g_0 \in \mathcal{G}$ by solving the **weighted learning problem**:

$$g_0 := \operatorname*{argmin}_{g \in \mathcal{G}} \frac{1}{m} \sum_{i \in [m]} k(\mathrm{x}_0, \mathrm{x}_i) L(g_0(\mathrm{x}_i), y_i)$$

- Each example (x_i, y_i) is **weighted by its similarity to x** using a kernel k, e.g., a Gaussian kernel:

$$k(\mathrm{x}_0, \mathrm{x}_i) = \exp(-\gamma \cdot \|\mathrm{x}_0 - \mathrm{x}_i\|^2)$$

The *closer* to x_0, the more *important* getting the label of x_i right is.

Remark: notice that the kernel upscales (exponentially) all points closer than a threshold and downscales (exponentially) all points farther than the threshold.

LIME

Given a classifier $f \in \mathcal{F}$ and a point x_0, find a white-box classifier $g_0 \in \mathcal{G}$ that approximates the predictions of f **in the neighborhood of x_0**.

Algorithm:

- Sample a set of instances $\{\mathrm{x}_1, \ldots, \mathrm{x}_m\}$ from an "appropriate" distribution [**same as before**]
- Label all samples using f, obtaining $y_i = f(\mathrm{x}_i)$ for all $i \in [m]$ [**same as before**]
- Fit $g_0 \in \mathcal{G}$ by solving the **weighted learning problem**:

$$g_0 := \underset{g \in \mathcal{G}}{\operatorname{argmin}} \frac{1}{m} \sum_{i \in [m]} k(\mathrm{x}_0, \mathrm{x}_i) L(g_0(\mathrm{x}_i), y_i)$$

- Each example (x_i, y_i) is **weighted by its similarity to x** using a kernel k, e.g., a Gaussian kernel:

$$k(\mathrm{x}_0, \mathrm{x}_i) = \exp(-\gamma \cdot \|\mathrm{x}_0 - \mathrm{x}_i\|^2)$$

The *closer* to x_0, the more *important* getting the label of x_i right is.

Remark: notice that the kernel upscales (exponentially) all points closer than a threshold and downscales (exponentially) all points farther than the threshold.

LIME

Given a classifier $f \in \mathcal{F}$ and a point x_0, find a white-box classifier $g_0 \in \mathcal{G}$ that approximates the predictions of f **in the neighborhood of x_0**.

Algorithm:

- Sample a set of instances $\{\mathrm{x}_1, \ldots, \mathrm{x}_m\}$ from an "appropriate" distribution [**same as before**]
- Label all samples using f, obtaining $y_i = f(\mathrm{x}_i)$ for all $i \in [m]$ [**same as before**]
- Fit $g_0 \in \mathcal{G}$ by solving the **weighted learning problem**:

$$g_0 := \operatorname*{argmin}_{g \in \mathcal{G}} \frac{1}{m} \sum_{i \in [m]} k(\mathrm{x}_0, \mathrm{x}_i) L(g_0(\mathrm{x}_i), y_i)$$

- Each example (x_i, y_i) is **weighted by its similarity to x** using a kernel k, e.g., a Gaussian kernel:

$$k(\mathrm{x}_0, \mathrm{x}_i) = \exp(-\gamma \cdot \|\mathrm{x}_0 - \mathrm{x}_i\|^2)$$

The *closer* to x_0, the more *important* getting the label of x_i right is.

Remark: notice that the kernel upscales (exponentially) all points closer than a threshold and downscales (exponentially) all points farther than the threshold.

LIME

Given a classifier $f \in \mathcal{F}$ and a point $\mathbf{x}_0$, find a white-box classifier $g_0 \in \mathcal{G}$ that approximates the predictions of f **in the neighborhood of** $\mathbf{x}_0$.

Algorithm:

- Sample a set of instances $\{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$ from an "appropriate" distribution [**same as before**]
- Label all samples using f, obtaining $y_i = f(\mathbf{x}_i)$ for all $i \in [m]$ [**same as before**]
- Fit $g_0 \in \mathcal{G}$ by solving the **weighted learning problem**:

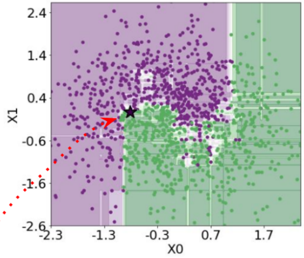
$$g_0 := \operatorname*{argmin}_{g \in \mathcal{G}} \frac{1}{m} \sum_{i \in [m]} k(\mathbf{x}_0, \mathbf{x}_i) L(g_0(\mathbf{x}_i), y_i)$$

- Each example $(\mathbf{x}_i, y_i)$ is **weighted by its similarity to** $\mathbf{x}$ using a kernel k, e.g., a Gaussian kernel:

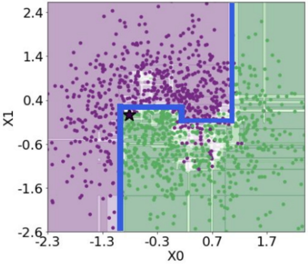
$$k(\mathbf{x}_0, \mathbf{x}_i) = \exp(-\gamma \cdot \|\mathbf{x}_0 - \mathbf{x}_i\|^2)$$

  The *closer* to $\mathbf{x}_0$, the more *important* getting the label of $\mathbf{x}_i$ right is.

  Remark: notice that the kernel upscales (exponentially) all points closer than a threshold and downscales (exponentially) all points farther than the threshold.



2D projection of the decision surface of a random forest classifier + **random instances** sampled around the **prediction to be explained** (illustrated by a black star).



Simpler but explainable **decision tree** learned from the synthetic dataset.

Credit: [Guidotti et al., 2019]

■ Sample $\{\mathbf{x}_i\}$ from some distribution $P(\mathbf{X})$. What distribution?

Idea: Use the ground-truth distribution $P^*(\mathbf{X})$. This way, g_0 ends up "looking like" f in regions that actually occur in the data.

- In practice, P^* is **unknown**, so it must be either:
- **Replaced with the empirical distribution**, i.e., the training set used to train f, which is however likely too small to truly capture the neighborhood of any given instance $\mathbf{x}_0$.
- **Replaced with a generative model** $\hat{P}(\mathbf{X})$ estimated on the training data used for f. Sampling from $\hat{P}(\mathbf{X})$ may be computationally challenging & estimation of generative models is non-trivial.

Sampling from $P^*(\mathbf{X})$ neglects the behavior of f in regions that do not normally occur: this can **hide C-H behavior**.

If the goal is to **understand** why the decision $f(\mathbf{x}_0) = y_0$ was made, so to build or reject trust in f, there is no reason to restrict the synthetic samples $\{\mathbf{x}_i\}$ to high-density regions: *the whole neighborhood of* $\mathbf{x}_0$ *should be covered!*

Sample $\{\mathbf{x}_i\}$ from some distribution $P(\mathbf{X})$. What distribution?

Idea: Use the ground-truth distribution $P^*(\mathbf{X})$. This way, g_0 ends up "looking like" f in regions that actually occur in the data.

- In practice, P^* is **unknown**, so it must be either:
- **Replaced with the empirical distribution**, i.e., the training set used to train f, which is however likely too small to truly capture the neighborhood of any given instance $\mathbf{x}_0$.
- **Replaced with a generative model** $\hat{P}(\mathbf{X})$ estimated on the training data used for f.
  Sampling from $\hat{P}(\mathbf{X})$ may be computationally challenging & estimation of generative models is non-trivial.

Sampling from $P^*(\mathbf{X})$ neglects the behavior of f in regions that do not normally occur: this can **hide C-H behavior**.

If the goal is to **understand** why the decision $f(\mathbf{x}_0) = y_0$ was made, so to build or reject trust in f, there is no reason to restrict the synthetic samples $\{\mathbf{x}_i\}$ to high-density regions: *the whole neighborhood of* $\mathbf{x}_0$ *should be covered!*

- Sample $\{\mathrm{x}_i\}$ from some distribution $P(\mathbf{X})$. What distribution?

Idea: Use the ground-truth distribution $P^*(\mathbf{X})$. This way, g_0 ends up "looking like" f in regions that actually occur in the data.

- In practice, P^* is **unknown**, so it must be either:
- **Replaced with the empirical distribution**, i.e., the training set used to train f, which is however likely too small to truly capture the neighborhood of any given instance x_0.
- **Replaced with a generative model** $\hat{P}(\mathrm{X})$ estimated on the training data used for f. Sampling from $\hat{P}(\mathrm{X})$ may be computationally challenging & estimation of generative models is non-trivial.

Sampling from $P^*(\mathrm{X})$ neglects the behavior of f in regions that do not normally occur: this can **hide C-H behavior**.

If the goal is to **understand** why the decision $f(\mathrm{x}_0) = y_0$ was made, so to build or reject trust in f, there is no reason to restrict the synthetic samples $\{\mathrm{x}_i\}$ to high-density regions: *the whole neighborhood of* x_0 *should be covered!*

- Sample $\{\mathbf{x}_i\}$ from some distribution $P(\mathbf{X})$. What distribution?

Idea: Use the ground-truth distribution $P^*(\mathbf{X})$. This way, g_0 ends up "looking like" f in regions that actually occur in the data.

- In practice, P^* is **unknown**, so it must be either:
- **Replaced with the empirical distribution**, i.e., the training set used to train f, which is however likely too small to truly capture the neighborhood of any given instance $\mathbf{x}_0$.
- **Replaced with a generative model** $\hat{P}(\mathbf{X})$ estimated on the training data used for f.
  Sampling from $\hat{P}(\mathbf{X})$ may be computationally challenging & estimation of generative models is non-trivial.

Sampling from $P^*(\mathbf{X})$ neglects the behavior of f in regions that do not normally occur: this can **hide C-H behavior**.

If the goal is to **understand** why the decision $f(\mathbf{x}_0) = y_0$ was made, so to build or reject trust in f, there is no reason to restrict the synthetic samples $\{\mathbf{x}_i\}$ to high-density regions: *the whole neighborhood of* $\mathbf{x}_0$ *should be covered!*

■ Sample $\{\mathbf{x}_i\}$ from some distribution $P(\mathbf{X})$. What distribution?

Idea: Use the ground-truth distribution $P^*(\mathbf{X})$. This way, g_0 ends up "looking like" f in regions that actually occur in the data.

- In practice, P^* is **unknown**, so it must be either:
- **Replaced with the empirical distribution**, i.e., the training set used to train f, which is however likely too small to truly capture the neighborhood of any given instance $\mathbf{x}_0$.
- **Replaced with a generative model** $\hat{P}(\mathbf{X})$ estimated on the training data used for f.
  Sampling from $\hat{P}(\mathbf{X})$ may be computationally challenging & estimation of generative models is non-trivial.

Sampling from $P^*(\mathbf{X})$ neglects the behavior of f in regions that do not normally occur: this can **hide C-H behavior**.

If the goal is to **understand** why the decision $f(\mathbf{x}_0) = y_0$ was made, so to build or reject trust in f, there is no reason to restrict the synthetic samples $\{\mathbf{x}_i\}$ to high-density regions: *the whole neighborhood of* $\mathbf{x}_0$ *should be covered!*

■ Sample $\{\mathbf{x}_i\}$ from some distribution $P(\mathbf{X})$. What distribution?

Idea: Use the ground-truth distribution $P^*(\mathbf{X})$. This way, g_0 ends up "looking like" f in regions that actually occur in the data.

- In practice, P^* is **unknown**, so it must be either:
- **Replaced with the empirical distribution**, i.e., the training set used to train f, which is however likely too small to truly capture the neighborhood of any given instance $\mathbf{x}_0$.
- **Replaced with a generative model** $\hat{P}(\mathbf{X})$ estimated on the training data used for f.
  Sampling from $\hat{P}(\mathbf{X})$ may be computationally challenging & estimation of generative models is non-trivial.

Sampling from $P^*(\mathbf{X})$ neglects the behavior of f in regions that do not normally occur: this can **hide C-H behavior**.

If the goal is to **understand** why the decision $f(\mathbf{x}_0) = y_0$ was made, so to build or reject trust in f, there is no reason to restrict the synthetic samples $\{\mathbf{x}_i\}$ to high-density regions: *the whole neighborhood of* $\mathbf{x}_0$ *should be covered!*

■ Sample $\{\mathbf{x}_i\}$ from some distribution $P(\mathbf{X})$. What distribution?

Idea: Use the ground-truth distribution $P^*(\mathbf{X})$. This way, g_0 ends up "looking like" f in regions that actually occur in the data.

- In practice, P^* is **unknown**, so it must be either:
- **Replaced with the empirical distribution**, i.e., the training set used to train f, which is however likely too small to truly capture the neighborhood of any given instance $\mathbf{x}_0$.
- **Replaced with a generative model** $\hat{P}(\mathbf{X})$ estimated on the training data used for f. Sampling from $\hat{P}(\mathbf{X})$ may be computationally challenging & estimation of generative models is non-trivial.

Sampling from $P^*(\mathbf{X})$ neglects the behavior of f in regions that do not normally occur: this can **hide C-H behavior**.

If the goal is to **understand** why the decision $f(\mathbf{x}_0) = y_0$ was made, so to build or reject trust in f, there is no reason to restrict the synthetic samples $\{\mathbf{x}_i\}$ to high-density regions: *the whole neighborhood of* $\mathbf{x}_0$ *should be covered!*

It depends on the type of variables:

- If x_i is a **categorical** variable and all its values are known, then simply pick from a value uniformly at random.
  Example: $x_i \in \{winter, autumn, summer, spring\}$, pick any choice at random.
- If x_i is a **continuous** variable, sample from either a uniform distribution or a Gaussian.
  The width of the distribution can be chosen by looking at the data.
  Example: use empirical std. deviation to define the Gaussian.

Issue: the samples look distinctly "different" from regular points sampled from $P^*(\mathbf{X})$. This makes it easy to build attacks on the explanations computed by LIME, see [Slack et al., 2020].

It depends on the type of variables:

- If x_i is a **categorical** variable and all its values are known, then simply pick from a value uniformly at random.
  Example: $x_i \in \{winter, autumn, summer, spring\}$, pick any choice at random.
- If x_i is a **continuous** variable, sample from either a uniform distribution or a Gaussian.
  The width of the distribution can be chosen by looking at the data.
  Example: use empirical std. deviation to define the Gaussian.

Issue: the samples look distinctly "different" from regular points sampled from $P^*(\mathbf{X})$. This makes it easy to build attacks on the explanations computed by LIME, see [Slack et al., 2020].

LIME requires to solve:

$$\operatorname*{argmin}_{g \in \mathcal{G}} \frac{1}{m} \sum_{i \in [m]} k(\mathbf{x}, \mathbf{x}_i) \underbrace{L(g_0(\mathbf{x}_i), y_i)}_{\text{loss on } (\mathbf{x}_i, y_i)}$$

One would expect L to be a loss for **classification**, right?

However, if the surrogate g is a linear model, then LIME uses an **L_2 loss**:

$$L(\hat{y}, y) = (y - \hat{y})^2$$

This immediately gives:

$$\operatorname*{argmin}_{g \in \mathcal{G}} \frac{1}{m} \sum_{i \in [m]} k(\mathbf{x}, \mathbf{x}_i)(g_0(\mathbf{x}_i) - f(\mathbf{x}_i))^2$$

This problem admits a **closed-form** solution and it can be computed in a **numerically stable** manner using **least squares**.

LIME has one more trick: learning a **k-sparse** weight vector $\mathbf{w}$ using a modification of least squares called LASSO [Tibshirani, 1996].

However, if the surrogate g is a linear model, then LIME uses an L_2 **loss**:

$$L(\hat{y}, y) = (y - \hat{y})^2$$

This immediately gives:

$$\underset{g \in \mathcal{G}}{\operatorname{argmin}} \ \frac{1}{m} \sum_{i \in [m]} k(\mathbf{x}, \mathbf{x}_i)(g_0(\mathbf{x}_i) - f(\mathbf{x}_i))^2$$

This problem admits a **closed-form** solution and it can be computed in a **numerically stable** manner using **least squares**.

LIME has one more trick: learning a ***k*-sparse** weight vector $\mathbf{w}$ using a modification of least squares called LASSO [Tibshirani, 1996].

Let $g_0(\mathbf{x})$ be a **linear model**:

$$g_0(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b = \sum_{j\in[d]} w_j x_j + b$$

Remark: the offset b can be ignored if we center the data.

Replacing g_0 with the above in the LIME objective, we obtain:

$$\frac{1}{m}\sum_{i\in[m]} k(\mathbf{x},\mathbf{x}_i)(g_0(\mathbf{x}_i)-y_i)^2 = \sum_{i\in[m]} \alpha_i^2(\mathbf{w}^\top\mathbf{x}_i - y_i)^2 \qquad \alpha_i := \sqrt{\frac{k(\mathbf{x},\mathbf{x}_i)}{m}}$$

$$= \|\mathbf{a}\odot(\mathbf{w}^\top X - \mathbf{y})\|^2 = \|\mathbf{w}^\top X' - \mathbf{y}'\|^2 \qquad X', \mathbf{y}' \text{ absorbed } \mathbf{a}$$

where $\odot$ is the Hadamard (element-wise) product and we used:

$$\mathbf{a} := (\alpha_1, \ldots, \alpha_m), \qquad X := [\mathbf{x}_1, \ldots, \mathbf{x}_m], \qquad \mathbf{y} = (y_1, \ldots, y_m)$$

Hence fitting a linear g_0 in LIME boils down to solving **least squares**:

$$\operatorname*{argmin}_{\mathbf{w}\in\mathbb{R}^d} \|\mathbf{w}^\top X' - \mathbf{y}'\| \quad \text{s.t.} \quad \|\mathbf{w}\| \leq 1$$

Let $g_0(\mathbf{x})$ be a **linear model**:

$$g_0(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b = \sum_{j\in[d]} w_j x_j + b$$

Remark: the offset b can be ignored if we center the data.

Replacing g_0 with the above in the LIME objective, we obtain:

$$\frac{1}{m}\sum_{i\in[m]} k(\mathbf{x},\mathbf{x}_i)(g_0(\mathbf{x}_i) - y_i)^2 = \sum_{i\in[m]} \alpha_i^2(\mathbf{w}^\top \mathbf{x}_i - y_i)^2 \qquad \alpha_i := \sqrt{\frac{k(\mathbf{x},\mathbf{x}_i)}{m}}$$

$$= \|\mathbf{a}\odot(\mathbf{w}^\top X - \mathbf{y})\|^2 = \|\mathbf{w}^\top X' - \mathbf{y}'\|^2 \qquad \text{X', } \mathbf{y}\text{' absorbed } \mathbf{a}$$

where $\odot$ is the Hadamard (element-wise) product and we used:

$$\mathbf{a} := (\alpha_1,\dots,\alpha_m), \qquad X := [\mathbf{x}_1,\dots,\mathbf{x}_m], \qquad \mathbf{y} = (y_1,\dots,y_m)$$

Hence fitting a linear g_0 in LIME boils down to solving **least squares**:

$$\operatorname*{argmin}_{\mathbf{w}\in\mathbb{R}^d} \|\mathbf{w}^\top X' - \mathbf{y}'\| \quad \text{s.t.} \quad \|\mathbf{w}\| \le 1$$

Let $g_0(\mathbf{x})$ be a **linear model**:

$$g_0(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b = \sum_{j\in[d]} w_j x_j + b$$

Remark: the offset b can be ignored if we center the data.

Replacing g_0 with the above in the LIME objective, we obtain:

$$\frac{1}{m}\sum_{i\in[m]} k(\mathbf{x},\mathbf{x}_i)(g_0(\mathbf{x}_i)-y_i)^2 = \sum_{i\in[m]} \alpha_i^2(\mathbf{w}^\top\mathbf{x}_i - y_i)^2 \qquad \alpha_i := \sqrt{\frac{k(\mathbf{x},\mathbf{x}_i)}{m}}$$

$$= \|\mathbf{a}\odot(\mathbf{w}^\top X - \mathbf{y})\|^2 = \|\mathbf{w}^\top X' - \mathbf{y}'\|^2 \qquad \text{X', } \mathbf{y}\text{' absorbed } \mathbf{a}$$

where $\odot$ is the Hadamard (element-wise) product and we used:

$$\mathbf{a} := (\alpha_1,\dots,\alpha_m), \qquad X := [\mathbf{x}_1,\dots,\mathbf{x}_m], \qquad \mathbf{y} = (y_1,\dots,y_m)$$

Hence fitting a linear g_0 in LIME boils down to solving **least squares**:

$$\operatorname*{argmin}_{\mathbf{w}\in\mathbb{R}^d} \|\mathbf{w}^\top X' - \mathbf{y}'\| \quad \text{s.t.} \quad \|\mathbf{w}\| \le 1$$

LIME has one more trick: learning a ***k*-sparse** weight vector $\mathbf{w}$.

This can be achieved by solving:

$$\underset{\mathbf{w}\in\mathbb{R}^d}{\operatorname{argmin}} \|\mathbf{w}^\top X' - \mathbf{y}'\| \quad \text{s.t.} \quad \|\mathbf{w}\|_0 \leq b$$

where $\|\mathbf{w}\|_0 = \sum_{j\in[d]} \mathbb{1}\left(w_j \neq 0\right)$ is the L_0 pseudo-norm.

- Solving this is a **hard (combinatorial) optimization problem**.

- Use LASSO instead [Tibshirani, 1996], which involves solving:

$$\underset{\mathbf{w}\in\mathbb{R}^d}{\operatorname{argmin}} \|\mathbf{w}^\top X' - \mathbf{y}'\| + \lambda \cdot \|\mathbf{w}\|_1, \qquad \|\mathbf{w}\|_1 = \sum_{j\in[d]} |w_j|$$

It turns out that solving this (non-combinatorial) surrogate provably solves the original problem (under assumptions).

LIME has one more trick: learning a ***k*-sparse** weight vector $\mathbf{w}$.

This can be achieved by solving:

$$\operatorname*{argmin}_{\mathbf{w} \in \mathbb{R}^d} \|\mathbf{w}^\top X' - \mathbf{y}'\| \quad \text{s.t.} \quad \|\mathbf{w}\|_0 \leq b$$

where $\|\mathbf{w}\|_0 = \sum_{j \in [d]} \mathbb{1}\left(w_j \neq 0\right)$ is the L_0 pseudo-norm.

- Solving this is a **hard (combinatorial) optimization problem**.
- Use LASSO instead [Tibshirani, 1996], which involves solving:

$$\operatorname*{argmin}_{\mathbf{w} \in \mathbb{R}^d} \|\mathbf{w}^\top X' - \mathbf{y}'\| + \lambda \cdot \|\mathbf{w}\|_1, \qquad \|\mathbf{w}\|_1 = \sum_{j \in [d]} |w_j|$$

It turns out that solving this (non-combinatorial) surrogate provably solves the original problem (under assumptions).

LIME has one more trick: learning a *k*-**sparse** weight vector $\mathbf{w}$.

This can be achieved by solving:

$$\operatorname*{argmin}_{\mathbf{w}\in\mathbb{R}^d} \|\mathbf{w}^\top X' - \mathbf{y}'\| \quad \text{s.t.} \quad \|\mathbf{w}\|_0 \leq b$$

where $\|\mathbf{w}\|_0 = \sum_{j\in[d]} \mathbb{1}\left(w_j \neq 0\right)$ is the L_0 pseudo-norm.

- Solving this is a **hard (combinatorial) optimization problem**.

- Use LASSO instead [Tibshirani, 1996], which involves solving:

$$\operatorname*{argmin}_{\mathbf{w}\in\mathbb{R}^d} \|\mathbf{w}^\top X' - \mathbf{y}'\| + \lambda \cdot \|\mathbf{w}\|_1, \qquad \|\mathbf{w}\|_1 = \sum_{j\in[d]} |w_j|$$

It turns out that solving this (non-combinatorial) surrogate provably solves the original problem (under assumptions).

LIME has one more trick: learning a ***k*-sparse** weight vector $\mathbf{w}$.

This can be achieved by solving:

$$\underset{\mathbf{w}\in\mathbb{R}^d}{\text{argmin}} \ \|\mathbf{w}^\top X' - \mathbf{y}'\| \quad \text{s.t.} \quad \|\mathbf{w}\|_0 \leq b$$

where $\|\mathbf{w}\|_0 = \sum_{j\in[d]} \mathbb{1}\left(w_j \neq 0\right)$ is the L_0 pseudo-norm.

- Solving this is a **hard (combinatorial) optimization problem**.

- Use LASSO instead [Tibshirani, 1996], which involves solving:

$$\underset{\mathbf{w}\in\mathbb{R}^d}{\text{argmin}} \ \|\mathbf{w}^\top X' - \mathbf{y}'\| + \lambda \cdot \|\mathbf{w}\|_1, \qquad \|\mathbf{w}\|_1 = \sum_{j\in[d]} |w_j|$$

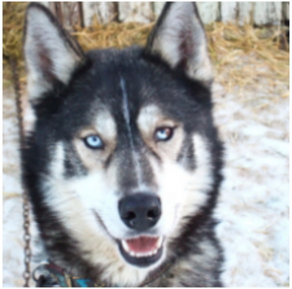
It turns out that solving this (non-combinatorial) surrogate provably solves the original problem (under assumptions).

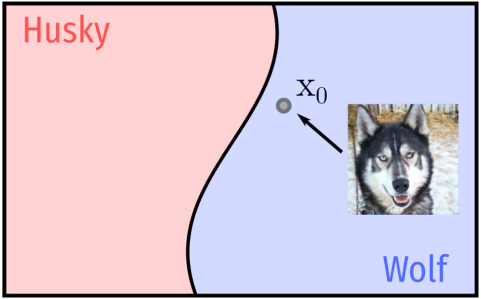
Illustration

Consider the task of discriminating between (images of) **wolves** and **husky dogs**.

Illustration

You receive this image x_0, which the black-box classifier f predicts as **wolf**

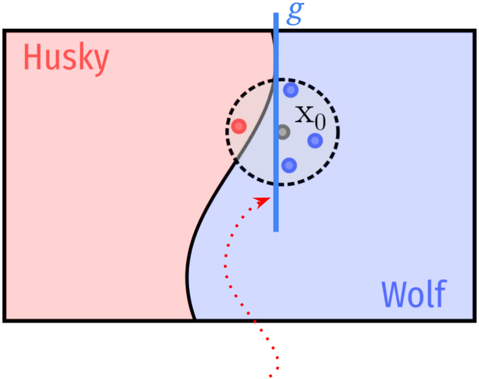




How does LIME construct an explanation for this decision?

Illustration

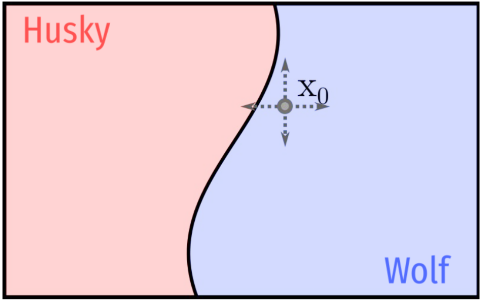
You receive this image $\mathbf{x}_0$, which the black-box classifier f predicts as **wolf**



LIME samples points in the neighborhood of $\mathbf{x}_0$ and fits a **sparse linear classifier** g_0 on them

Illustration

You receive this image x_0, which the black-box classifier f predicts as **wolf**



Roughly equivalent to **randomly perturbing** (*aka* “wiggling”) x_0, checking where the output of f changes, and then fitting a white-box model that mimics those changes.

- What about the input variables x_i are **not** interpretable?

- Black-box models often rely on complex features of the inputs $\mathbf{x} = (x_1, \ldots, x_n)$:
  - **Text**: tagging documents by looking for **sequences** of words
  - **Images**: classifying pictures by leveraging **high-order correlations** between pixels

Explanations extracted from white-box modes based on these features are **not interpretable**!

- LIME assumes to be given a function $\psi : \mathbb{R}^d \to \{0,1\}^q$ that maps inputs $\mathbf{x}$ to an **interpretable representation** $\psi(\mathbf{x})$:
  - **Text**: $\psi(\mathbf{x})$ represents document $\mathbf{x}$ in terms of presence/absence of **individual words**
  - **Images**: ψ represents image $\mathbf{x}$ in terms of presence/absence of **objects**

- What about the input variables x_i are **not** interpretable?

- Black-box models often rely on complex features of the inputs $\mathbf{x} = (x_1, \ldots, x_n)$:
  - **Text**: tagging documents by looking for **sequences** of words
  - **Images**: classifying pictures by leveraging **high-order correlations** between pixels

Explanations extracted from white-box modes based on these features are **not interpretable**!

- LIME assumes to be given a function $\psi : \mathbb{R}^d \to \{0, 1\}^q$ that maps inputs $\mathbf{x}$ to an **interpretable representation** $\psi(\mathbf{x})$:
  - **Text**: $\psi(\mathbf{x})$ represents document $\mathbf{x}$ in terms of presence/absence of **individual words**
  - **Images**: ψ represents image $\mathbf{x}$ in terms of presence/absence of **objects**

- What about the input variables x_i are **not** interpretable?

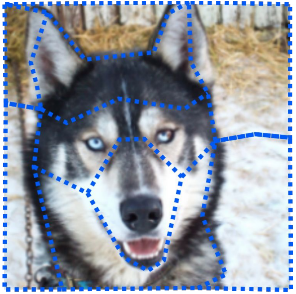
- Black-box models often rely on complex features of the inputs $\mathbf{x} = (x_1, \ldots, x_n)$:
  - **Text**: tagging documents by looking for **sequences** of words
  - **Images**: classifying pictures by leveraging **high-order correlations** between pixels

Explanations extracted from white-box modes based on these features are **not interpretable**!

- LIME assumes to be given a function $\psi : \mathbb{R}^d \to \{0, 1\}^q$ that maps inputs $\mathbf{x}$ to an **interpretable representation** $\psi(\mathbf{x})$:
  - **Text**: $\psi(\mathbf{x})$ represents document $\mathbf{x}$ in terms of presence/absence of **individual words**
  - **Images**: ψ represents image $\mathbf{x}$ in terms of presence/absence of **objects**

Illustration

You receive this image $\mathbf{x}_0$, which the black-box classifier f predicts as **wolf**



For images, LIME builds an **instance-specific map** $\psi_0(\mathbf{x})$ by **segmenting** the target image $\mathbf{x}_0$. In this case, the "wiggling" corresponds to filling individual segments with noise.

LIME (Updated)

Given a classifier $f \in \mathcal{F}$ and a point x_0, find a white-box classifier $g_0 \in \mathcal{G}$ that approximates the predictions of f **in the neighborhood of x_0**.

Algorithm:

- Sample a set of instances $\{\mathrm{x}_1, \ldots, \mathrm{x}_m\}$ from an "appropriate" distribution [**same as before**]
- Label all samples using f, obtaining $y_i = f(\mathrm{x}_i)$ for all $i \in [m]$ [**same as before**]
- Fit $g_0 \in \mathcal{G}$ by solving the weighted learning problem:

$$g_0 := \operatorname*{argmin}_{g \in \mathcal{G}} \frac{1}{m} \sum_{i \in [m]} k(\mathrm{x}_0, \mathrm{x}_i) L(g_0(\psi(\mathrm{x}_i)), y_i)$$

  The white-box model g_0 is now learned on the **interpretable feature space $\psi(\mathrm{x})$** → *its explanations will also be given in terms of the interpretable concepts*

- **Important**: ψ does not have to stay the same for different targets x_0 – so long as the features that it extracts are interpretable, we are good.

LIME (Updated)

Given a classifier $f \in \mathcal{F}$ and a point x_0, find a white-box classifier $g_0 \in \mathcal{G}$ that approximates the predictions of f **in the neighborhood of x_0**.

Algorithm:

- Sample a set of instances $\{\mathrm{x}_1, \ldots, \mathrm{x}_m\}$ from an "appropriate" distribution [**same as before**]
- Label all samples using f, obtaining $y_i = f(\mathrm{x}_i)$ for all $i \in [m]$ [**same as before**]
- Fit $g_0 \in \mathcal{G}$ by solving the weighted learning problem:

$$g_0 := \operatorname*{argmin}_{g \in \mathcal{G}} \frac{1}{m} \sum_{i \in [m]} k(\mathrm{x}_0, \mathrm{x}_i) L(g_0(\psi(\mathrm{x}_i)), y_i)$$

  The white-box model g_0 is now learned on the **interpretable feature space $\psi(\mathrm{x})$** → *its explanations will also be given in terms of the interpretable concepts*

- **Important**: ψ does not have to stay the same for different targets x_0 – so long as the features that it extracts are interpretable, we are good.

LIME (Updated)

Given a classifier $f \in \mathcal{F}$ and a point x_0, find a white-box classifier $g_0 \in \mathcal{G}$ that approximates the predictions of f **in the neighborhood of x_0**.

Algorithm:

- Sample a set of instances $\{\mathrm{x}_1, \ldots, \mathrm{x}_m\}$ from an "appropriate" distribution [**same as before**]
- Label all samples using f, obtaining $y_i = f(\mathrm{x}_i)$ for all $i \in [m]$ [**same as before**]
- Fit $g_0 \in \mathcal{G}$ by solving the weighted learning problem:

$$g_0 := \operatorname*{argmin}_{g \in \mathcal{G}} \frac{1}{m} \sum_{i \in [m]} k(\mathrm{x}_0, \mathrm{x}_i) L(g_0(\psi(\mathrm{x}_i)), y_i)$$

  The white-box model g_0 is now learned on the **interpretable feature space $\psi(\mathrm{x})$** → *its explanations will also be given in terms of the interpretable concepts*

- **Important**: ψ does not have to stay the same for different targets x_0 – so long as the features that it extracts are interpretable, we are good.

LIME (Updated)

Given a classifier $f \in \mathcal{F}$ and a point $\mathbf{x}_0$, find a white-box classifier $g_0 \in \mathcal{G}$ that approximates the predictions of f **in the neighborhood of $\mathbf{x}_0$**.

Algorithm:

- Sample a set of instances $\{\mathbf{x}_1, \dots, \mathbf{x}_m\}$ from an "appropriate" distribution [**same as before**]
- Label all samples using f, obtaining $y_i = f(\mathbf{x}_i)$ for all $i \in [m]$ [**same as before**]
- Fit $g_0 \in \mathcal{G}$ by solving the weighted learning problem:

$$g_0 := \operatorname*{argmin}_{g \in \mathcal{G}} \frac{1}{m} \sum_{i \in [m]} k(\mathbf{x}_0, \mathbf{x}_i) L(g_0(\psi(\mathbf{x}_i)), y_i)$$

The white-box model g_0 is now learned on the **interpretable feature space $\psi(\mathbf{x})$** → *its explanations will also be given in terms of the interpretable concepts*

- **Important**: ψ does not have to stay the same for different targets $\mathbf{x}_0$ – so long as the features that it extracts are interpretable, we are good.

LIME (Updated)

Given a classifier $f \in \mathcal{F}$ and a point $\mathbf{x}_0$, find a white-box classifier $g_0 \in \mathcal{G}$ that approximates the predictions of f **in the neighborhood of $\mathbf{x}_0$**.

Algorithm:

- Sample a set of instances $\{\mathbf{x}_1, \dots, \mathbf{x}_m\}$ from an "appropriate" distribution [**same as before**]
- Label all samples using f, obtaining $y_i = f(\mathbf{x}_i)$ for all $i \in [m]$ [**same as before**]
- Fit $g_0 \in \mathcal{G}$ by solving the weighted learning problem:

$$g_0 := \operatorname*{argmin}_{g \in \mathcal{G}} \frac{1}{m} \sum_{i \in [m]} k(\mathbf{x}_0, \mathbf{x}_i) L(g_0(\psi(\mathbf{x}_i)), y_i)$$

The white-box model g_0 is now learned on the **interpretable feature space $\psi(\mathbf{x})$** → *its explanations will also be given in terms of the interpretable concepts*

- **Important**: ψ does not have to stay the same for different targets $\mathbf{x}_0$ – so long as the features that it extracts are interpretable, we are good.

LIME (Updated)

Given a classifier $f \in \mathcal{F}$ and a point $\mathbf{x}_0$, find a white-box classifier $g_0 \in \mathcal{G}$ that approximates the predictions of f **in the neighborhood of $\mathbf{x}_0$**.

Algorithm:

- Sample a set of instances $\{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$ from an "appropriate" distribution [**same as before**]
- Label all samples using f, obtaining $y_i = f(\mathbf{x}_i)$ for all $i \in [m]$ [**same as before**]
- Fit $g_0 \in \mathcal{G}$ by solving the weighted learning problem:

$$g_0 := \operatorname*{argmin}_{g \in \mathcal{G}} \frac{1}{m} \sum_{i \in [m]} k(\mathbf{x}_0, \mathbf{x}_i) L(g_0(\psi(\mathbf{x}_i)), y_i)$$

  The white-box model g_0 is now learned on the **interpretable feature space $\psi(\mathbf{x})$** → *its explanations will also be given in terms of the interpretable concepts*

- **Important**: ψ does not have to stay the same for different targets $\mathbf{x}_0$ – so long as the features that it extracts are interpretable, we are good.

- Once g_0 is obtained, LIME extracts an explanation for $\hat{y}_0 = g_0(\mathrm{x}_0)$ – this is easy, because g_0 is a **white-box** model – and uses it as an explanation for $y_0 = f(\mathrm{x}_0)$.

- If g_0 is a sparse linear model:

$$g_0(\mathrm{x}) = \sum_{j \in [d]} w_j \psi_j(\mathrm{x}) + b$$

  - $w_i > 0 \implies \psi_i(\mathrm{x})$ votes for" positive class
  - $w_i < 0 \implies \psi_i(\mathrm{x})$ "votes against" positive class
  - $w_i \approx 0 \implies \psi(\mathrm{x})_i$ is irrelevant

Back to papayas

$$\begin{aligned} f(\mathrm{x}) = \big(\ & 1.3 \cdot \mathbb{1}(\mathrm{x} \text{ pulp is orange}) + \\ & \dots \\ & \mathbf{0} \cdot \mathbb{1}(\mathrm{x} \text{ is round}) + \\ & \dots \\ & -2.3 \cdot \mathbb{1}(\mathrm{x} \text{ is moldy})\ \big) \end{aligned}$$

- The interpretable features $\psi(\mathrm{x})$ can be semantically meaningful image segments, words, high-level concepts, *etc.*

- Once g_0 is obtained, LIME extracts an explanation for $\hat{y}_0 = g_0(\mathrm{x}_0)$ – this is easy, because g_0 is a **white-box** model – and uses it as an explanation for $y_0 = f(\mathrm{x}_0)$.

- If g_0 is a sparse linear model:

$$g_0(\mathrm{x}) = \sum_{j \in [d]} w_j \psi_j(\mathrm{x}) + b$$

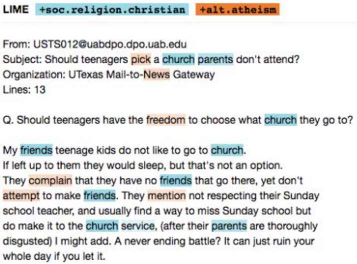
  - $w_i > 0 \implies \psi_i(\mathrm{x})$ votes for” positive class
  - $w_i < 0 \implies \psi_i(\mathrm{x})$ “votes against” positive class
  - $w_i \approx 0 \implies \psi(\mathrm{x})_i$ is irrelevant

Back to papayas

$$\begin{aligned} f(\mathrm{x}) = \Big(\; & 1.3 \cdot \mathbb{1}(\mathrm{x} \text{ pulp is orange}) + \\ & \dots \\ & \mathbf{0} \cdot \mathbb{1}(\mathrm{x} \text{ is round}) + \\ & \dots \\ & -2.3 \cdot \mathbb{1}(\mathrm{x} \text{ is moldy}) \Big) \end{aligned}$$

- The interpretable features $\psi(\mathrm{x})$ can be semantically meaningful image segments, words, high-level concepts, *etc.*

Examples

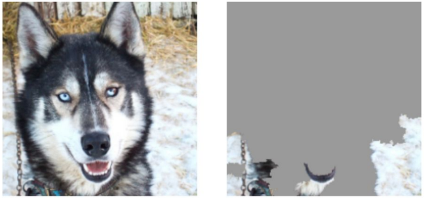


LIME +soc.religion.christian +alt.atheism

From: USTS012@uabdpo.dpo.uab.edu
Subject: Should teenagers pick a church parents don't attend?
Organization: UTexas Mail-to-News Gateway
Lines: 13

Q. Should teenagers have the freedom to choose what church they go to?

My friends teenage kids do not like to go to church.
If left up to them they would sleep, but that's not an option.
They complain that they have no friends that go there, yet don't
attempt to make friends. They mention not respecting their Sunday
school teacher, and usually find a way to miss Sunday school but
do make it to the church service, (after their parents are thoroughly
disgusted) I might add. A never ending battle? It can just ruin your
whole day if you let it.



(a) Husky classified as wolf (b) Explanation

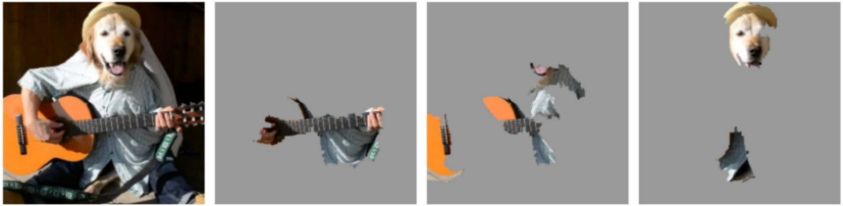
Figure 11: Raw data and explanation of a bad model's prediction in the "Husky vs Wolf" task.

Left: LIME explains document classification by highlighting relevant words.

Left: LIME explains document classification by highlighting relevant words.

Credit [Ribeiro et al., 2016]

Examples



(a) Original Image (b) Explaining *Electric guitar* (c) Explaining *Acoustic guitar* (d) Explaining *Labrador*

Figure 4: Explaining an image classification prediction made by Google's Inception neural network. The top 3 classes predicted are "Electric Guitar" ($p = 0.32$), "Acoustic guitar" ($p = 0.24$) and "Labrador" ($p = 0.21$)

Bonus: in the multi-class case ($c > 2$), learn a different g for each class $y \in [c]$ using a one-vs-all setup.

Credit [Ribeiro et al., 2016]

Question

There are:

- A team $T = \{1, \ldots, d\}$.
- A subset of players $S \subseteq T$.
- An evaluation function $v(S)$ that tells you how well the subset of players S performs as a whole.

How much does player $i \in T$ contribute to the performance of the whole team $v(T)$?

Source: depositphotos.

■ We can always assume that $v(\varnothing) = 0$.

■ No other assumptions. E.g., adding a player can make the team stronger or weaker, or v could be highly non-linear.

Shapley values

- **Setup**: $T = \{1, \dots, d\}$, $S \subseteq T$, $v(S) \in \mathbb{R}$.

- The contribution of player i depends on **the order in which it is added** to T! Let S be the players already there before i is added:
  - If the players in S are strong, the contribution of i will be minimal.
  - Vice versa, if they are weak, the contribution of i will be higher.

- The **marginal contribution** Δ of adding $i \in T \setminus S$ to $S \subseteq T$ is the value generated by adding i to S:

$$\Delta(i, S) := v(S \cup \{i\}) - v(S)$$

What if we don't know the **order** in which players are added to the team?

- The **Shapley value** ϕ of i is the **average marginal contribution** w.r.t all possible subsets of players coming before i:

$$\phi(i) := \frac{1}{d!} \sum_{\pi} \Delta(i, S_{i,\pi}), \qquad S_{i,\pi} := \{j \ : \ \pi(j) < \pi(i)\}$$

where π is any possible **permutation** of T and $S_{i,\pi}$ are the players coming before i according to π. The Shapley value of i expresses the **average impact** of player i on the output of $v(T)$.

Shapley values

- **Setup**: $T = \{1, \dots, d\}$, $S \subseteq T$, $v(S) \in \mathbb{R}$.

- The contribution of player i depends on **the order in which it is added** to T! Let S be the players already there before i is added:
  - If the players in S are strong, the contribution of i will be minimal.
  - Vice versa, if they are weak, the contribution of i will be higher.

- The **marginal contribution** Δ of adding $i \in T \setminus S$ to $S \subseteq T$ is the value generated by adding i to S:

$$\Delta(i, S) := v(S \cup \{i\}) - v(S)$$

What if we don't know the **order** in which players are added to the team?

- The **Shapley value** ϕ of i is the **average marginal contribution** w.r.t all possible subsets of players coming before i:

$$\phi(i) := \frac{1}{d!} \sum_{\pi} \Delta(i, S_{i,\pi}), \qquad S_{i,\pi} := \{j \ : \ \pi(j) < \pi(i)\}$$

where π is any possible **permutation** of T and $S_{i,\pi}$ are the players coming before i according to π. The Shapley value of i expresses the **average impact** of player i on the output of $v(T)$.

Shapley values

- **Setup**: $T = \{1, \ldots, d\}$, $S \subseteq T$, $v(S) \in \mathbb{R}$.

- The contribution of player i depends on **the order in which it is added** to T! Let S be the players already there before i is added:
  - If the players in S are strong, the contribution of i will be minimal.
  - Vice versa, if they are weak, the contribution of i will be higher.

- The **marginal contribution** Δ of adding $i \in T \setminus S$ to $S \subseteq T$ is the value generated by adding i to S:

$$\Delta(i, S) := v(S \cup \{i\}) - v(S)$$

What if we don't know the **order** in which players are added to the team?

- The **Shapley value** ϕ of i is the **average marginal contribution** w.r.t all possible subsets of players coming before i:

$$\phi(i) := \frac{1}{d!} \sum_{\pi} \Delta(i, S_{i,\pi}), \qquad S_{i,\pi} := \{j \; : \; \pi(j) < \pi(i)\}$$

where π is any possible **permutation** of T and $S_{i,\pi}$ are the players coming before i according to π. The Shapley value of i expresses the **average impact** of player i on the output of $v(T)$.

- The **Shapley value** of $\phi(i)$ is:

$$\phi(i) := \frac{1}{d!} \sum_{\pi} \Delta(i, S_{i,\pi})$$

- The value of $\Delta(i, S)$ is the **same** regardless of the order of the elements in S and in $T \setminus (S \cup \{i\})$.

- Hence, we can **rewrite the Shapley value** as:

$$\phi(i) = \sum_{S \subseteq T} \frac{|S|!(d - |S| - 1)!}{d!} \Delta(i, S)$$

The coefficients simply count how many ways the element in S and $T \setminus (S \cup \{i\})$ can be reordered. The new formula iterates over subsets of T (which are 2^d) rather than over permutations of T (which are $d!$), and $2^d \ll d!$ for large enough d. This **reduces the computational cost by an exponential factor**.

d	1	2	3	4	5	6	7	8	9	10
2^d	1	2	4	8	16	32	64	128	256	512
$d!$	1	1	2	6	24	120	720	5040	40320	362880

- The number of summands is still exponential in d though.

- The **Shapley value** of $\phi(i)$ is:

$$\phi(i) := \frac{1}{d!} \sum_{\pi} \Delta(i, S_{i,\pi})$$

- The value of $\Delta(i, S)$ is the **same** regardless of the order of the elements in S and in $T \setminus (S \cup \{i\})$.

- Hence, we can **rewrite the Shapley value** as:

$$\phi(i) = \sum_{S \subseteq T} \frac{|S|!(d - |S| - 1)!}{d!} \Delta(i, S)$$

The coefficients simply count how many ways the element in S and $T \setminus (S \cup \{i\})$ can be reordered. The new formula iterates over subsets of T (which are 2^d) rather than over permutations of T (which are $d!$), and $2^d \ll d!$ for large enough d. This **reduces the computational cost by an exponential factor**.

d	1	2	3	4	5	6	7	8	9	10
2^d	1	2	4	8	16	32	64	128	256	512
$d!$	1	1	2	6	24	120	720	5040	40320	362880

- The number of summands is still exponential in d though.

- The **Shapley value** of $\phi(i)$ is:

$$\phi(i) := \frac{1}{d!} \sum_{\pi} \Delta(i, S_{i,\pi})$$

- The value of $\Delta(i, S)$ is the **same** regardless of the order of the elements in S and in $T \setminus (S \cup \{i\})$.

- Hence, we can **rewrite the Shapley value** as:

$$\phi(i) = \sum_{S \subseteq T} \frac{|S|!(d - |S| - 1)!}{d!} \Delta(i, S)$$

The coefficients simply count how many ways the element in S and $T \setminus (S \cup \{i\})$ can be reordered. The new formula iterates over subsets of T (which are 2^d) rather than over permutations of T (which are $d!$), and $2^d \ll d!$ for large enough d. This **reduces the computational cost by an exponential factor**.

d	1	2	3	4	5	6	7	8	9	10
2^d	1	2	4	8	16	32	64	128	256	512
$d!$	1	1	2	6	24	120	720	5040	40320	362880

- The number of summands is still exponential in d though.

Properties of Shapley values

- Shapley values have a number of useful **properties**:

  Symmetry For any two players i, j, if $\Delta(i,S) = \Delta(j,S)$ for any $S \subseteq T$, then $\phi(i) = \phi(j)$.

  Dummy For any player i, if $\Delta(i,S) = 0$ for all S, then $\phi(i) = 0$.

  Additivity For any player i and value functions v, w, $\phi(i;v) = \phi(i;w) = \phi(i;v+w)$.

All of them make intuitive sense!

Shapley values for Input Relevance

- Use Shapley values to estimate **relevance** of ith input variable X_i on the score of class y. **Idea**:
  - Fix a classifier f and a decision $(\mathbf{x}, y)$.
  - Let $\mathrm{score}(\mathbf{x}, y) \in \mathbb{R}$ be the score associated by f to that prediction, e.g., the output of a neural network *before* or *after* the top softmax.
  - Team T are the input variables in $\mathbf{x}$.

- Define the **value $v(S)$ of a subset of input variables** $S \subseteq T$ as the average score obtained by fixing those inputs to the values in $\mathbf{x}$ and marginializing over the remaining variables [Štrumbelj and Kononenko, 2014, Lundberg and Lee, 2017]:

$$v(S) = \mathbb{E}_{\mathbf{X}_{\bar{S}}}[\mathrm{score}(\mathbf{x}) \mid \mathbf{X}_S = \mathbf{x}_S] = \int \mathrm{score}(\mathbf{x}_S, \mathbf{x}_{\bar{S}}) \underbrace{p(\mathbf{x}_{\bar{S}})}_{\textbf{prior}} d\mathbf{x}_{\bar{S}}$$

Here $\bar{S} = T \setminus S$ and $\mathbf{X}_S$, $\mathbf{x}_S$ are the corresponding variables and their values in $\mathbf{x}$. Essentially, this is equivalent to *fixing* $\mathbf{X}_s = \mathbf{x}_s$ *and looking at average impact of randomizing the remaining input variables.*

- The contribution of the ith input variable to a decision $(\mathbf{x}, y)$ according to f is the **SHAP value**, given by:

$$\phi(i) = \sum_{S \subseteq [d]} \frac{|S|!(d-|S|-1)!}{d!} \Delta(i, S) \tag{1}$$

$$= \sum_{S \subseteq [d]} \frac{|S|!(d-|S|-1)!}{d!} \Big(v(S \cup \{i\}) - v(S) \Big) \tag{2}$$

$$= \sum_{S \subseteq [d]} \frac{|S|!(d-|S|-1)!}{d!} \Big(\mathbb{E}_{\mathbf{X}_{\overline{S \cup \{i\}}}}[\mathrm{score}(\mathbf{x}) \mid \mathbf{X}_{S \cup \{i\}} = \mathbf{x}_{S \cup \{i\}}] - \mathbb{E}_{\mathbf{X}_{\bar{S}}}[\mathrm{score}(\mathbf{x}) \mid \mathbf{X}_S = \mathbf{x}_S] \Big) \tag{3}$$

  - Eq. (1) is the definition of Shapley value ϕ.
  - Eq. (2) replaces the marginal improvement Δ with its definition.
  - Eq. (3) replaces the value function v with the definition given in the previous slide.

- Computing SHAP values is **non-trivial**:[4]
  - The sum runs over 2^d subsets of variables.
  - For each subset, must solve an expectation.
  - Each expectation requires to integrate over the model.

[4] Exact computation of SHAP values is **intractable** even for simple models [Van den Broeck et al., 2021].

- The contribution of the ith input variable to a decision $(\mathbf{x}, y)$ according to f is the **SHAP value**, given by:

$$\phi(i) = \sum_{S\subseteq[d]} \frac{|S|!(d-|S|-1)!}{d!} \Delta(i, S) \tag{1}$$

$$= \sum_{S\subseteq[d]} \frac{|S|!(d-|S|-1)!}{d!} \Big(v(S \cup \{i\}) - v(S) \Big) \tag{2}$$

$$= \sum_{S\subseteq[d]} \frac{|S|!(d-|S|-1)!}{d!} \Big(\mathbb{E}_{\mathbf{X}_{\overline{S\cup\{i\}}}}[\mathrm{score}(\mathbf{x}) \mid \mathbf{X}_{S\cup\{i\}} = \mathbf{x}_{S\cup\{i\}}] - \mathbb{E}_{\mathbf{X}_{\bar{S}}}[\mathrm{score}(\mathbf{x}) \mid \mathbf{X}_S = \mathbf{x}_S] \Big) \tag{3}$$

  - Eq. (1) is the definition of Shapley value ϕ.
  - Eq. (2) replaces the marginal improvement Δ with its definition.
  - Eq. (3) replaces the value function v with the definition given in the previous slide.

- Computing SHAP values is **non-trivial**:[4]
  - The sum runs over 2^d subsets of variables.
  - For each subset, must solve an expectation.
  - Each expectation requires to integrate over the model.

[4] Exact computation of SHAP values is **intractable** even for simple models [Van den Broeck et al., 2021].

Computing SHAP

- Consider the SHAP equation (shortened for convenience):

$$\sum_{S\subseteq[d]} \frac{|S|!(d-|S|-1)!}{d!}\left(\underbrace{\mathbb{E}_{\mathbf{x}_{\overline{S\cup\{i\}}}}[\mathrm{score}(\mathbf{x}) \mid \mathbf{x}_{S\cup\{i\}}]}_{\text{expectation}} - \underbrace{\mathbb{E}_{\mathbf{x}_{\bar{S}}}[\mathrm{score}(\mathbf{x}) \mid \mathbf{x}_S]}_{\text{expectation}}\right)$$

The tricky bit is to evaluate the two **conditional expectations**. Once we have them all, the SHAP value is simply their (weighted) sum.

- Recall that $\mathbb{E}_{\mathbf{x}_{\bar{S}}}[\mathrm{score}(\mathbf{x})\mathbf{x}_S]$ is:

$$\int \mathrm{score}(\mathbf{x}_S, \mathbf{x}_{\bar{S}})p(\mathbf{x}_{\bar{S}} \mid \mathbf{x}_S)d\mathbf{x}_{\bar{S}}$$

We can approximate this via **sampling**! In practice, take the input $\mathbf{x}$ and repeatedly randomize $\mathbf{X}_{\bar{S}}$ obtaining k random vectors $\{\mathbf{x}_{\bar{S}}^{(1)}, \dots, \mathbf{x}_{\bar{S}}^{(k)}\}$. Then approximate the integral with a sum:

$$\frac{1}{k}\sum_{j=1}^{k} \mathrm{score}(\mathbf{x}_S, \mathbf{x}_{\bar{S}}^{(j)})$$

- The sampling step is similar to that of LIME, but SHAP values have a sound game theoretic interpretation.

Computing SHAP

- Consider the SHAP equation (shortened for convenience):

$$\sum_{S \subseteq [d]} \frac{|S|!(d-|S|-1)!}{d!} \left(\underbrace{\mathbb{E}_{\mathbf{X}_{\overline{S \cup \{i\}}}}[\text{score}(\mathbf{x}) \mid \mathbf{x}_{S \cup \{i\}}]}_{\text{expectation}} - \underbrace{\mathbb{E}_{\mathbf{X}_{\bar{S}}}[\text{score}(\mathbf{x}) \mid \mathbf{x}_S]}_{\text{expectation}} \right)$$

The tricky bit is to evaluate the two **conditional expectations**. Once we have them all, the SHAP value is simply their (weighted) sum.

- Recall that $\mathbb{E}_{\mathbf{X}_{\bar{S}}}[\text{score}(\mathbf{x})\mathbf{x}_S]$ is:

$$\int \text{score}(\mathbf{x}_S, \mathbf{x}_{\bar{S}}) p(\mathbf{x}_{\bar{S}} \mid \mathbf{x}_S) d\mathbf{x}_{\bar{S}}$$

We can approximate this via **sampling**! In practice, take the input $\mathbf{x}$ and repeatedly randomize $\mathbf{X}_{\bar{S}}$ obtaining k random vectors $\{\mathbf{x}_{\bar{S}}^{(1)}, \ldots, \mathbf{x}_{\bar{S}}^{(k)}\}$. Then approximate the integral with a sum:

$$\frac{1}{k} \sum_{j=1}^{k} \text{score}(\mathbf{x}_S, \mathbf{x}_{\bar{S}}^{(j)})$$

- The sampling step is similar to that of LIME, but SHAP values have a sound **game theoretic** interpretation.

Computing SHAP

- Consider the SHAP equation (shortened for convenience):

$$\sum_{S \subseteq [d]} \frac{|S|!(d-|S|-1)!}{d!} \left(\underbrace{\mathbb{E}_{\mathbf{X}_{\overline{S \cup \{i\}}}}[\mathrm{score}(\mathbf{x}) \mid \mathbf{x}_{S \cup \{i\}}]}_{\text{expectation}} - \underbrace{\mathbb{E}_{\mathbf{X}_{\bar{S}}}[\mathrm{score}(\mathbf{x}) \mid \mathbf{x}_S]}_{\text{expectation}} \right)$$

The tricky bit is to evaluate the two **conditional expectations**. Once we have them all, the SHAP value is simply their (weighted) sum.

- Recall that $\mathbb{E}_{\mathbf{X}_{\bar{S}}}[\mathrm{score}(\mathbf{x})\mathbf{x}_S]$ is:

$$\int \mathrm{score}(\mathbf{x}_S, \mathbf{x}_{\bar{S}}) p(\mathbf{x}_{\bar{S}} \mid \mathbf{x}_S) d\mathbf{x}_{\bar{S}}$$

We can approximate this via **sampling**! In practice, take the input $\mathbf{x}$ and repeatedly randomize $\mathbf{X}_{\bar{S}}$ obtaining k random vectors $\{\mathbf{x}_{\bar{S}}^{(1)}, \dots, \mathbf{x}_{\bar{S}}^{(k)}\}$. Then approximate the integral with a sum:

$$\frac{1}{k} \sum_{j=1}^{k} \mathrm{score}(\mathbf{x}_S, \mathbf{x}_{\bar{S}}^{(j)})$$

- The sampling step is similar to that of LIME, but SHAP values have a sound **game theoretic** interpretation.

Approximating SHAP

- **Trick #1**: Assume that input variables are **independent**. This yields:

$$\mathbb{E}_{\mathbf{X}_{\bar{S}}}[\mathrm{score}(\mathbf{x}) \mid \mathbf{X}_S = \mathbf{x}_S] \approx \mathbb{E}_{\mathbf{X}_{\bar{S}}}[\mathrm{score}(\mathbf{x})]$$

Brutal approximation, but makes expectation independent of $\mathbf{x}_S$, meaning that it has the same value for all inputs and as such it admits **caching**.

- **Trick #2**: Assume that score is **linear** (or approximate it using a linear model):

$$\mathbb{E}_{\mathbf{X}_{\bar{S}}}[\mathrm{score}(\mathbf{x})] \approx \mathrm{score}(\mathbb{E}_{\mathbf{X}_{\bar{S}}}[\mathbf{x}])$$

We can factor the score out of the expectation because the expectation is itself a linear operation.

This approximation yields an **enormous** speed-up, because it replaces the expectation (i.e., an integral over all possible values $\mathbf{x}_{\bar{S}}$) with the the score of the average element $\mathbb{E}[\mathbf{x}]$. This is trivial to do.

- **Trick #3**: use **model-specific approximations**, e.g., for random forests. Notice that the linear approximation above is also exact for linear models.

Approximating SHAP

- **Trick #1**: Assume that input variables are **independent**. This yields:

$$\mathbb{E}_{\mathbf{X}_{\bar{S}}}[\text{score}(\mathbf{x}) \mid \mathbf{X}_S = \mathbf{x}_S] \approx \mathbb{E}_{\mathbf{X}_{\bar{S}}}[\text{score}(\mathbf{x})]$$

Brutal approximation, but makes expectation independent of $\mathbf{x}_S$, meaning that it has the same value for all inputs and as such it admits **caching**.

- **Trick #2**: Assume that score is **linear** (or approximate it using a linear model):

$$\mathbb{E}_{\mathbf{X}_{\bar{S}}}[\text{score}(\mathbf{x})] \approx \text{score}(\mathbb{E}_{\mathbf{X}_{\bar{S}}}[\mathbf{x}])$$

We can factor the score out of the expectation because the expectation is itself a linear operation.

This approximation yields an **enormous** speed-up, because it replaces the expectation (i.e., an integral over all possible values $\mathbf{x}_{\bar{S}}$) with the the score of the average element $\mathbb{E}[\mathbf{x}]$. This is trivial to do.

- **Trick #3**: use **model-specific approximations**, e.g., for random forests. Notice that the linear approximation above is also exact for linear models.

Approximating SHAP

- **Trick #1**: Assume that input variables are **independent**. This yields:

$$\mathbb{E}_{\mathbf{X}_{\bar{S}}}[\text{score}(\mathbf{x}) \mid \mathbf{X}_S = \mathbf{x}_S] \approx \mathbb{E}_{\mathbf{X}_{\bar{S}}}[\text{score}(\mathbf{x})]$$

Brutal approximation, but makes expectation independent of $\mathbf{x}_S$, meaning that it has the same value for all inputs and as such it admits **caching**.

- **Trick #2**: Assume that score is **linear** (or approximate it using a linear model):

$$\mathbb{E}_{\mathbf{X}_{\bar{S}}}[\text{score}(\mathbf{x})] \approx \text{score}(\mathbb{E}_{\mathbf{X}_{\bar{S}}}[\mathbf{x}])$$

We can factor the score out of the expectation because the expectation is itself a linear operation.

This approximation yields an **enormous** speed-up, because it replaces the expectation (i.e., an integral over all possible values $\mathbf{x}_{\bar{S}}$) with the the score of the average element $\mathbb{E}[\mathbf{x}]$. This is trivial to do.

- **Trick #3**: use **model-specific approximations**, e.g., for random forests. Notice that the linear approximation above is also exact for linear models.

- LIME and SHAP are **model-agnostic**:
  - Only require access to the *predictions* of the model
  - Leverage this to probe f's decision surface near at selected points

- Computing an explanation can be **slow** and **high-variance**:
  - Large number of samples must be predicted
  - Explaining requires to fit a white-box model
  - Result depends statistically on choice of samples (& how well the kernel is tuned)

- *Are there more efficient alternatives?*

- LIME and SHAP are **model-agnostic**:
  - Only require access to the *predictions* of the model
  - Leverage this to probe f's decision surface near at selected points
- Computing an explanation can be **slow** and **high-variance**:
  - Large number of samples must be predicted
  - Explaining requires to fit a white-box model
  - Result depends statistically on choice of samples (& how well the kernel is tuned)
- *Are there more efficient alternatives?*

- LIME and SHAP are **model-agnostic**:
  - Only require access to the *predictions* of the model
  - Leverage this to probe f's decision surface near at selected points

- Computing an explanation can be **slow** and **high-variance**:
  - Large number of samples must be predicted
  - Explaining requires to fit a white-box model
  - Result depends statistically on choice of samples (& how well the kernel is tuned)

- *Are there more efficient alternatives?*

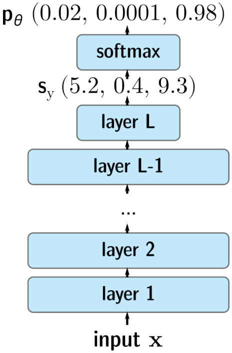
Typically the architecture **can be accessed**![a] Not *literally* a black-box.

For instance, a neural net looks like this:

$$f(\mathbf{x}) = \underset{y \in [c]}{\text{argmax}}\ p_\theta(y \mid \mathbf{x})$$

where $p_\theta(y \mid \mathbf{x})$ is a conditional distribution defined by a softmax activation layer on top of a dense "scoring" layer $\mathbf{s}(\mathbf{x}; \theta) \in \mathbb{R}^c$, i.e.

$$p_\theta(y \mid \mathbf{x}) = \mathrm{softmax}(\mathbf{s}(\mathbf{x}; \theta))_y \qquad \mathrm{softmax}(\mathbf{s})_y = \frac{\exp s_y(\mathbf{x}; \theta)}{\sum_{j \in [c]} \exp s_j(\mathbf{x}; \theta)}$$



$\mathbf{p}_\theta$ $(0.02,\ 0.0001,\ 0.98)$

softmax

$\mathbf{s}_\mathrm{y}$ $(5.2,\ 0.4,\ 9.3)$

layer L

layer L-1

...

layer 2

layer 1

input $\mathbf{x}$

In addition to the predictions, we also have access to the network's gradients. Is this useful?

[a] If this is not the case, for instace when querying a website, then it all depends on what queries can be asked to the model.

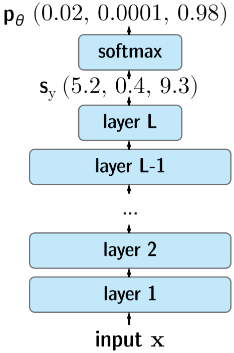
Typically the architecture **can be accessed**![a] Not *literally* a black-box.

For instance, a neural net looks like this:

$$f(\mathbf{x}) = \underset{y \in [c]}{\operatorname{argmax}}\ p_\theta(y \mid \mathbf{x})$$

where $p_\theta(y \mid \mathbf{x})$ is a conditional distribution defined by a softmax activation layer on top of a dense "scoring" layer $\mathbf{s}(\mathbf{x}; \theta) \in \mathbb{R}^c$, i.e.

$$p_\theta(y \mid \mathbf{x}) = \mathrm{softmax}(\mathbf{s}(\mathbf{x}; \theta))_y \qquad \mathrm{softmax}(\mathbf{s})_y = \frac{\exp s_y(\mathbf{x}; \theta)}{\sum_{j \in [c]} \exp s_j(\mathbf{x}; \theta)}$$


$\mathbf{p}_\theta$ (0.02, 0.0001, 0.98)
softmax
$\mathbf{s}_\mathrm{y}$ (5.2, 0.4, 9.3)
layer L
layer L-1
...
layer 2
layer 1
input $\mathbf{x}$

In addition to the predictions, we also have access to the network's gradients. Is this useful?

[a] If this is not the case, for instace when querying a website, then it all depends on what queries can be asked to the model.

Gradients ≈ Wiggling

Derivative as "Wiggling"

Let $f : \mathbb{R} \to \mathbb{R}$. The **derivative** of f w.r.t. x evaluated at $x_0 \in \mathbb{R}$ is:

$$f'(x_0) = \left(\frac{d}{dx} f(x)\right)\bigg|_{x=x_0} := \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

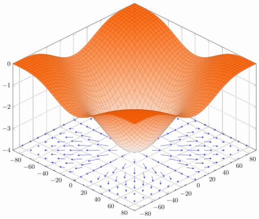
It measures how much **perturbing the input x by an infinitesimal amount ϵ affects the output of f at x_0**

Gradient as "Wiggling"

For $f : \mathbb{R}^d \to \mathbb{R}$, the **gradient** w.r.t. $\mathbf{x}$ is the vector of partial derivatives:

$$\nabla_{\mathbf{x}} f(\mathbf{x}_0) = \left(\nabla_{\mathbf{x}} f(\mathbf{x})\right)\bigg|_{\mathbf{x}=\mathbf{x}_0} = \left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_d}\right)$$

So it captures the effect of **perturbing each input** x_i, $i \in [d]$, on the output of $f(\mathbf{x})$



Hence, the **length** of the gradient vector, written as $\|\nabla_{\mathbf{x}} f(\mathbf{x}_0)\|$, measures the **sensitivity** of the output of f if we "wiggle" $\mathbf{x}_0$ around

Gradients ≈ Wiggling

Derivative as "Wiggling"

Let $f : \mathbb{R} \to \mathbb{R}$. The **derivative** of f w.r.t. x evaluated at $x_0 \in \mathbb{R}$ is:

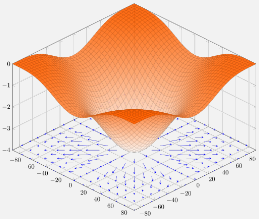
$$f'(x_0) = \left(\frac{d}{dx} f(x)\right)\Big|_{x=x_0} := \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

It measures how much **perturbing the input x by an infinitesimal amount ϵ affects the output of f at x_0**

Gradient as "Wiggling"

For $f : \mathbb{R}^d \to \mathbb{R}$, the **gradient** w.r.t. $\mathbf{x}$ is the vector of partial derivatives:

$$\nabla_{\mathbf{x}} f(\mathbf{x}_0) = \left(\nabla_{\mathbf{x}} f(\mathbf{x})\right)\Big|_{\mathbf{x}=\mathbf{x}_0} = \left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_d}\right)$$



So it captures the effect of **perturbing each input** x_i, $i \in [d]$, on the output of $f(\mathbf{x})$

Hence, the **length** of the gradient vector, written as $\|\nabla_{\mathbf{x}} f(\mathbf{x}_0)\|$, measures the **sensitivity** of the output of f if we "wiggle" $\mathbf{x}_0$ around

Input Gradients

- The absolute value of the **partial derivative** of p_θ w.r.t. x_i:

$$w_i := \frac{\partial}{\partial x_i} p_\theta(\mathbf{x}_0) \in \mathbb{R}$$

This conveys information about how much perturbing (wiggling) the *i*th input x_i from its current value in $\mathbf{x}_0$, while leaving all other inputs untouched, affects the score of class y.

- Just like for **linear models**:
  - $u_i > 0 \implies x_i$ correlates with, aka "votes for", class y
  - $u_i < 0 \implies x_i$ anti-correlates with, aka "votes against", class y
  - $|u_i| \approx 0 \implies x_i$ is irrelevant: changing it does not affect the probability of class y

References: [Baehrens et al., 2010, Simonyan et al., 2013]

Input Gradients

- The absolute value of the **partial derivative** of p_θ w.r.t. x_i:

$$w_i := \frac{\partial}{\partial x_i} p_\theta(\mathbf{x}_0) \in \mathbb{R}$$

This conveys information about how much perturbing (wiggling) the ith input x_i from its current value in $\mathbf{x}_0$, while leaving all other inputs untouched, affects the score of class y.

- Just like for **linear models**:
  - $u_i > 0 \implies x_i$ correlates with, aka "votes for", class y
  - $u_i < 0 \implies x_i$ anti-correlates with, aka "votes against", class y
  - $|u_i| \approx 0 \implies x_i$ is irrelevant: changing it does not affect the probability of class y

References: [Baehrens et al., 2010, Simonyan et al., 2013]

Input Gradients:

- Given $\mathbf{x}_0 \in \mathbb{R}^d$ and neural network $f(\mathbf{x})$ with conditional class distribution $p_\theta(Y \mid \mathbf{X})$
- Compute the all partial derivatives:

$$w_i := \left| \frac{\partial}{\partial x_i} p_\theta(\mathbf{x}_0) \right| \qquad i \in [d]$$

  This is easy to do using automatic differentiation packages (Tensorflow, Pytorch, JAX, ...). This gives you an $\mathbb{R}^d$ vector $\mathbf{w} = (w_1, \ldots, w_d)$.

- Transform this vector into an image → saliency map.

Input Gradients:

- Given $\mathbf{x}_0 \in \mathbb{R}^d$ and neural network $f(\mathbf{x})$ with conditional class distribution $p_\theta(Y \mid \mathbf{X})$
- Compute the all partial derivatives:

  $$w_i := \left| \frac{\partial}{\partial x_i} p_\theta(\mathbf{x}_0) \right| \qquad i \in [d]$$

  This is easy to do using automatic differentiation packages (Tensorflow, Pytorch, JAX, . . .).
  This gives you an $\mathbb{R}^d$ vector $\mathbf{w} = (w_1, \ldots, w_d)$.
- Transform this vector into an image → saliency map.

Gradient w.r.t. Input or Parameters?

- Input gradients:

$$\nabla_{\mathrm{x}} p_\theta(\mathrm{x}_0)$$

This conveys information about sensitivity of the output to perturbations of the **input**.

- This is **different** from the gradients used for *training* via SGD:

$$\nabla_\theta \ell(p_\theta, (\mathrm{x}_i, y_i))$$

This measures sensitivity of the **loss** to perturbations of the **parameters** (or weights)

- The first gradients are w.r.t. the model's output p_θ, the second ones are w.r.t. the **loss function** ℓ – they are *not* the same.
- Both methods identify relevant elements: relevant inputs (which have responsibility for a particular decision) vs relevant weights (which are responsible for how badly p_θ behaves on a particular training example (x_i, y_i))

Remark: the gradients $\nabla_\theta p_\theta(\mathrm{x})$ will be discussed later on.

Gradient w.r.t. Input or Parameters?

- Input gradients:

$$\nabla_{\mathbf{x}} p_\theta(\mathbf{x}_0)$$

This conveys information about sensitivity of the output to perturbations of the **input**.

- This is **different** from the gradients used for *training* via SGD:

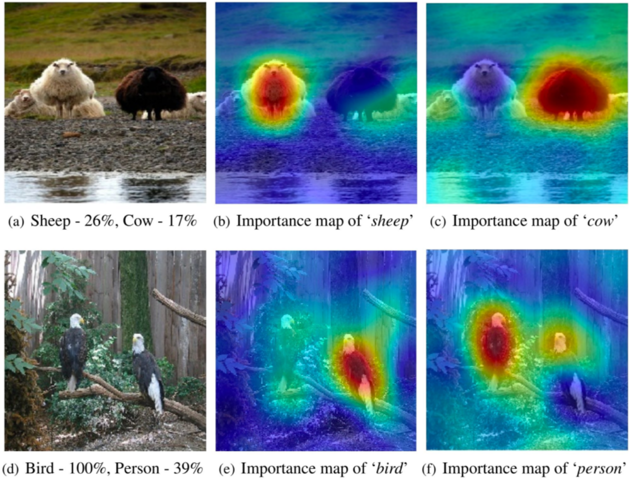
$$\nabla_\theta \ell(p_\theta, (\mathbf{x}_i, y_i))$$

This measures sensitivity of the **loss** to perturbations of the **parameters** (or weights)

- The first gradients are w.r.t. the model's output p_θ, the second ones are w.r.t. the **loss function** ℓ – they are *not* the same.
- Both methods identify relevant elements: relevant inputs (which have responsibility for a particular decision) vs relevant weights (which are responsible for how badly p_θ behaves on a particular training example $(\mathbf{x}_i, y_i)$)

Remark: the gradients $\nabla_\theta p_\theta(\mathbf{x})$ will be discussed later on.

Examples



(a) Sheep - 26%, Cow - 17% (b) Importance map of '*sheep*' (c) Importance map of '*cow*'

(d) Bird - 100%, Person - 39% (e) Importance map of '*bird*' (f) Importance map of '*person*'

Example images with predictions and saliency maps computed with (variants of) input gradients.

Aside: Feature interactions

- The input gradient $\nabla_{\mathrm{x}} p_\theta(\mathrm{x}_0)$ ignores feature interactions. This can be viewed with a Taylor decomposition of p_θ:

$$p_\theta(\mathrm{x} + \varepsilon) \approx p_\theta(\mathrm{x}) + \nabla_{\mathrm{x}}^\top \varepsilon$$

so for instance if the probability is large when both x_i and x_j are large but low when x_i, x_j are individually large, the input gradient will attribute relevance to either/both features **depending on x_0**.

- How to recover feature interations? Again, use the Taylor expansion:

$$p_\theta(\mathrm{x} + \varepsilon) \approx p_\theta(\mathrm{x}) + \nabla_{\mathrm{x}}^\top \varepsilon + \frac{1}{2} \varepsilon^\top H \varepsilon$$

where $H_{ij} = \frac{\partial^2}{\partial x_i \partial x_j} p_\theta$ is the **Hessian matrix**.

The term $|H_{ij}|$ encodes the contribution of the pair of features $x_i x_j$.

- Can be non-trivial to compute.

Aside: Feature interactions

- The input gradient $\nabla_{\mathrm{x}} p_\theta(\mathrm{x}_0)$ ignores feature interactions. This can be viewed with a Taylor decomposition of p_θ:

$$p_\theta(\mathrm{x} + \varepsilon) \approx p_\theta(\mathrm{x}) + \nabla_{\mathrm{x}}^\top \varepsilon$$

so for instance if the probability is large when both x_i and x_j are large but low when x_i, x_j are individually large, the input gradient will attribute relevance to either/both features **depending on x_0**.

- How to recover feature interations? Again, use the Taylor expansion:

$$p_\theta(\mathrm{x} + \varepsilon) \approx p_\theta(\mathrm{x}) + \nabla_{\mathrm{x}}^\top \varepsilon + \frac{1}{2}\varepsilon^\top H \varepsilon$$

where $H_{ij} = \frac{\partial^2}{\partial x_i \partial x_j} p_\theta$ is the **Hessian matrix**.

The term $|H_{ij}|$ encodes the contribution of the pair of features $x_i x_j$.

- Can be non-trivial to compute.

Aside: Feature interactions

- The input gradient $\nabla_{\mathrm{x}} p_\theta(\mathrm{x}_0)$ ignores feature interactions. This can be viewed with a Taylor decomposition of p_θ:

$$p_\theta(\mathrm{x} + \varepsilon) \approx p_\theta(\mathrm{x}) + \nabla_{\mathrm{x}}^\top \varepsilon$$

so for instance if the probability is large when both x_i and x_j are large but low when x_i, x_j are individually large, the input gradient will attribute relevance to either/both features **depending on x_0**.

- How to recover feature interations? Again, use the Taylor expansion:

$$p_\theta(\mathrm{x} + \varepsilon) \approx p_\theta(\mathrm{x}) + \nabla_{\mathrm{x}}^\top \varepsilon + \frac{1}{2}\varepsilon^\top H \varepsilon$$

where $H_{ij} = \frac{\partial^2}{\partial x_i \partial x_j} p_\theta$ is the **Hessian matrix**.

The term $|H_{ij}|$ encodes the contribution of the pair of features $x_i x_j$.

- Can be non-trivial to compute.

Two models f and f' are *functionally equivalent* if $p_\theta(\mathbf{x}) = p_\omega(\mathbf{x})$ for all inputs $\mathbf{x} \in \mathbb{R}^d$.

Implementation Invariance

An attribution method satisfies the **implementation invariance** axiom if, for every pair of functionally equivalent models f and f' and every input $\mathbf{x}$, it outputs the same attributions for both models.

- Input gradients **satisfy** implementation invariance.

Intuition: consider a neural network:

$$p_\theta(\mathbf{x}) = (h_L \circ h_{L-1} \circ \cdots \circ h_2 \circ h_1)(\mathbf{x})$$

where h_ℓ is the ℓ-th layer. The layers are implementation details. Gradients satisfy – and are computed in practice using – the **chain rule**:

$$\frac{\partial f}{\partial x_i} = \frac{\partial f}{\partial h_\ell} \frac{\partial h_\ell}{\partial x_i}$$

On the LHS, the gradient ignores implementation details, on the RHS it depends on them. Intuitively, the chain rule states that implementation details **do not matter** when computing gradients.

- Attribution methods that do not work analogously to the chain rule – for instance LRP and DeepLIFT – **violate** implementation invariance [Sundararajan et al., 2017]

Two models f and f' are *functionally equivalent* if $p_\theta(\mathbf{x}) = p_\omega(\mathbf{x})$ for all inputs $\mathbf{x} \in \mathbb{R}^d$.

Implementation Invariance

An attribution method satisfies the **implementation invariance** axiom if, for every pair of functionally equivalent models f and f' and every input $\mathbf{x}$, it outputs the same attributions for both models.

- Input gradients **satisfy** implementation invariance.

Intuition: consider a neural network:

$$p_\theta(\mathbf{x}) = (h_L \circ h_{L-1} \circ \cdots \circ h_2 \circ h_1)(\mathbf{x})$$

where h_ℓ is the ℓ-th layer. The layers are implementation details. Gradients satisfy – and are computed in practice using – the **chain rule**:

$$\frac{\partial f}{\partial x_i} = \frac{\partial f}{\partial h_\ell} \frac{\partial h_\ell}{\partial x_i}$$

On the LHS, the gradient ignores implementation details, on the RHS it depends on them. Intuitively, the chain rule states that implementation details **do not matter** when computing gradients.

- Attribution methods that do not work analogously to the chain rule – for instance LRP and DeepLIFT – **violate** implementation invariance [Sundararajan et al., 2017]

Two models f and f' are *functionally equivalent* if $p_\theta(\mathbf{x}) = p_\omega(\mathbf{x})$ for all inputs $\mathbf{x} \in \mathbb{R}^d$.

Implementation Invariance

An attribution method satisfies the **implementation invariance** axiom if, for every pair of functionally equivalent models f and f' and every input $\mathbf{x}$, it outputs the same attributions for both models.

- Input gradients **satisfy** implementation invariance.

Intuition: consider a neural network:

$$p_\theta(\mathbf{x}) = (h_L \circ h_{L-1} \circ \cdots \circ h_2 \circ h_1)(\mathbf{x})$$

where h_ℓ is the ℓ-th layer. The layers are implementation details. Gradients satisfy – and are computed in practice using – the **chain rule**:

$$\frac{\partial f}{\partial x_i} = \frac{\partial f}{\partial h_\ell} \frac{\partial h_\ell}{\partial x_i}$$

On the LHS, the gradient ignores implementation details, on the RHS it depends on them. Intuitively, the chain rule states that implementation details **do not matter** when computing gradients.

- Attribution methods that do not work analogously to the chain rule – for instance LRP and DeepLIFT – **violate** implementation invariance [Sundararajan et al., 2017]

Sensitivity

An attribution method satisfies the **sensitivity** axiom if, for every two inputs $\mathbf{x}$ and $\mathbf{x}'$ that differ in one feature (e.g., x_i) and have different predictions $p_\theta(\mathbf{x}) \neq p_\theta(\mathbf{x}')$, then the differing feature has non-zero responsibility.

- Unfortunately, input gradients **violate** sensitivity.

Consider a function [Sundararajan et al., 2017]:

$$f(x) = 1 - \mathrm{ReLU}(1 - x) = 1 - \max\{0, 1 - x\}$$

Pick $x = 0$ and $x' = 2$. Then $f(0) = 1 - 1 = 0$ and $f(2) = 1 - 0 = 1$, so the output at the two points is *different*. However, since f is "flat" at $x = 1$, **the gradient gives attribution** 0 **to** x:

$$f'(0) = 1 \qquad f'(1) = 0$$

Sensitivity

An attribution method satisfies the **sensitivity** axiom if, for every two inputs $\mathbf{x}$ and $\mathbf{x}'$ that differ in one feature (e.g., x_i) and have different predictions $p_\theta(\mathbf{x}) \neq p_\theta(\mathbf{x}')$, then the differing feature has non-zero responsibility.

- Unfortunately, input gradients **violate** sensitivity.

Consider a function [Sundararajan et al., 2017]:

$$f(x) = 1 - \mathrm{ReLU}(1-x) = 1 - \max\{0, 1-x\}$$

Pick $x = 0$ and $x' = 2$. Then $f(0) = 1 - 1 = 0$ and $f(2) = 1 - 0 = 1$, so the output at the two points is *different*. However, since f is "flat" at $x = 1$, **the gradient gives attribution** 0 **to** x:

$$f'(0) = 1 \qquad f'(1) = 0$$

Sensitivity

An attribution method satisfies the **sensitivity** axiom if, for every two inputs $\mathbf{x}$ and $\mathbf{x}'$ that differ in one feature (e.g., x_i) and have different predictions $p_\theta(\mathbf{x}) \neq p_\theta(\mathbf{x}')$, then the differing feature has non-zero responsibility.

- Unfortunately, input gradients **violate** sensitivity.

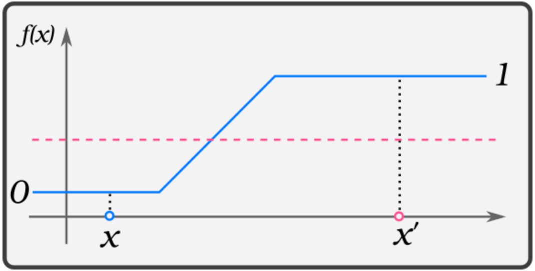
Consider a function [Sundararajan et al., 2017]:

$$f(x) = 1 - \mathrm{ReLU}(1 - x) = 1 - \max\{0, 1 - x\}$$

Pick $x = 0$ and $x' = 2$. Then $f(0) = 1 - 1 = 0$ and $f(2) = 1 - 0 = 1$, so the output at the two points is *different*. However, since f is "flat" at $x = 1$, **the gradient gives attribution 0 to x**:

$$f'(0) = 1 \qquad f'(1) = 0$$

- Unfortunately, input gradients **violate** sensitivity.



- Input gradients break sensitivity **because the prediction function may “flatten” at any fixed point** and thus have zero input gradient!
- This means that input gradients may ignore relevant features and focus on **irrelevant** ones!

Idea: instead of looking at the gradient only at x, consider a **baseline** x' and how the gradients **change** across the two.

- This gives **integrated gradients**:

$$\mathrm{intg}_i(\mathrm{x}) := (x_i - x_i') \cdot \int_0^1 \frac{\partial}{\partial x_i} p_\theta(\mathrm{x}' + \alpha \cdot (\mathrm{x} - \mathrm{x}'))d\alpha$$

Integrated gradients are the **path intergral** of the input gradients along the straightline path from the baseline x' to the target point x

The baseline x' is simply:

- A black or random image
- An all-zero embeddings for text models
- Typically, $\mathrm{x}' := \mathbb{E}_{p^*(\mathrm{x})}[\mathrm{x}]$ in theoretical papers.

Integrated gradients capture features that **account fo the change in output** between the baseline x' and the target point x. This intuitively matches what we do with *counterfactual reasoning*.

Idea: instead of looking at the gradient only at x, consider a **baseline** x' and how the gradients **change** across the two.

- This gives **integrated gradients**:

$$\mathrm{intg}_i(\mathbf{x}) := (x_i - x_i') \cdot \int_0^1 \frac{\partial}{\partial x_i} p_\theta(\mathbf{x}' + \alpha \cdot (\mathbf{x} - \mathbf{x}'))d\alpha$$

Integrated gradients are the **path intergral** of the input gradients along the straightline path from the baseline x' to the target point x

The baseline x' is simply:

- A black or random image
- An all-zero embeddings for text models
- Typically, $\mathrm{x}' := \mathbb{E}_{p^*(\mathrm{x})}[\mathrm{x}]$ in theoretical papers.

Integrated gradients capture features that **account fo the change in output** between the baseline x' and the target point x. This intuitively matches what we do with *counterfactual reasoning*.

Idea: instead of looking at the gradient only at x, consider a **baseline** x' and how the gradients **change** across the two.

- This gives **integrated gradients**:

$$\mathrm{intg}_i(\mathbf{x}) := (x_i - x_i') \cdot \int_0^1 \frac{\partial}{\partial x_i} p_\theta(\mathbf{x}' + \alpha \cdot (\mathbf{x} - \mathbf{x}'))d\alpha$$

Integrated gradients are the **path intergral** of the input gradients along the straightline path from the baseline x' to the target point x

The baseline x' is simply:

- A black or random image
- An all-zero embeddings for text models
- Typically, $\mathrm{x}' := \mathbb{E}_{p^*(\mathrm{x})}[\mathrm{x}]$ in theoretical papers.

Integrated gradients capture features that **account fo the change in output** between the baseline x' and the target point x. This intuitively matches what we do with *counterfactual reasoning*.

Completeness

An attribution method satisfies the **completeness** axiom if its attributions add up to the difference between the output of f at the target point x and the baseline x'.

In other words, the attributions “account for all changes”.

- Integrated gradients satisfy completeness, by the fundamental theorem of calculus:

$$\sum_{i \in [d]} \mathrm{intg}_i(\mathrm{x}) = p_\theta(\mathrm{x}) - p_\theta(\mathrm{x}')$$

i.e., that integrating the derivative gives the original function.

- Completeness implies sensitivity! If the sum of integrated integrals recovers the change in output, and only one feature changes between the baseline x' and the target output x, then that feature **must** have non-zero integrated gradent attribution!

- Integrated gradients satisfy sensitivity!

Completeness

An attribution method satisfies the **completeness** axiom if its attributions add up to the difference between the output of f at the target point x and the baseline x'.

In other words, the attributions "account for all changes".

- Integrated gradients satisfy completeness, by the fundamental theorem of calculus:

$$\sum_{i \in [d]} \mathrm{intg}_i(\mathrm{x}) = p_\theta(\mathrm{x}) - p_\theta(\mathrm{x}')$$

i.e., that integrating the derivative gives the original function.

- Completeness implies sensitivity! If the sum of integrated integrals recovers the change in output, and only one feature changes between the baseline x' and the target output x, then that feature **must** have non-zero integrated gradent attribution!

- Integrated gradients satisfy sensitivity!

Completeness

An attribution method satisfies the **completeness** axiom if its attributions add up to the difference between the output of f at the target point $\mathbf{x}$ and the baseline $\mathbf{x}'$.

In other words, the attributions "account for all changes".

- Integrated gradients **satisfy** completeness, by the fundamental theorem of calculus:

$$\sum_{i \in [d]} \mathrm{intg}_i(\mathbf{x}) = p_\theta(\mathbf{x}) - p_\theta(\mathbf{x}')$$

i.e., that integrating the derivative gives the original function.

- Completeness implies sensitivity! If the sum of integrated integrals recovers the change in output, and only one feature changes between the baseline $\mathbf{x}'$ and the target output $\mathbf{x}$, then that feature **must** have non-zero integrated gradent attribution!

- Integrated gradients **satisfy** sensitivity!

Two models f and f' are *functionally equivalent* if $p_\theta(\mathbf{x}) = p_\omega(\mathbf{x})$ for all inputs $\mathbf{x} \in \mathbb{R}^d$.

Implementation Invariance

An attribution method satisfies the **implementation invariance** axiom if, for every pair of functionally equivalent models f and f' and every input $\mathbf{x}$, it outputs the same attributions for both models.

- Integrated gradients **satisfy** implementation invariance!

Because they are defined on top of input gradients, which *are* implementation invariant.

Two models f and f' are *functionally equivalent* if $p_\theta(\mathbf{x}) = p_\omega(\mathbf{x})$ for all inputs $\mathbf{x} \in \mathbb{R}^d$.

Implementation Invariance

An attribution method satisfies the **implementation invariance** axiom if, for every pair of functionally equivalent models f and f' and every input $\mathbf{x}$, it outputs the same attributions for both models.

- Integrated gradients **satisfy** implementation invariance!

Because they are defined on top of input gradients, which *are* implementation invariant.

- Other properties satisfied by integrated gradients (and path integrals in general) are:

Dummy

If the output of f does not depend on a particular input variable x_i, then the attribution to that variable is zero.

Linearity

Take the linear combination of two networks p_θ and p_ω, i.e., $p(\mathbf{x}) = ap_\theta + bp_\omega$. Then the attributions of any input x_i for p are the linear combination of the attributions in p_θ and p_ω.

Symmetry Preserving

If the output of f is invariant to swapping the value of two input variables x_i and x_j, then an attribution method is symmetry preserving if it assignes the same attribution to both x_i and x_j.

- Computing integrated gradients is not straightforward:

$$\mathrm{intg}_i(\mathbf{x}) := (x_i - x_i') \cdot \int_0^1 \frac{\partial}{\partial x_i} p_\theta(\mathbf{x}' + \alpha \cdot (\mathbf{x} - \mathbf{x}'))d\alpha$$

This requires integration.

- Replace integral with finite summation:

$$(x_i - x_i') \cdot \sum_{k \in [n]} \frac{1}{n} \cdot \frac{\partial}{\partial x_i} p_\theta(\mathrm{x}' + \frac{k}{n} \cdot (\mathrm{x} - \mathrm{x}'))$$

This involves calling the autodiff package once for every step.

Trick: use a Jacobian operation to compute the input gradient at all steps of the computation jointly. If the autodiff package is smart enough, it will parallelize/batch-ize the computation.

- Computing integrated gradients is not straightforward:

$$\mathrm{intg}_i(\mathbf{x}) := (x_i - x_i') \cdot \int_0^1 \frac{\partial}{\partial x_i} p_\theta(\mathbf{x}' + \alpha \cdot (\mathbf{x} - \mathbf{x}'))d\alpha$$

This requires integration.

- Replace integral with finite summation:

$$(x_i - x_i') \cdot \sum_{k \in [n]} \frac{1}{n} \cdot \frac{\partial}{\partial x_i} p_\theta(\mathbf{x}' + \frac{k}{n} \cdot (\mathbf{x} - \mathbf{x}'))$$

This involves calling the autodiff package once for every step.

Trick: use a Jacobian operation to compute the input gradient at all steps of the computation jointly. If the autodiff package is smart enough, it will parallelize/batch-ize the computation.

- Computing integrated gradients is not straightforward:

$$\mathrm{intg}_i(\mathbf{x}) := (x_i - x_i') \cdot \int_0^1 \frac{\partial}{\partial x_i} p_\theta(\mathbf{x}' + \alpha \cdot (\mathbf{x} - \mathbf{x}')) d\alpha$$

This requires integration.

- Replace integral with finite summation:

$$(x_i - x_i') \cdot \sum_{k \in [n]} \frac{1}{n} \cdot \frac{\partial}{\partial x_i} p_\theta(\mathbf{x}' + \frac{k}{n} \cdot (\mathbf{x} - \mathbf{x}'))$$

This involves calling the autodiff package once for every step.

Trick: use a Jacobian operation to compute the input gradient at all steps of the computation jointly. If the autodiff package is smart enough, it will parallelize/batch-ize the computation.

Many Gradient-like Approaches

- Input gradients: $\frac{\partial}{\partial x_i} p_\theta(\mathbf{x})$
- Integrated gradients: integrate input gradients over a path between baseline $\mathbf{x}'$ and target point $\mathbf{x}$
- Gradient Times Input: $\mathbf{x} \odot \frac{\partial}{\partial x_i} p_\theta(\mathbf{x})$
- SmoothGrad: $\frac{1}{n} \sum_{k \in [n]} \frac{\partial}{\partial x_i} p_\theta(\mathbf{x} + \mathbf{u}_k)$ with $\mathbf{u}_k \sim \mathcal{N}(0, \sigma I)$.
- Guided Backpropagation: similar to input gradients, except that negative gradients are suppressed in the computation at all steps of the chain rule.
- Guided GradCAM: similar but for GradCAM.

- Saliency methods really look like edge detectors [Adebayo et al., 2018]

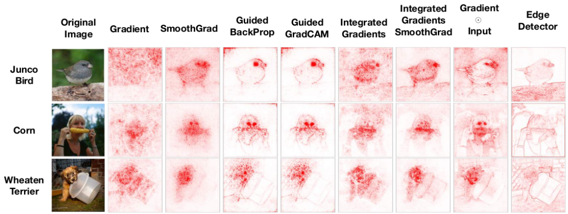


Figure 1: **Saliency maps for some common methods compared to an edge detector.** Saliency masks for 3 inputs for an Inception v3 model trained on ImageNet. We see that an edge detector produces outputs that are strikingly similar to the outputs of some saliency methods. In fact, edge detectors can also produce masks that highlight features which coincide with what appears to be relevant to a model's class prediction. We find that the methods most similar (see Appendix for SSIM metric) to an edge detector, i.e., Guided Backprop and its variants, show minimal sensitivity to our randomization tests.

Question: *do these methods provide extra insight into the model or do they just find edges (which do not depend on the model?)*

- This is what happens if we randomize the weights of different layers:

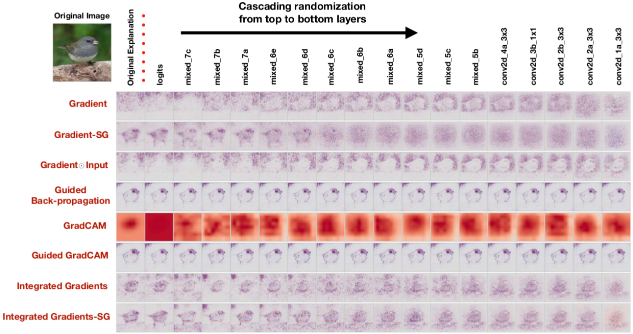


Figure 2: **Cascading randomization on Inception v3 (ImageNet).** Figure shows the original explanations (first column) for the Junco bird. Progression from left to right indicates complete randomization of network weights (and other trainable variables) up to that 'block' inclusive. We show images for 17 blocks of randomization. Coordinate (Gradient, mixed_7b) shows the gradient explanation for the network in which the top layers starting from Logits up to mixed_7b have been reinitialized. The last column corresponds to a network with completely reinitialized weights.

- Highlights the risks of judging explanation quality only visually.

Aside: Gradients vs LIME

Both input gradients and LIME estimate the sensibility of the output $p_\theta(\mathrm{x})$ to perturbations. Are they **related** somehow?

- **Yes!** Intuitively, if the kernel $k(\mathrm{x}_0, \mathrm{x}_i)$ is "pointy" enough, then LIME essentially becomes a **0-th order approximation of the input gradient**[5]

- Does this mean that LIME also fails to satisfy sensitivity? Not exactly, precisely because it looks at synthetic points different from x_0 – so in a sense these points play the role of baselines x'.

[5] Formally studied in [Garreau and Luxburg, 2020].

Aside: Gradients vs LIME

Both input gradients and LIME estimate the sensibility of the output $p_\theta(\mathrm{x})$ to perturbations. Are they **related** somehow?

- **Yes!** Intuitively, if the kernel $k(\mathrm{x}_0, \mathrm{x}_i)$ is "pointy" enough, then LIME essentially becomes a **0-th order approximation of the input gradient**[5]

- Does this mean that LIME also fails to satisfy sensitivity? Not exactly, precisely because it looks at synthetic points different from x_0 – so in a sense these points play the role of baselines x'.

[5] Formally studied in [Garreau and Luxburg, 2020].

Aside: Gradients vs LIME

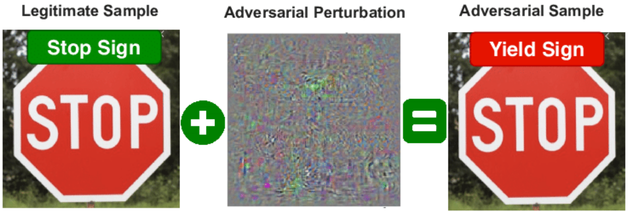
Both input gradients and LIME estimate the sensibility of the output $p_\theta(\mathbf{x})$ to perturbations. Are they **related** somehow?

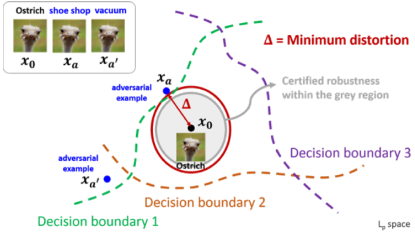
- **Yes!** Intuitively, if the kernel $k(\mathbf{x}_0, \mathbf{x}_i)$ is "pointy" enough, then LIME essentially becomes a **0-th order approximation of the input gradient**[5]

- Does this mean that LIME also fails to satisfy sensitivity? Not exactly, precisely because it looks at synthetic points different from $\mathbf{x}_0$ – so in a sense these points play the role of baselines $\mathbf{x}'$.

[5] Formally studied in [Garreau and Luxburg, 2020].

Aside: Adversarial Attacks

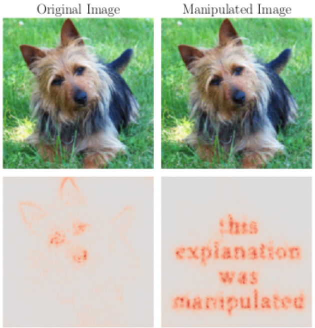




Intuition: build adversarial example x_{adv} by "searching" in the neighborhood of x, so that the difference is not perceptible to a human eye, while changing the output probability as much as possible. (Can be done by following the gradient.)

Image credit: IBM

Attribution approaches can be **fooled** by adversarial attacks too!



Algorithm:

- Given a target adversarial attribution map $\mathbf{a}_{\text{adv}}$ and a target input $\mathbf{x}$ with attribution $\mathbf{a}$
- Find a new input $\mathbf{x}_{\text{adv}}$ such that:
  - $\mathbf{x}_{\text{adv}}$ is perceptually similar to $\mathbf{x}$
  - Output of the network stays the same: $p_\theta(\mathbf{x}_{\text{adv}}) \approx p_\theta(\mathbf{x})$
  - Attribution is as close as possible to the adversarial map: $\mathrm{attr}(\mathbf{x}_{\text{adv}}) \approx \mathbf{a}_{\text{adv}}$

■ Simply apply gradient descent to optimize:

$$\min_{\mathbf{x}_{\text{adv}}} \|\mathrm{attr}(\mathbf{x}_{\text{adv}}) - \mathbf{a}_{\text{adv}}\| + \gamma \cdot \|p_\theta(Y \mid \mathbf{x}_{\text{adv}}) - p_\theta(Y \mid \mathbf{x})\|$$

In practice, do a small step of gradient descent, then project $\mathbf{x}_{\text{adv}}$ back close to $\mathbf{x}$.

Algorithm:

- Given a target adversarial attribution map $\mathbf{a}_{\text{adv}}$ and a target input $\mathbf{x}$ with attribution $\mathbf{a}$
- Find a new input $\mathbf{x}_{\text{adv}}$ such that:
  - $\mathbf{x}_{\text{adv}}$ is perceptually similar to $\mathbf{x}$
  - Output of the network stays the same: $p_\theta(\mathbf{x}_{\text{adv}}) \approx p_\theta(\mathbf{x})$
  - Attribution is as close as possible to the adversarial map: $\text{attr}(\mathbf{x}_{\text{adv}}) \approx \mathbf{a}_{\text{adv}}$

■ Simply apply gradient descent to optimize:

$$\min_{\mathbf{x}_{\text{adv}}} \|\text{attr}(\mathbf{x}_{\text{adv}}) - \mathbf{a}_{\text{adv}}\| + \gamma \cdot \|p_\theta(Y \mid \mathbf{x}_{\text{adv}}) - p_\theta(Y \mid \mathbf{x})\|$$

In practice, do a small step of gradient descent, then project $\mathbf{x}_{\text{adv}}$ back close to $\mathbf{x}$.

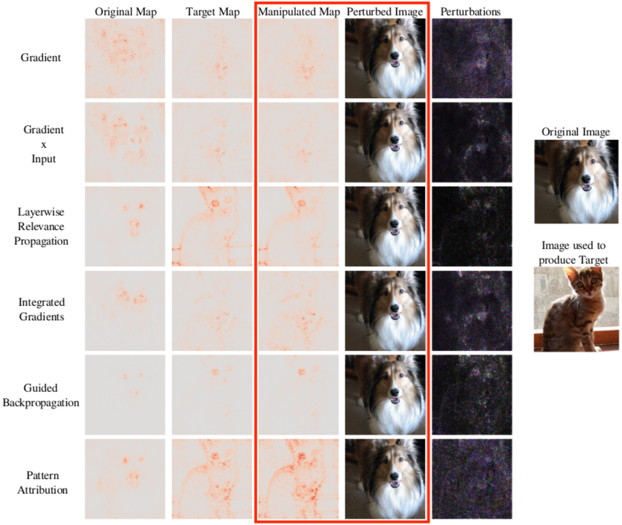


Figure 2: The explanation map of the cat is used as the target and the image of the dog is perturbed. The red box contains the manipulated images and the corresponding explanations. The first column corresponds to the original explanations of the unperturbed dog image. The target map, shown in the second column, is generated with the cat image. The last column visualizes the perturbations.

Take-away

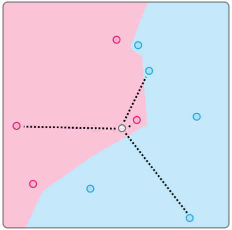
- Perturbation-based techniques (LIME, SHAP):
  - **Model-agnostic**: can be applied even to non-smooth black-box models (e.g., ensembles)
  - Supports mapping complex objects to **interpretable high-level features**
  - Requires sampling & training on a large number of points, which is **slow**
  - The estimated white-box model can have a **large variance**; depends strongly on hyper-parameters (# of samples, kernel, . . .) → can have **poor faithfulness**

- Gradient-based techniques:
  - Does not require sampling or retraining, which is **much faster**
  - Gradient can be **computed cheaply** using automatic differentiation packages
  - Since no translation takes place, the **explanation is usually stable & "faithful"**
  - **Model-specific**: can only be applied to models for which the gradient w.r.t. $\mathbf{x}$ exists almost everywhere, requires continuous inputs $\mathbf{x}$

Example-level Explanations

- **Input attributions** tell you what input variables or high-level concepts are responsible for a particular prediction $y_0 = f(\mathbf{x}_0)$.
  This explanation assumes that the model f is given and fixed, however this is not the case: f is learned from data, which may or may not be trustworthy.

- **Example attributions** tell you what **training examples** are responsible for a particular prediction.
  This is useful to figure out if the data that underlies a prediction is high-quality or not.



We are interested in answering the question: what training images, documents, etc. did determine the prediction that the model gave me?

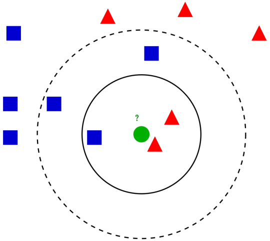
- Is this easy for general neural nets? **No**.

- Your “usual” feed-forward network consists of several steps:
  - The input $\mathbf{x}$ in some latent (or embedding) space $\mathbf{z} = h(\mathbf{x})$,
  - A prediction is made by performing logistic regression on $\mathbf{z}$, that is, $p(Y = 1 \mid \mathbf{x}) = \sigma\Big(\sum_i z_i w_i + b \Big)$.

Note that **none of these steps says anything useful about what training examples determined the prediction**: training examples are used for **learning** the parameters of the network, then completely **forgotten**. Moreover, they cannot be retrieved easily from the parameters themselves.

For some models it is **straightforward** to determine what training examples determine a particular prediction $y_0 = f(\mathbf{x})$.

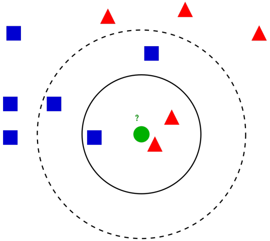
Example: *k* nearest neighbors (*k*NN)



- So long as *k* is sufficiently small, is **white-box**: the prediction is due to few examples that are close to $\mathbf{x}_0$ in terms of the distance function (e.g., Euclidean distance)

For some models it is **straightforward** to determine what training examples determine a particular prediction $y_0 = f(\mathbf{x})$.

Example: k nearest neighbors (kNN)



- So long as k is sufficiently small, is **white-box**: the prediction is due to few examples that are close to $\mathbf{x}_0$ in terms of the distance function (e.g., Euclidean distance)

Kernel Methods

For some models it is **straightforward** to determine what training examples determine a particular prediction $y_0 = f(\mathrm{x})$.

Example: kernel methods, e.g., support vector machines.

- An SVM is simply a **linear model** built on top of a feature function $\varphi : \mathbb{R}^d \to \mathbb{R}^k$:

$$\mathrm{score}(\mathrm{x}) = \sum_{j \in [k]} w_j \varphi_j(\mathrm{x}) + b$$

What makes it special is that (w, b) are the max-margin solution, obtained by solving a very special, convex learning problem.

- The **Representer Theorem** implies that this specific choice of parameters (w, b) admits a dual representation in terms of a kernel $k(\mathrm{x}, \mathrm{x}') := \langle \varphi(\mathrm{x}), \varphi(\mathrm{x}') \rangle$, namely:

$$\mathrm{score}(\mathrm{x}) = \sum_{i \in [m]} \alpha_i k(\mathrm{x}, \mathrm{x}_i)$$

where $S = \{(\mathrm{x}_i, y_i) : i \in [m]\}$ is the training set.

- This is the **analogue of linear models** in the dual!

Kernel Methods

For some models it is **straightforward** to determine what training examples determine a particular prediction $y_0 = f(\mathrm{x})$.

Example: kernel methods, e.g., support vector machines.

- An SVM is simply a **linear model** built on top of a feature function $\varphi : \mathbb{R}^d \to \mathbb{R}^k$:

$$\mathrm{score}(\mathrm{x}) = \sum_{j \in [k]} w_j \varphi_j(\mathrm{x}) + b$$

What makes it special is that (w, b) are the max-margin solution, obtained by solving a very special, convex learning problem.

- The **Representer Theorem** implies that this specific choice of parameters (w, b) admits a dual representation in terms of a kernel $k(\mathrm{x}, \mathrm{x}') := \langle \varphi(\mathrm{x}), \varphi(\mathrm{x}') \rangle$, namely:

$$\mathrm{score}(\mathrm{x}) = \sum_{i \in [m]} \alpha_i k(\mathrm{x}, \mathrm{x}_i)$$

where $S = \{(\mathrm{x}_i, y_i) : i \in [m]\}$ is the training set.

- This is the **analogue of linear models** in the dual!

Kernel Methods

For some models it is **straightforward** to determine what training examples determine a particular prediction $y_0 = f(\mathbf{x})$.

Example: kernel methods, e.g., support vector machines.

- An SVM is simply a **linear model** built on top of a feature function $\varphi : \mathbb{R}^d \to \mathbb{R}^k$:

$$\mathrm{score}(\mathbf{x}) = \sum_{j \in [k]} w_j \varphi_j(\mathbf{x}) + b$$

What makes it special is that $(\mathbf{w}, b)$ are the max-margin solution, obtained by solving a very special, convex learning problem.

- The **Representer Theorem** implies that this specific choice of parameters $(\mathbf{w}, b)$ admits a dual representation in terms of a kernel $k(\mathbf{x}, \mathbf{x}') := \langle \varphi(\mathbf{x}), \varphi(\mathbf{x}') \rangle$, namely:

$$\mathrm{score}(\mathbf{x}) = \sum_{i \in [m]} \alpha_i k(\mathbf{x}, \mathbf{x}_i)$$

where $S = \{(\mathbf{x}_i, y_i) : i \in [m]\}$ is the training set.

- This is the **analogue of linear models** in the dual!

Kernel Methods

For some models it is **straightforward** to determine what training examples determine a particular prediction $y_0 = f(\mathbf{x})$.

Example: kernel methods, e.g., support vector machines.

- An SVM is simply a **linear model** built on top of a feature function $\varphi : \mathbb{R}^d \to \mathbb{R}^k$:

$$\mathrm{score}(\mathbf{x}) = \sum_{j \in [k]} w_j \varphi_j(\mathbf{x}) + b$$

What makes it special is that $(\mathbf{w}, b)$ are the max-margin solution, obtained by solving a very special, convex learning problem.

- The **Representer Theorem** implies that this specific choice of parameters $(\mathbf{w}, b)$ admits a dual representation in terms of a kernel $k(\mathbf{x}, \mathbf{x}') := \langle \varphi(\mathbf{x}), \varphi(\mathbf{x}') \rangle$, namely:

$$\mathrm{score}(\mathbf{x}) = \sum_{i \in [m]} \alpha_i k(\mathbf{x}, \mathbf{x}_i)$$

where $S = \{(\mathbf{x}_i, y_i) : i \in [m]\}$ is the training set.

- This is the **analogue of linear models** in the dual!

Training examples $(\mathbf{x}_i, y_i)$ with $\alpha_i > 0$ are called **support vectors** (SV)

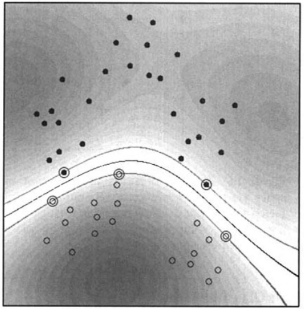


Figure 1.7 Example of an SV classifier found using a radial basis function kernel $k(x, x') = \exp(-\|x - x'\|^2)$ (here, the input space is $\mathcal{X} = [-1, 1]^2$). Circles and disks are two classes of training examples; the middle line is the decision surface; the outer lines precisely meet the constraint (1.25). Note that the SVs found by the algorithm (marked by extra circles) are not centers of clusters, but examples which are critical for the given classification task. Gray values code $|\sum_{i=1}^{m} y_i \alpha_i k(x, x_i) + b|$, the modulus of the argument of the decision function (1.35). The top and the bottom lines indicate places where it takes the value 1 (from [471]).

Intuitively, removing or perturbing an SV changes f, while changing a non-SV has no effect.

- kNN and SVMs do **not** quite answer the same question:
  - kNN identifies those training examples that affect **a particular prediction** $f(\mathbf{x}_0) = y_0$
  - α_i identifies those training examples on which **all of f** relies on

In order to obtain this information, one has to compute $\alpha_i \cdot k(\mathbf{x}_i, \mathbf{x}_0)$ for all i's: this takes $\mathbf{x}_0$ into consideration!

- kNN and SVMs do **not** quite answer the same question:
  - kNN identifies those training examples that affect **a particular prediction** $f(\mathrm{x}_0) = y_0$
  - α_i identifies those training examples on which **all of f** relies on

In order to obtain this information, one has to compute $\alpha_i \cdot k(\mathrm{x}_i, \mathrm{x}_0)$ for all i's: this takes x_0 into consideration!

What about general models?

How to generalize this to **general models**, including neural networks?

From Support Vectors to Relevant Examples

- For Support Vector Machines, an example x' is:
  - A **support vector** if *removing* it from the training set and retraining changes the decision surface of f.
  - A **support vector for x_0** if *removing* it from the training set and retraining changes the decision $f(\mathrm{x}_0) = y_0$ and, as a consequence, the predictive distribution $p_\theta(Y \mid \mathrm{x}_0$.

- For general models, we say that an example is **relevant for a decision $y_0 = f(\mathrm{x}_0)$** if *removing* it from the training set and retraining changes $f(\mathrm{x}_0)$, or in a more relaxed form, just $p_\theta(Y \mid \mathrm{x}_0)$.

Algorithm: Remove & Retrain

- Given:
  - A training set $S = \{(\mathrm{x}_i, y_i) : i \in [m]\}$
  - A classifier f trained on it
  - A target prediction $f(\mathrm{x}_0) = y_0$
- For each (x_i, y_i), remove it from the S, obtaining S_{-i}, learn f_{-1} on it
- The **relevance** of (x_i, y_i) is the difference between $p_\theta(y_0 \mid \mathrm{x}_0)$ and $p_{\theta_{-1}}(y_0 \mid \mathrm{x}_0)$

- This is the so-called **deletion metric**.

From Support Vectors to Relevant Examples

- For Support Vector Machines, an example $\mathbf{x}'$ is:
  - A **support vector** if *removing* it from the training set and retraining changes the decision surface of f.
  - A **support vector for $\mathbf{x}_0$** if *removing* it from the training set and retraining changes the decision $f(\mathbf{x}_0) = y_0$ and, as a consequence, the predictive distribution $p_\theta(Y \mid \mathbf{x}_0$.

- For general models, we say that an example is **relevant for a decision $y_0 = f(\mathbf{x}_0)$** if *removing* it from the training set and retraining changes $f(\mathbf{x}_0)$, or in a more relaxed form, just $p_\theta(Y \mid \mathbf{x}_0)$.

Algorithm: Remove & Retrain

- Given:
  - A training set $S = \{(\mathbf{x}_i, y_i) : i \in [m]\}$
  - A classifier f trained on it
  - A target prediction $f(\mathbf{x}_0) = y_0$
- For each $(\mathbf{x}_i, y_i)$, remove it from the S, obtaining S_{-i}, learn f_{-1} on it
- The **relevance** of $(\mathbf{x}_i, y_i)$ is the difference between $p_\theta(y_0 \mid \mathbf{x}_0)$ and $p_{\theta_{-1}}(y_0 \mid \mathbf{x}_0)$

- This is the so-called **deletion metric**.

From Support Vectors to Relevant Examples

- For Support Vector Machines, an example $\mathbf{x}'$ is:
  - A **support vector** if *removing* it from the training set and retraining changes the decision surface of f.
  - A **support vector for $\mathbf{x}_0$** if *removing* it from the training set and retraining changes the decision $f(\mathbf{x}_0) = y_0$ and, as a consequence, the predictive distribution $p_\theta(Y \mid \mathbf{x}_0$.

- For general models, we say that an example is **relevant for a decision $y_0 = f(\mathbf{x}_0)$** if *removing* it from the training set and retraining changes $f(\mathbf{x}_0)$, or in a more relaxed form, just $p_\theta(Y \mid \mathbf{x}_0)$.

Algorithm: Remove & Retrain

- Given:
  - A training set $S = \{(\mathbf{x}_i, y_i) : i \in [m]\}$
  - A classifier f trained on it
  - A target prediction $f(\mathbf{x}_0) = y_0$
- For each $(\mathbf{x}_i, y_i)$, remove it from the S, obtaining S_{-i}, learn f_{-1} on it
- The **relevance** of $(\mathbf{x}_i, y_i)$ is the difference between $p_\theta(y_0 \mid \mathbf{x}_0)$ and $p_{\theta_{-1}}(y_0 \mid \mathbf{x}_0)$

- This is the so-called **deletion metric**.

- **Example**: if for a prediction $f(\mathrm{x}) = \mathtt{dog}$ we have:

$$p_\theta(Y \mid \mathrm{x}_0) = \{\mathtt{dog} : 0.9, \mathtt{cat} : 0.1\} \qquad p_{\theta_{-i}}(Y \mid \mathrm{x}_0) = \{\mathtt{dog} : 0.4, \mathtt{cat} : 0.5\}$$

after removing x_i, then the deletion metric is $|0.9 - 0.4| = 0.5$. This example is quite influential!

Algorithm: Remove & Retrain

- Given:
  - A training set $S = \{(\mathrm{x}_i, y_i) : i \in [m]\}$
  - A classifier f trained on it
  - A target prediction $f(\mathrm{x}_0) = y_0$
- For each (x_i, y_i), remove it from the S, obtaining S_{-i}, **learn f_{-1} on it**
- The relevance of (x_i, y_i) is the difference between $p_\theta(y_0 \mid \mathrm{x}_0)$ and $p_{\theta_{-1}}(y_0 \mid \mathrm{x}_0)$

- Quite challenging if S is very large and/or f is a complex model (large nets can take hours/days to retrain)
- Especially because one must retrain once for *each* i!

- **Example**: if for a prediction $f(\mathbf{x}) = \mathtt{dog}$ we have:

$$p_\theta(Y \mid \mathbf{x}_0) = \{\mathtt{dog} : 0.9, \mathtt{cat} : 0.1\} \qquad p_{\theta_{-i}}(Y \mid \mathbf{x}_0) = \{\mathtt{dog} : 0.4, \mathtt{cat} : 0.5\}$$

after removing $\mathbf{x}_i$, then the deletion metric is $|0.9 - 0.4| = 0.5$. This example is quite influential!

Algorithm: Remove & Retrain

- Given:
  - A training set $S = \{(\mathbf{x}_i, y_i) : i \in [m]\}$
  - A classifier f trained on it
  - A target prediction $f(\mathbf{x}_0) = y_0$
- For each $(\mathbf{x}_i, y_i)$, remove it from the S, obtaining S_{-i}, **learn f_{-1} on it**
- The relevance of $(\mathbf{x}_i, y_i)$ is the difference between $p_\theta(y_0 \mid \mathbf{x}_0)$ and $p_{\theta_{-1}}(y_0 \mid \mathbf{x}_0)$

- Quite challenging if S is very large and/or f is a complex model (large nets can take hours/days to retrain)
- Especially because one must retrain once for *each* i!

Influence Functions

Influence functions (IFs) is a technique born in robust statistics that helps us to estimate the impact of a training examples **without retraining** [Koh and Liang, 2017].

- Fix a loss function ℓ (say, the cross-entropy loss) and a data set $S = \{z_i = (\mathrm{x}_i, y_i) \ : \ i = 1, \ldots, m\}$
- Let the classifier f be parameterized by θ
- Let θ_m be the parameters of the **empirical risk minimizer** on S:

$$\theta_m \leftarrow \underset{\theta}{\operatorname{argmin}} \ \frac{1}{m} \sum_k \ell(\theta, z_k)$$

- Let $\theta_m(z, \epsilon)$ be the parameters of the **empirical risk minimizer** after example z is upscaled by ϵ:

$$\theta_m(z, \epsilon) \leftarrow \underset{\theta}{\operatorname{argmin}} \ \left(\frac{1}{m} \sum_k \ell(\theta, z_k) \right) + \epsilon \ell(\theta, z)$$

- Notice that $\theta_m = \theta_m(z, 0)$.

■ Setting $\epsilon = \frac{1}{t}$ is equivalent to **deleting** example z!

Influence Functions

Influence functions (IFs) is a technique born in robust statistics that helps us to estimate the impact of a training examples **without retraining** [Koh and Liang, 2017].

- Fix a loss function ℓ (say, the cross-entropy loss) and a data set $S = \{z_i = (\mathrm{x}_i, y_i) \ : \ i = 1, \ldots, m\}$
- Let the classifier f be parameterized by θ
- Let θ_m be the parameters of the **empirical risk minimizer** on S:

$$\theta_m \leftarrow \operatorname*{argmin}_{\theta} \ \frac{1}{m} \sum_k \ell(\theta, z_k)$$

- Let $\theta_m(z, \epsilon)$ be the parameters of the **empirical risk minimizer** after example z is upscaled by ϵ:

$$\theta_m(z, \epsilon) \leftarrow \operatorname*{argmin}_{\theta} \ \left(\frac{1}{m} \sum_k \ell(\theta, z_k)\right) + \epsilon \ell(\theta, z)$$

- Notice that $\theta_m = \theta_m(z, 0)$.

■ Setting $\epsilon = \frac{1}{t}$ is equivalent to **deleting** example z!

Influence Functions

Influence functions (IFs) is a technique born in robust statistics that helps us to estimate the impact of a training examples **without retraining** [Koh and Liang, 2017].

- Fix a loss function ℓ (say, the cross-entropy loss) and a data set $S = \{z_i = (\mathrm{x}_i, y_i) \ : \ i = 1, \ldots, m\}$
- Let the classifier f be parameterized by θ
- Let θ_m be the parameters of the **empirical risk minimizer** on S:

$$\theta_m \leftarrow \underset{\theta}{\operatorname{argmin}} \ \frac{1}{m} \sum_k \ell(\theta, z_k)$$

- Let $\theta_m(z, \epsilon)$ be the parameters of the **empirical risk minimizer** after example z is upscaled by ϵ:

$$\theta_m(z, \epsilon) \leftarrow \underset{\theta}{\operatorname{argmin}} \ \left(\frac{1}{m} \sum_k \ell(\theta, z_k) \right) + \epsilon \ell(\theta, z)$$

- Notice that $\theta_m = \theta_m(z, 0)$.

■ Setting $\epsilon = \frac{1}{t}$ is equivalent to **deleting** example z!

Influence Functions

Influence functions (IFs) is a technique born in robust statistics that helps us to estimate the impact of a training examples **without retraining** [Koh and Liang, 2017].

- Fix a loss function ℓ (say, the cross-entropy loss) and a data set $S = \{z_i = (\mathbf{x}_i, y_i) \ : \ i = 1, \dots, m\}$
- Let the classifier f be parameterized by θ
- Let θ_m be the parameters of the **empirical risk minimizer** on S:

$$\theta_m \leftarrow \underset{\theta}{\operatorname{argmin}} \ \frac{1}{m} \sum_k \ell(\theta, z_k)$$

- Let $\theta_m(z, \epsilon)$ be the parameters of the **empirical risk minimizer** after example z is upscaled by ϵ:

$$\theta_m(z, \epsilon) \leftarrow \underset{\theta}{\operatorname{argmin}} \ \left(\frac{1}{m} \sum_k \ell(\theta, z_k) \right) + \epsilon \ell(\theta, z)$$

- Notice that $\theta_m = \theta_m(z, 0)$.

■ Setting $\epsilon = \frac{1}{t}$ is equivalent to **deleting** example z!

Influence Functions

Influence functions (IFs) is a technique born in robust statistics that helps us to estimate the impact of a training examples **without retraining** [Koh and Liang, 2017].

- Fix a loss function ℓ (say, the cross-entropy loss) and a data set $S = \{z_i = (\mathbf{x}_i, y_i) \ : \ i = 1, \ldots, m\}$
- Let the classifier f be parameterized by θ
- Let θ_m be the parameters of the **empirical risk minimizer** on S:

$$\theta_m \leftarrow \underset{\theta}{\operatorname{argmin}} \ \frac{1}{m} \sum_k \ell(\theta, z_k)$$

- Let $\theta_m(z, \epsilon)$ be the parameters of the **empirical risk minimizer** after example z is upscaled by ϵ:

$$\theta_m(z, \epsilon) \leftarrow \underset{\theta}{\operatorname{argmin}} \ \left(\frac{1}{m} \sum_k \ell(\theta, z_k) \right) + \epsilon \ell(\theta, z)$$

- Notice that $\theta_m = \theta_m(z, 0)$.

■ Setting $\epsilon = \frac{1}{t}$ is equivalent to **deleting** example z!

Influence Functions

Influence functions (IFs) is a technique born in robust statistics that helps us to estimate the impact of a training examples **without retraining** [Koh and Liang, 2017].

- Fix a loss function ℓ (say, the cross-entropy loss) and a data set $S = \{z_i = (\mathbf{x}_i, y_i) \ : \ i = 1, \ldots, m\}$
- Let the classifier f be parameterized by θ
- Let θ_m be the parameters of the **empirical risk minimizer** on S:

$$\theta_m \leftarrow \underset{\theta}{\operatorname{argmin}} \ \frac{1}{m} \sum_k \ell(\theta, z_k)$$

- Let $\theta_m(z, \epsilon)$ be the parameters of the **empirical risk minimizer** after example z is upscaled by ϵ:

$$\theta_m(z, \epsilon) \leftarrow \underset{\theta}{\operatorname{argmin}} \ \left(\frac{1}{m} \sum_k \ell(\theta, z_k) \right) + \epsilon \ell(\theta, z)$$

- Notice that $\theta_m = \theta_m(z, 0)$.

- Setting $\epsilon = \frac{1}{t}$ is equivalent to **deleting** example z!

Influence Functions

Influence functions (IFs) is a technique born in robust statistics that helps us to estimate the impact of a training examples **without retraining** [Koh and Liang, 2017].

- Fix a loss function ℓ (say, the cross-entropy loss) and a data set $S = \{z_i = (\mathrm{x}_i, y_i) \ : \ i = 1, \dots, m\}$
- Let the classifier f be parameterized by θ
- Let θ_m be the parameters of the **empirical risk minimizer** on S:

$$\theta_m \leftarrow \operatorname*{argmin}_{\theta} \ \frac{1}{m} \sum_k \ell(\theta, z_k)$$

- Let $\theta_m(z, \epsilon)$ be the parameters of the **empirical risk minimizer** after example z is upscaled by ϵ:

$$\theta_m(z, \epsilon) \leftarrow \operatorname*{argmin}_{\theta} \ \left(\frac{1}{m} \sum_k \ell(\theta, z_k)\right) + \epsilon \ell(\theta, z)$$

- Notice that $\theta_m = \theta_m(z, 0)$.

■ Setting $\epsilon = \frac{1}{t}$ is equivalent to **deleting** example z!

- Take a first-order Taylor expansion:

$$\theta_m(z,\epsilon) - \theta_m(z,0) \approx \epsilon \cdot \underbrace{\left(\frac{d}{d\epsilon}\theta_m(z,\epsilon)\Big|_{\epsilon=0} \right)}_{\text{influence function } \mathcal{I}(z)} \tag{4}$$

  - The effect on θ_m of **adding an example** z to S is:

$$\approx \frac{1}{t} \cdot \mathcal{I}(z)$$

  - The effect on θ_m of **removing an example** z from S is:

$$\approx -\frac{1}{t} \cdot \mathcal{I}(z)$$

- No retraining required! **But**... how do we compute $\mathcal{I}(z)$?

- Take a first-order Taylor expansion:

$$\theta_m(z,\epsilon) - \theta_m(z,0) \approx \epsilon \cdot \underbrace{\left(\frac{d}{d\epsilon}\theta_m(z,\epsilon)\bigg|_{\epsilon=0} \right)}_{\text{influence function } \mathcal{I}(z)} \tag{4}$$

  - The effect on θ_m of **adding an example** z to S is:

$$\approx \frac{1}{t} \cdot \mathcal{I}(z)$$

  - The effect on θ_m of **removing an example** z from S is:

$$\approx -\frac{1}{t} \cdot \mathcal{I}(z)$$

- No retraining required! **But**... how do we compute $\mathcal{I}(z)$?

- Take a first-order Taylor expansion:

$$\theta_m(z,\epsilon) - \theta_m(z,0) \approx \epsilon \cdot \underbrace{\left(\frac{d}{d\epsilon}\theta_m(z,\epsilon)\bigg|_{\epsilon=0} \right)}_{\text{influence function } \mathcal{I}(z)} \tag{4}$$

  - The effect on θ_m of **adding an example** z to S is:

$$\approx \frac{1}{t} \cdot \mathcal{I}(z)$$

  - The effect on θ_m of **removing an example** z from S is:

$$\approx -\frac{1}{t} \cdot \mathcal{I}(z)$$

- No retraining required! **But**. . . how do we compute $\mathcal{I}(z)$?

Idea: if the loss function $\ell(\theta, z)$ is **strongly convex** and **twice differentiable**, then [Koh and Liang, 2017]:

$$\mathcal{I}(z) = -H(\theta_m)^{-1} \nabla_\theta \ell(z, \theta_m)$$

where $H(\theta_m)$ is the **Hessian** computed on the data set S:

$$H(\theta_m) := \frac{1}{t} \sum_{k=1}^{t} \nabla_\theta^2 \ell(z_k, \theta_m), \qquad \nabla_\theta^2 \ell(z_k, \theta_m) = \left[\frac{\partial}{\partial \theta_s \partial \theta_t} \ell(z_k, \theta) \Big|_{\theta=\theta_m} \right]_{st}$$

- The above estimates the **change in parameters** – it's a **vector**. In order to convert this into a example **relevance score**, just compute its norm: $\|H(\theta_m)^{-1} \nabla_\theta \ell(z, \theta_m)\|$.

  - This can be derived formally for **convex** models
  - IFs were shown to be applicable to **non-convex** models (e.g., deep nets) too!

- The Hessian can be **computed** using Pytorch!

Idea: if the loss function $\ell(\theta, z)$ is **strongly convex** and **twice differentiable**, then [Koh and Liang, 2017]:

$$\mathcal{I}(z) = -H(\theta_m)^{-1} \nabla_\theta \ell(z, \theta_m)$$

where $H(\theta_m)$ is the **Hessian** computed on the data set S:

$$H(\theta_m) := \frac{1}{t} \sum_{k=1}^{t} \nabla_\theta^2 \ell(z_k, \theta_m), \qquad \nabla_\theta^2 \ell(z_k, \theta_m) = \left[\frac{\partial}{\partial \theta_s \partial \theta_t} \ell(z_k, \theta) \Big|_{\theta=\theta_m} \right]_{st}$$

- The above estimates the **change in parameters** – it's a **vector**. In order to convert this into a example **relevance score**, just compute its norm: $\|H(\theta_m)^{-1} \nabla_\theta \ell(z, \theta_m)\|$.
  - This can be derived formally for **convex** models
  - IFs were shown to be applicable to **non-convex** models (e.g., deep nets) too!
- The Hessian can be **computed** using Pytorch!

Idea: if the loss function $\ell(\theta, z)$ is **strongly convex** and **twice differentiable**, then [Koh and Liang, 2017]:

$$\mathcal{I}(z) = -H(\theta_m)^{-1} \nabla_\theta \ell(z, \theta_m)$$

where $H(\theta_m)$ is the **Hessian** computed on the data set S:

$$H(\theta_m) := \frac{1}{t} \sum_{k=1}^{t} \nabla_\theta^2 \ell(z_k, \theta_m), \qquad \nabla_\theta^2 \ell(z_k, \theta_m) = \left[\frac{\partial}{\partial \theta_s \partial \theta_t} \ell(z_k, \theta) \Big|_{\theta=\theta_m} \right]_{st}$$

■ The above estimates the **change in parameters** – it's a **vector**. In order to convert this into a example **relevance score**, just compute its norm: $\|H(\theta_m)^{-1} \nabla_\theta \ell(z, \theta_m)\|$.

- This can be derived formally for **convex** models
- IFs were shown to be applicable to **non-convex** models (e.g., deep nets) too!

■ The Hessian can be **computed** using Pytorch!

Recall that:

$$\mathcal{I}(z) = -H(\theta_m)^{-1}\nabla_\theta \ell(z, \theta_m)$$

- The above estimates the **change in parameters** – it's a **vector**. In order to convert this into a example **relevance score**, just compute its norm: $\|H(\theta_m)^{-1}\nabla_\theta \ell(z, \theta_m)\|$.

- What about the influence of removing z on the likelihood of z^*?

Using the **chain rule**, we get:

$$\begin{aligned} \frac{d}{d\epsilon} P(y^* \mid \mathbf{x}^*; \theta_m(z_k, \epsilon))\Big|_{\epsilon=0} &= \nabla_\theta P(y^* \mid \mathbf{x}^*; \theta_m)^\top \frac{d}{d\epsilon}\theta_m(z_k, \epsilon)\Big|_{\epsilon=0} \\ &= \nabla_\theta P(y^* \mid \mathbf{x}^*; \theta_m)^\top \mathcal{I}(z_k) \\ &= \underbrace{-\nabla_\theta P(y^* \mid \mathbf{x}^*; \theta_m)^\top H(\theta_m)^{-1}\nabla_\theta \ell(z, \theta_m)}_{\text{this is what we care about!}} \end{aligned}$$

This is a **scalar**, it approximates the **change in likelihood** at z^* by upscaling z by ϵ.

- Computing the quantity in blue is tricky. Doing so by naïvely computing the Hessian matrix and inverting it is far too slow, because the Hessian can be very large (it has size square in the number of parameters θ). Thankfully, there are very clever **fast approximations** [Koh and Liang, 2017]. We will skip them in the interest of time.

Recall that:

$$\mathcal{I}(z) = -H(\theta_m)^{-1}\nabla_\theta \ell(z, \theta_m)$$

- The above estimates the **change in parameters** – it's a **vector**. In order to convert this into a example **relevance score**, just compute its norm: $\|H(\theta_m)^{-1}\nabla_\theta \ell(z, \theta_m)\|$.

- What about the influence of removing z on the likelihood of z^*?

Using the **chain rule**, we get:

$$\begin{aligned} \frac{d}{d\epsilon} P(y^* \mid \mathbf{x}^*; \theta_m(z_k, \epsilon))\bigg|_{\epsilon=0} &= \nabla_\theta P(y^* \mid \mathbf{x}^*; \theta_m)^\top \frac{d}{d\epsilon}\theta_m(z_k, \epsilon)\bigg|_{\epsilon=0} \\ &= \nabla_\theta P(y^* \mid \mathbf{x}^*; \theta_m)^\top \mathcal{I}(z_k) \\ &= \underbrace{-\nabla_\theta P(y^* \mid \mathbf{x}^*; \theta_m)^\top H(\theta_m)^{-1}\nabla_\theta \ell(z, \theta_m)}_{\text{this is what we care about!}} \end{aligned}$$

This is a **scalar**, it approximates the **change in likelihood** at z^* by upscaling z by ϵ.

- Computing the quantity in **blue** is tricky. Doing so by naïvely computing the Hessian matrix and inverting it is far too slow, because the Hessian can be very large (it has size square in the number of parameters θ). Thankfully, there are very clever **fast approximations** [Koh and Liang, 2017]. We will skip them in the interest of time.

Recall that:

$$\mathcal{I}(z) = -H(\theta_m)^{-1}\nabla_\theta \ell(z, \theta_m)$$

- The above estimates the **change in parameters** – it's a **vector**. In order to convert this into a example **relevance score**, just compute its norm: $\|H(\theta_m)^{-1}\nabla_\theta \ell(z, \theta_m)\|$.

- What about the influence of removing z on the likelihood of z^*?

Using the **chain rule**, we get:

$$\begin{aligned}
\frac{d}{d\epsilon} P(y^* \mid \mathbf{x}^*; \theta_m(z_k, \epsilon))\Big|_{\epsilon=0} &= \nabla_\theta P(y^* \mid \mathbf{x}^*; \theta_m)^\top \frac{d}{d\epsilon} \theta_m(z_k, \epsilon)\Big|_{\epsilon=0} \\
&= \nabla_\theta P(y^* \mid \mathbf{x}^*; \theta_m)^\top \mathcal{I}(z_k) \\
&= \underbrace{-\nabla_\theta P(y^* \mid \mathbf{x}^*; \theta_m)^\top H(\theta_m)^{-1}\nabla_\theta \ell(z, \theta_m)}_{\text{this is what we care about!}}
\end{aligned}$$

This is a **scalar**, it approximates the **change in likelihood** at z^* by upscaling z by ϵ.

- Computing the quantity in **blue** is tricky. Doing so by naïvely computing the Hessian matrix and inverting it is far too slow, because the Hessian can be very large (it has size square in the number of parameters θ). Thankfully, there are very clever **fast approximations** [Koh and Liang, 2017]. We will skip them in the interest of time.

The **change in likelihood** is approximated as:

$$-\nabla_\theta P(y^* \mid \mathbf{x}^*; \theta_m)^\top H(\theta_m)^{-1} \nabla_\theta \ell(z, \theta_m)$$

with:

$$H(\theta_m) := \frac{1}{t} \sum_{k=1}^{t} \nabla^2_\theta \ell(z_k, \theta_m), \qquad \nabla^2_\theta \ell(z_k, \theta_m) = \left[\frac{\partial}{\partial\theta_s \partial\theta_t} \ell(z_k, \theta) \Big|_{\theta=\theta_m} \right]_{st}$$

- Cool but houses a heap of **numerical and computational issues**:
  - Requires computing H: a bunch of second-order derivatives for every k
  - H is $|\theta| \times |\theta|$: quadratic in the # of parameters, *huge* for even moderately sized networks
  - Requires computing H^{-1}: time cubic in $|\theta|$, may not be unique, may not be numerically stable, ...
  - Often must be computed once for every training point

The **change in likelihood** is approximated as:

$$-\nabla_\theta P(y^* \mid \mathrm{x}^*; \theta_m)^\top H(\theta_m)^{-1} \nabla_\theta \ell(z, \theta_m)$$

with:

$$H(\theta_m) := \frac{1}{t} \sum_{k=1}^{t} \nabla_\theta^2 \ell(z_k, \theta_m), \qquad \nabla_\theta^2 \ell(z_k, \theta_m) = \left[\frac{\partial}{\partial \theta_s \partial \theta_t} \ell(z_k, \theta) \Big|_{\theta=\theta_m} \right]_{st}$$

- Cool but houses a heap of **numerical and computational issues**:
  - Requires computing H: a bunch of second-order derivatives for every k
  - H is $|\theta| \times |\theta|$: quadratic in the # of parameters, *huge* for even moderately sized networks
  - Requires computing H^{-1}: time cubic in $|\theta|$, may not be unique, may not be numerically stable, ...
  - Often must be computed once for every training point

Idea: use **implicit Hessian-vector product** (HVPs)

Algorithm:

- Approximate $s^* := H(\theta_m)^{-1}\nabla_\theta P(y^* \mid \mathrm{x}^*; \theta_m)$ using an efficient HVP technique (see below)
- Compute $-s^* \cdot \nabla_\theta \ell(z, \theta_m)$

■ If we manage to do this, we also solve the second problem: s^* depends on test point z^* but it is **independent** from training point z, so we can **cache it**

Idea: use **implicit Hessian-vector product** (HVPs)

Algorithm:

- Approximate $s^* := H(\theta_m)^{-1}\nabla_\theta P(y^* \mid \mathbf{x}^*; \theta_m)$ using an efficient HVP technique (see below)
- Compute $-s^* \cdot \nabla_\theta \ell(z, \theta_m)$

■ If we manage to do this, we also solve the second problem: s^* depends on test point z^* but it is **independent** from training point z, so we can **cache it**

Idea: use **implicit Hessian-vector product** (HVPs)

Algorithm:

- Approximate $s^* := H(\theta_m)^{-1}\nabla_\theta P(y^* \mid \mathbf{x}^*; \theta_m)$ using an efficient HVP technique (see below)
- Compute $-s^* \cdot \nabla_\theta \ell(z, \theta_m)$

■ If we manage to do this, we also solve the second problem: s^* depends on test point z^* but it is **independent** from training point z, so we can **cache it**

LISSA

- HVP via **stochastic estimation** (LISSA) [Agarwal et al., 2017]

- Fix $j \in \mathbb{N}_0$ and define:

$$H_j^{-1} = \sum_{i=0}^{j} (I - H)^i$$

This is first j terms of the **Taylor expansion** of H^{-1},[6] and $H_j^{-1} \to H^{-1}$ as $j \to \infty$

[6] This is the so-called Neumann series: https://en.wikipedia.org/wiki/Neumann_series

LISSA

- HVP via **stochastic estimation** (LISSA) [Agarwal et al., 2017]

- Fix $j \in \mathbb{N}_0$ and define:

$$H_j^{-1} = \sum_{i=0}^{j} (I - H)^i$$

This is first j terms of the **Taylor expansion** of H^{-1},[6] and $H_j^{-1} \to H^{-1}$ as $j \to \infty$

[6] This is the so-called Neumann series: https://en.wikipedia.org/wiki/Neumann_series

LISSA

- This can be rewritten using a **recursion** as:

$$H_j^{-1} = I + (I - H)H_{j-1}^{-1}$$

This works because of the following identity:

$$\begin{aligned} H_j^{-1} &= \sum_{i=0}^{j} (I - H)^i \\ &= (I - H)^0 + \sum_{i=1}^{j} (I - H)^i \\ &= I + \sum_{i=1}^{j} (I - H)^i \\ &= I + (I - H) \underbrace{\sum_{i=0}^{j-1} (I - H)^i}_{H_{j-1}^{-1}} \end{aligned}$$

This means that by **iterating the recursion, we obtain H^{-1}**

- HVP via **stochastic estimation** (LISSA) [Agarwal et al., 2017]

Idea: $\nabla^2_\theta \ell(\theta, z_s$, where z_s is a random training point, is an **unbiased estimator** of H! In other words, $\mathbb{E}_s[\nabla^2_\theta \ell(\theta, z_s)] = H$

Algorithm for stochastic approximation of $H^{-1}\mathbf{v}$:

- Initialize $\tilde{H}_0^{-1}\mathbf{v} \leftarrow \mathbf{v}$
- Repeatedly apply the **recursion** (right-multiplied by $\mathbf{v}$):

$$\tilde{H}_j^{-1}\mathbf{v} \leftarrow I\mathbf{v} + (I - \nabla^2_\theta \ell(\theta, z_s))\tilde{H}_{j-1}^{-1}\mathbf{v}$$

  where $z_s \sim S$ is a single, random training set example.

- In the long run, this computes $\mathbb{E}_s[H_j^{-1}\mathbf{v}]$, which converges to $H^{-1}\mathbf{v}$ as $j \to \infty$ (= sample many points)

- Computing $\nabla^2_\theta \ell(\theta, z)$ is relatively cheap *if the model does not have too many parameters*

- HVP via **stochastic estimation** (LISSA) [Agarwal et al., 2017]

Idea: $\nabla^2_\theta \ell(\theta, z_s$, where z_s is a random training point, is an **unbiased estimator** of H! In other words, $\mathbb{E}_s[\nabla^2_\theta \ell(\theta, z_s)] = H$

Algorithm for stochastic approximation of $H^{-1}\mathbf{v}$:

- Initialize $\tilde{H}_0^{-1}\mathbf{v} \leftarrow \mathbf{v}$
- Repeatedly apply the **recursion** (right-multiplied by $\mathbf{v}$):

$$\tilde{H}_j^{-1}\mathbf{v} \leftarrow I\mathbf{v} + (I - \nabla^2_\theta \ell(\theta, z_s))\tilde{H}_{j-1}^{-1}\mathbf{v}$$

where $z_s \sim S$ is a single, random training set example.

- In the long run, this computes $\mathbb{E}_s[H_j^{-1}\mathbf{v}]$, which converges to $H^{-1}\mathbf{v}$ as $j \to \infty$ (= sample many points)

- Computing $\nabla^2_\theta \ell(\theta, z)$ is relatively cheap *if the model does not have too many parameters*

- HVP via **stochastic estimation** (LISSA) [Agarwal et al., 2017]

Idea: $\nabla^2_\theta \ell(\theta, z_s$, where z_s is a random training point, is an **unbiased estimator** of H! In other words, $\mathbb{E}_s[\nabla^2_\theta \ell(\theta, z_s)] = H$

Algorithm for stochastic approximation of $H^{-1}\mathbf{v}$:

- Initialize $\tilde{H}_0^{-1}\mathbf{v} \leftarrow \mathbf{v}$
- Repeatedly apply the **recursion** (right-multiplied by $\mathbf{v}$):

$$\tilde{H}_j^{-1}\mathbf{v} \leftarrow I\mathbf{v} + (I - \nabla^2_\theta \ell(\theta, z_s))\tilde{H}_{j-1}^{-1}\mathbf{v}$$

where $z_s \sim S$ is a single, random training set example.

- In the long run, this computes $\mathbb{E}_s[H_j^{-1}\mathbf{v}]$, which converges to $H^{-1}\mathbf{v}$ as $j \to \infty$ (= sample many points)

- Computing $\nabla^2_\theta \ell(\theta, z)$ is relatively cheap *if the model does not have too many parameters*

- HVP via **stochastic estimation** (LISSA) [Agarwal et al., 2017]

Idea: $\nabla^2_\theta \ell(\theta, z_s$, where z_s is a random training point, is an **unbiased estimator** of H! In other words, $\mathbb{E}_s[\nabla^2_\theta \ell(\theta, z_s)] = H$

Algorithm for stochastic approximation of $H^{-1}\mathbf{v}$:

- Initialize $\tilde{H}_0^{-1}\mathbf{v} \leftarrow \mathbf{v}$
- Repeatedly apply the **recursion** (right-multiplied by $\mathbf{v}$):

$$\tilde{H}_j^{-1}\mathbf{v} \leftarrow I\mathbf{v} + (I - \nabla^2_\theta \ell(\theta, z_s))\tilde{H}_{j-1}^{-1}\mathbf{v}$$

where $z_s \sim S$ is a single, random training set example.

- In the long run, this computes $\mathbb{E}_s[H_j^{-1}\mathbf{v}]$, which converges to $H^{-1}\mathbf{v}$ as $j \to \infty$ (= sample many points)

- Computing $\nabla^2_\theta \ell(\theta, z)$ is relatively cheap *if the model does not have too many parameters*

- HVP via **stochastic estimation** (LISSA) [Agarwal et al., 2017]

Idea: $\nabla^2_\theta \ell(\theta, z_s$, where z_s is a random training point, is an **unbiased estimator** of H! In other words, $\mathbb{E}_s[\nabla^2_\theta \ell(\theta, z_s)] = H$

Algorithm for stochastic approximation of $H^{-1}\mathbf{v}$:

- Initialize $\tilde{H}_0^{-1}\mathbf{v} \leftarrow \mathbf{v}$
- Repeatedly apply the **recursion** (right-multiplied by $\mathbf{v}$):

$$\tilde{H}_j^{-1}\mathbf{v} \leftarrow I\mathbf{v} + (I - \nabla^2_\theta \ell(\theta, z_s))\tilde{H}_{j-1}^{-1}\mathbf{v}$$

where $z_s \sim S$ is a single, random training set example.

- In the long run, this computes $\mathbb{E}_s[H_j^{-1}\mathbf{v}]$, which converges to $H^{-1}\mathbf{v}$ as $j \to \infty$ (= sample many points)

- Computing $\nabla^2_\theta \ell(\theta, z)$ is relatively cheap *if the model does not have too many parameters*

- Can we approximate the impact of removing $(\mathbf{x}_i, y_i)$ **without** retraining? Yes, using influence functions (IFs), a technique born in robust statistics that helps us to estimate the impact of a training examples **without retraining** [Koh and Liang, 2017]. They are quite advanced, but look them up if you are interested! ;-)

- How do IFs compare to leave-one-out retraining?

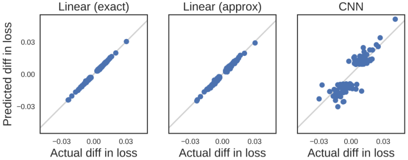


Figure 2. **Influence matches leave-one-out retraining.** We arbitrarily picked a wrongly-classified test point z_{test}, but this trend held more broadly. These results are from 10-class MNIST. **Left:** For each of the 500 training points z with largest $\left|\mathcal{I}_{\text{up,loss}}(z, z_{\text{test}})\right|$, we plotted $-\frac{1}{n} \cdot \mathcal{I}_{\text{up,loss}}(z, z_{\text{test}})$ against the actual change in test loss after removing that point and retraining. The inverse HVP was solved exactly with CG. **Mid:** Same, but with the stochastic approximation. **Right:** The same plot for a CNN, computed on the 100 most influential points with CG. For the actual difference in loss, we removed each point and retrained from $\tilde{\theta}$ for 30k steps.

- Looks pretty good!

Problem: H is seldom positive definite in practice:

- The model may be highly non-convex
- The loss may be non-convex
- Training often stopped early, before local optimum is reached
- Noisy data messes with the curvature of the decision surface

This means that H^{-1} does not technically exist and can be hard to "approximate"

This means that **computation of IFs to be unreliable** [Basu et al., 2020]: the recursion can **diverge**!

Solutions: standard remedies include:

- **Fine-tuning** θ using a second-order method like L-BFGS [Koh and Liang, 2017] → this is "cheating", second-order methods are quite slow (sometimes comparably to retraining)
- Implicitly preconditioning H^{-1} → this smooths out the curvature, may be *insufficient*
- Weight decay [Basu et al., 2020] keeps $\|\theta\|$ small, only indirectly affects 2nd order derivatives, may be *insufficient*

Problem: H is seldom positive definite in practice:

- The model may be highly non-convex
- The loss may be non-convex
- Training often stopped early, before local optimum is reached
- Noisy data messes with the curvature of the decision surface

This means that H^{-1} does not technically exist and can be hard to "approximate"

This means that **computation of IFs to be unreliable** [Basu et al., 2020]: the recursion can **diverge**!

Solutions: standard remedies include:

- **Fine-tuning** θ using a second-order method like L-BFGS [Koh and Liang, 2017] → this is "cheating", second-order methods are quite slow (sometimes comparably to retraining)
- Implicitly preconditioning H^{-1} → this smooths out the curvature, may be *insufficient*
- Weight decay [Basu et al., 2020] keeps $||\theta||$ small, only indirectly affects 2nd order derivatives, may be *insufficient*

Problem: H is seldom positive definite in practice:

- The model may be highly non-convex
- The loss may be non-convex
- Training often stopped early, before local optimum is reached
- Noisy data messes with the curvature of the decision surface

This means that H^{-1} does not technically exist and can be hard to "approximate"

This means that **computation of IFs to be unreliable** [Basu et al., 2020]: the recursion can **diverge**!

Solutions: standard remedies include:

- **Fine-tuning** θ using a second-order method like L-BFGS [Koh and Liang, 2017] → this is "cheating", second-order methods are quite slow (sometimes comparably to retraining)
- Implicitly preconditioning H^{-1} → this smooths out the curvature, may be *insufficient*
- Weight decay [Basu et al., 2020] keeps $\|\theta\|$ small, only indirectly affects 2nd order derivatives, may be *insufficient*

Idea: replace Hessian with Fisher information matrix $F(\theta)$:

$$F(\theta) := \frac{1}{t-1}\sum_{k=1}^{t-1}\mathbb{E}_{y\sim P(Y\,|\,\mathbf{x}_k,\theta)}\left[\nabla_\theta \log P(y\,|\,\mathbf{x}_k,\theta)\nabla_\theta \log P(y\,|\,\mathbf{x}_k,\theta)^\top\right]$$

- The FIM is useful because:
  - Positive semi-definite, so inverse always “almost exists” & numerically stabler to approximate
  - It the model approximates the data distribution, then $F(\theta) \approx H(\theta)$
  - Even if this does not hold, $F(\theta)$ still captures useful curvature information

Problem: both H and F are $|\theta| \times |\theta|$, very large. Can restrict either to just some layers of the network, e.g., the top layer.

Idea: replace Hessian with Fisher information matrix $F(\theta)$:

$$F(\theta) := \frac{1}{t-1}\sum_{k=1}^{t-1}\mathbb{E}_{y\sim P(Y\,|\,\mathbf{x}_k,\theta)}\left[\nabla_\theta \log P(y\,|\,\mathbf{x}_k,\theta)\nabla_\theta \log P(y\,|\,\mathbf{x}_k,\theta)^\top\right]$$

- The FIM is useful because:
  - Positive semi-definite, so inverse always "almost exists" & numerically stabler to approximate
  - It the model approximates the data distribution, then $F(\theta) \approx H(\theta)$
  - Even if this does not hold, $F(\theta)$ still captures useful curvature information

Problem: both H and F are $|\theta| \times |\theta|$, very large. Can restrict either to just some layers of the network, e.g., the top layer.

Idea: replace Hessian with Fisher information matrix $F(\theta)$:

$$F(\theta) := \frac{1}{t-1} \sum_{k=1}^{t-1} \mathbb{E}_{y \sim P(Y \mid \mathbf{x}_k, \theta)} \left[\nabla_\theta \log P(y \mid \mathbf{x}_k, \theta) \nabla_\theta \log P(y \mid \mathbf{x}_k, \theta)^\top \right]$$

- The FIM is useful because:
  - Positive semi-definite, so inverse always “almost exists” & numerically stabler to approximate
  - It the model approximates the data distribution, then $F(\theta) \approx H(\theta)$
  - Even if this does not hold, $F(\theta)$ still captures useful curvature information

Problem: both H and F are $|\theta| \times |\theta|$, very large. Can restrict either to just some layers of the network, e.g., the top layer.

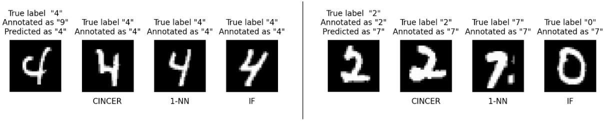


Figure 1: Suspicious example and counter-examples selected using (from left to right) CINCER, 1-NN and influence functions (IF), on noisy MNIST. **Left**: the suspicious example is mislabeled, the machine's suspicion is supported by a clean counter-example. **Right**: the suspicious example is not mislabeled, the machine is wrongly suspicious because the counter-example is mislabeled. CINCER's counter-example is contrastive and influential; 1-NN's is not influential and IF's is not pertinent, see desiderata D1–D3 below.

One simple technique to speed up computation:

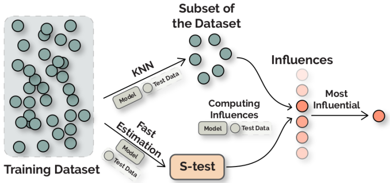


Figure 1: Workflow of our FASTIF w.r.t. a test data-point. First a subset of data-points are selected from the entire training set using kNN to reduce search space, then the inverse Hessian-vector product (s_{test}) is estimated based on Sec. 5.2. The influence values of data-points are computed using the outputs from these two steps. Finally, the most influential data-point(s) are returned.

This also avoids identifying far-away **outliers** that say little about (are very different from) the test point

Take-away

- Some approaches are **white-box** when it comes to example-based why questions
- Other – like neural nets – are **black-box**, but we can use influence functions to understand what examples they rely on for making predictions.
  - IFs are **sound** for convex models & can be meaningful for non-convex models too
  - IFs are not cheap to compute, but there are **fast approximations**.
  - IFs can be **brittle**, especially with noisy data
  - Influential examples tend to be outliers, restrict search to neighbors

Counterfactual Explanations

Limits of Factual Explanations

Example

You file a loan request at your bank. Unfortunately, the loan is refused. Your bank gives you a factual explanation that clarifies how the decision was based on your education level and work history. Which of these variables should you work on to increase the chance of getting a loan? For instance, in order to get the loan, should you i) obtain an additional master degree, or ii) look for a more stable or well-payed job?

- **Factual** explanations explain why a particular decision $y_0 = f(\mathrm{x}_0)$ was made. However, they say nothing about how to change x_0 to obtain a different, more desirable outcome y_1. In other words, they are **not actionable**.

- **Counterfactual** explanations tell you exactly that!

■ A **counterfactual** explanation tells you why a particular outcome (prediction) y_0 was obtained instead of a (more desirable) alternative y_1, e.g., "loan rejected" rather than "loan approved".

Intuition:

1. Given x_0, look for an alternative input $\mathrm{x}_1 \in \mathbb{R}^d$ such that:

$$f(\mathrm{x}_1) = y_1$$

where y_1 is either a specific, more desirable outcome, or simply any other outcome $y_1 \neq y_0$, depending on your needs.

2. x_1 should **differ in few attributes** from x_0, so that it is easier for the user to **understand** the difference and possibly **act** upon it.
3. Summarize the difference between x_0 and x_1 by, for instance, identifying the variables that differ between them:

$$\{i \in [d] \ : \ x_{0i} \neq x_{1i}\}$$

■ There is plenty of research on how **we** use counterfactuals (and what counterfactuals work best) in psychology, see [Dai et al., 2022].

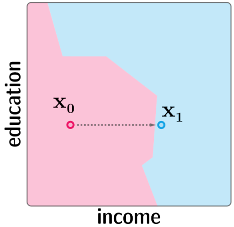


Illustration of a counterfactual example x_1 for a target example x_0. Notice that the two examples differ in few attributes – or rather, just one, **income** – and therefore are "close" in this sense, but have different labels. Here, **red** means "reject" while **blue** means "accept".

- A **counterfactual** explanation tells you why a particular outcome (prediction) y_0 was obtained instead of a (more desirable) alternative y_1, e.g., “loan rejected” rather than “loan approved”.

Intuition:

1. Given $\mathbf{x}_0$, look for an alternative input $\mathbf{x}_1 \in \mathbb{R}^d$ such that:

$$f(\mathbf{x}_1) = y_1$$

   where y_1 is either a specific, more desirable outcome, or simply any other outcome $y_1 \neq y_0$, depending on your needs.

2. $\mathbf{x}_1$ should **differ in few attributes** from $\mathbf{x}_0$, so that it is easier for the user to **understand** the difference and possibly **act** upon it.
3. Summarize the difference between $\mathbf{x}_0$ and $\mathbf{x}_1$ by, for instance, identifying the variables that differ between them:

$$\{i \in [d] \ : \ x_{0i} \neq x_{1i}\}$$

- There is plenty of research on how **we** use counterfactuals (and what counterfactuals work best) in psychology, see [Dai et al., 2022].

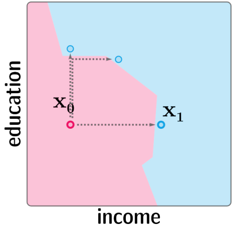


Illustration of a counterfactual example $\mathbf{x}_1$ for a target example $\mathbf{x}_0$. What is the **best** counterfactual?

Finding a Counterfactual

Given original input $\mathbf{x}_0$ predicted as y_0, find counterfactual input $\mathbf{x}_1$ with alternative label y_0 by solving:

$$\mathbf{x}_1 \leftarrow \operatorname*{argmin}_{\mathbf{x} \in \mathbb{R}^d} \|\mathbf{x}_0 - \mathbf{x}_1\| \qquad \text{s.t. } f(\mathbf{x}_1) = y_1 \qquad (\text{or } f(\mathbf{x}_1) \neq f(\mathbf{x}_0))$$

In other words, find $\mathbf{x}_1$ that is as close as possible to $\mathbf{x}_0$ but has a different label.

- The norm $\|\cdot\|$ is usually chosen to be one of the following:

$$\|\mathbf{x}_0 - \mathbf{x}_1\|_2 = \sqrt{\sum_i (x_{0,i} - x_{1,i})^2}, \qquad \|\mathbf{x}_0 - \mathbf{x}_1\|_1 = \sum_i |x_{0,i} - x_{1,i}|$$

The left one is the usual Euclidean norm (i.e., the normal distance between points). The right one is the L_1 norm, and tends to give counterfactuals that have **fewer differences** from the original.

Finding a Counterfactual

Given original input $\mathbf{x}_0$ predicted as y_0, find counterfactual input $\mathbf{x}_1$ with alternative label y_0 by solving:

$$\mathbf{x}_1 \leftarrow \underset{\mathbf{x}\in\mathbb{R}^d}{\text{argmin}} \ \|\mathbf{x}_0 - \mathbf{x}_1\|$$
$$\text{s.t. } f(\mathbf{x}_1) = y_1 \qquad (\text{or } f(\mathbf{x}_1) \neq f(\mathbf{x}_0))$$

In other words, find $\mathbf{x}_1$ that is as close as possible to $\mathbf{x}_0$ but has a different label.

- The norm $\|\cdot\|$ is usually chosen to be one of the following:

$$\|\mathbf{x}_0 - \mathbf{x}_1\|_2 = \sqrt{\sum_i (x_{0,i} - x_{1,i})^2}, \qquad \|\mathbf{x}_0 - \mathbf{x}_1\|_1 = \sum_i |x_{0,i} - x_{1,i}|$$

The left one is the usual Euclidean norm (i.e., the normal distance between points). The right one is the L_1 norm, and tends to give counterfactuals that have **fewer differences** from the original.

Finding a Counterfactual

Given original input $\mathbf{x}_0$ predicted as y_0, find counterfactual input $\mathbf{x}_1$ with alternative label y_0 by solving:

$$\mathbf{x}_1 \leftarrow \underset{\mathbf{x} \in \mathbb{R}^d}{\operatorname{argmin}} \|\mathbf{x}_0 - \mathbf{x}_1\|$$

$$\text{s.t. } f(\mathbf{x}_1) = y_1 \qquad (\text{or } f(\mathbf{x}_1) \neq f(\mathbf{x}_0))$$

In other words, find $\mathbf{x}_1$ that is as close as possible to $\mathbf{x}_0$ but has a different label.

- How can we **solve** this? Three common options:
  - Use **gradient descent** (or, "if you have a hammer, every problem looks like a nail")
  - Use **model-specific procedures**.
  - Use **mathematical programming**.

Finding a Counterfactual

Given original input x_0 predicted as y_0, find counterfactual input x_1 with alternative label y_0 by solving:

$$\mathrm{x}_1 \leftarrow \operatorname*{argmin}_{\mathrm{x}\in\mathbb{R}^d} \|\mathrm{x}_0 - \mathrm{x}_1\|$$
$$\text{s.t. } f(\mathrm{x}_1) = y_1 \qquad (\text{or } f(\mathrm{x}_1) \neq f(\mathrm{x}_0))$$

In other words, find x_1 that is as close as possible to x_0 but has a different label.

- How can we **solve** this? Three common options:
  - Use **gradient descent** (or, “if you have a hammer, every problem looks like a nail”)
  - Use **model-specific procedures**.
  - Use **mathematical programming**.

Finding a Counterfactual

Given original input x_0 predicted as y_0, find counterfactual input x_1 with alternative label y_0 by solving:

$$\begin{aligned} \mathrm{x}_1 \leftarrow \operatorname*{argmin}_{\mathrm{x}\in\mathbb{R}^d} &\ \|\mathrm{x}_0 - \mathrm{x}_1\| \\ \text{s.t. } &\ f(\mathrm{x}_1) = y_1 \qquad (\text{or } f(\mathrm{x}_1) \neq f(\mathrm{x}_0)) \end{aligned}$$

In other words, find x_1 that is as close as possible to x_0 but has a different label.

- How can we **solve** this? Three common options:
  - Use **gradient descent** (or, “if you have a hammer, every problem looks like a nail”)
  - Use **model-specific procedures**.
  - Use **mathematical programming**.

Finding a Counterfactual

Given original input $\mathbf{x}_0$ predicted as y_0, find counterfactual input $\mathbf{x}_1$ with alternative label y_0 by solving:

$$
\begin{aligned}
\mathbf{x}_1 \leftarrow \operatorname*{argmin}_{\mathbf{x}\in\mathbb{R}^d} & \;\|\mathbf{x}_0 - \mathbf{x}_1\| \\
\text{s.t. } & f(\mathbf{x}_1) = y_1 \qquad (\text{or } f(\mathbf{x}_1) \neq f(\mathbf{x}_0))
\end{aligned}
$$

In other words, find $\mathbf{x}_1$ that is as close as possible to $\mathbf{x}_0$ but has a different label.

- How can we **solve** this? Three common options:
  - Use **gradient descent** (or, “if you have a hammer, every problem looks like a nail”)
  - Use **model-specific procedures**.
  - Use **mathematical programming**.

Counterfactuals with Gradient Ascent

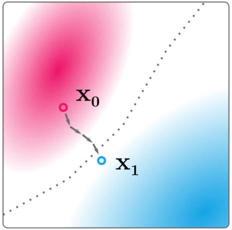
- Assume that the model outputs a score or probability for each class y_0 (**red**) and y_1 (**blue**), say:

$$f(\mathbf{x}) = (0.92, 0.08)$$

and that it is smooth in the input $\mathbf{x}$.

Algorithm

- Initialize candidate $\mathbf{x}$ to $\mathbf{x}_0$.
- Compute gradient of probability of the counterfactual class, in our case $(\nabla_{\mathbf{x}} f)_1$.
- Slightly move $\mathbf{x}$ in the direction of the gradient.
- Repeat until $\mathbf{x}$ has the desired label y_1.



Using gradient ascent to find a counterfactual example.

- **Issue**: $\mathbf{x}_1$ is necessarily the closest counterfactual to $\mathbf{x}_0$.

- **Issue**: algorightm may **fail** in some situationsm, e.g., local optima in the red region.

Counterfactuals for Decision Trees

Consider a **decision tree** f:

- Decision surface can be decomposed into **leaves** $\{\ell\}$
- Each leaf identifies a **region** ϕ_ℓ of input space that is described as the conjunction of logical conditions, for instance:

  $$\phi_\ell = (x_{\text{age}} > 21) \wedge (x_{\text{nsiblings}} \leq 2.5)$$

  The union of all leaves is $\mathbb{R}^d$
- Each leaf is associated to a label $y_\ell \in [c]$

Algorithm

Given $f(\mathbf{x}_0) = y_0$ and $y_1 \neq y_0$, finding a counterfactual example $\mathbf{x}_1$ with label y_1 amounts to:

1. Find leaf ℓ to which $\mathbf{x}_0$ belongs [easy]
2. Iterate over all other leaves $\ell' \neq \ell$ and keep those that have label $y_{\ell'} = y_1$.
3. For each such ℓ', compute $\min_{\mathbf{x}' \models \phi_{\ell'}} \|\mathbf{x}_0 - \mathbf{x}'\|_1$
4. Pick the closest such ℓ' and use the corresponding $\mathbf{x}'$ as $\mathbf{x}_1$.

■ Complexity is linear in the number of leaves, times the amount needed to solve the projection (Step 3)

Counterfactuals for Decision Trees

Consider a **decision tree** f:

- Decision surface can be decomposed into **leaves** $\{\ell\}$
- Each leaf identifies a **region** ϕ_ℓ of input space that is described as the conjunction of logical conditions, for instance:

  $$\phi_\ell = (x_{\text{age}} > 21) \wedge (x_{\text{nsiblings}} \leq 2.5)$$

  The union of all leaves is $\mathbb{R}^d$
- Each leaf is associated to a label $y_\ell \in [c]$

Algorithm

Given $f(\mathrm{x}_0) = y_0$ and $y_1 \neq y_0$, finding a counterfactual example x_1 with label y_1 amounts to:

1. Find leaf ℓ to which x_0 belongs [easy]
2. Iterate over all other leaves $\ell' \neq \ell$ and keep those that have label $y_{\ell'} = y_1$.
3. For each such ℓ', compute $\min_{\mathrm{x}' \models \phi_{\ell'}} \|\mathrm{x}_0 - \mathrm{x}'\|_1$
4. Pick the closest such ℓ' and use the corresponding x' as x_1.

■ Complexity is linear in the number of leaves, times the amount needed to solve the projection (Step 3)

Counterfactuals for Decision Trees

Consider a **decision tree** f:

- Decision surface can be decomposed into **leaves** $\{\ell\}$
- Each leaf identifies a **region** ϕ_ℓ of input space that is described as the conjunction of logical conditions, for instance:

  $$\phi_\ell = (x_{\text{age}} > 21) \wedge (x_{\text{nsiblings}} \leq 2.5)$$

  The union of all leaves is $\mathbb{R}^d$
- Each leaf is associated to a label $y_\ell \in [c]$

Algorithm

Given $f(\mathrm{x}_0) = y_0$ and $y_1 \neq y_0$, finding a counterfactual example x_1 with label y_1 amounts to:

1. Find leaf ℓ to which x_0 belongs [**easy**]
2. Iterate over all other leaves $\ell' \neq \ell$ and keep those that have label $y_{\ell'} = y_1$.
3. For each such ℓ', compute $\min_{\mathrm{x}' \models \phi_{\ell'}} \|\mathrm{x}_0 - \mathrm{x}'\|_1$
4. Pick the closest such ℓ' and use the corresponding x' as x_1.

■ Complexity is **linear** in the number of leaves, times the amount needed to solve the projection (Step 3)

Counterfactuals for Decision Trees

Consider a **decision tree** f:

- Decision surface can be decomposed into **leaves** $\{\ell\}$
- Each leaf identifies a **region** ϕ_ℓ of input space that is described as the conjunction of logical conditions, for instance:

$$\phi_\ell = (x_{\text{age}} > 21) \wedge (x_{\text{nsiblings}} \leq 2.5)$$

  The union of all leaves is $\mathbb{R}^d$
- Each leaf is associated to a label $y_\ell \in [c]$

Algorithm

Given $f(\mathbf{x}_0) = y_0$ and $y_1 \neq y_0$, finding a counterfactual example $\mathbf{x}_1$ with label y_1 amounts to:

1. Find leaf ℓ to which $\mathbf{x}_0$ belongs [easy]
2. Iterate over all other leaves $\ell' \neq \ell$ and keep those that have label $y_{\ell'} = y_1$.
3. For each such ℓ', compute $\min_{\mathbf{x}' \models \phi_{\ell'}} \|\mathbf{x}_0 - \mathbf{x}'\|_1$
4. Pick the closest such ℓ' and use the corresponding $\mathbf{x}'$ as $\mathbf{x}_1$.

■ Complexity is linear in the number of leaves, times the amount needed to solve the projection (Step 3)

Counterfactuals for Decision Trees

Consider a **decision tree** f:

- Decision surface can be decomposed into **leaves** $\{\ell\}$
- Each leaf identifies a **region** ϕ_ℓ of input space that is described as the conjunction of logical conditions, for instance:

  $$\phi_\ell = (x_{\text{age}} > 21) \land (x_{\text{nsiblings}} \leq 2.5)$$

  The union of all leaves is $\mathbb{R}^d$
- Each leaf is associated to a label $y_\ell \in [c]$

Algorithm

Given $f(\mathbf{x}_0) = y_0$ and $y_1 \neq y_0$, finding a counterfactual example $\mathbf{x}_1$ with label y_1 amounts to:

1. Find leaf ℓ to which $\mathbf{x}_0$ belongs [**easy**]
2. Iterate over all other leaves $\ell' \neq \ell$ and keep those that have label $y_{\ell'} = y_1$.
3. For each such ℓ', compute $\min_{\mathbf{x}' \models \phi_{\ell'}} \|\mathbf{x}_0 - \mathbf{x}'\|_1$
4. Pick the closest such ℓ' and use the corresponding $\mathbf{x}'$ as $\mathbf{x}_1$.

■ Complexity is **linear** in the number of leaves, times the amount needed to solve the projection (Step 3)

Counterfactuals for Decision Trees

Consider a **decision tree** f:

- Decision surface can be decomposed into **leaves** $\{\ell\}$
- Each leaf identifies a **region** ϕ_ℓ of input space that is described as the conjunction of logical conditions, for instance:

  $$\phi_\ell = (x_{\text{age}} > 21) \wedge (x_{\text{nsiblings}} \leq 2.5)$$

  The union of all leaves is $\mathbb{R}^d$
- Each leaf is associated to a label $y_\ell \in [c]$

Algorithm

Given $f(\mathbf{x}_0) = y_0$ and $y_1 \neq y_0$, finding a counterfactual example $\mathbf{x}_1$ with label y_1 amounts to:

1. Find leaf ℓ to which $\mathbf{x}_0$ belongs [**easy**]
2. Iterate over all other leaves $\ell' \neq \ell$ and keep those that have label $y_{\ell'} = y_1$.
3. For each such ℓ', compute $\min_{\mathbf{x}' \models \phi_{\ell'}} \|\mathbf{x}_0 - \mathbf{x}'\|_1$
4. Pick the closest such ℓ' and use the corresponding $\mathbf{x}'$ as $\mathbf{x}_1$.

■ Complexity is **linear** in the number of leaves, times the amount needed to solve the projection (Step 3)

Additional Properties

- **Actionability**: a counterfactual should never ask the user to change an immutable feature (e.g., ethnicity, age) but only features that the user has control over (e.g., amount of income, degree of education)
- **Causal Actionability**: features are rarely independent, e.g., in order to increase the degree of education one has to age a bit. Counterfactuals should take this into account.
- **Validity**: if x is structured – i.e., if it must obey structure constraints, for instance because of molecule (chemical validity) or a solution to a Sudoku problem (Sudoku rules) – then these constraints must be taken into consideration when computing counterfactual instances x'.
- **Believability**: It is hard to trust/believe a counterfactual if it includes a combination of features which are very different from observations the classifier has seen before. So we'd like $p^*(\mathrm{x}_1)$ to be large if possible, i.e., it should lie on the data manifold. (Otherwise we'd get an **adversarial example** instead.)

Additional Properties

- **Actionability**: a counterfactual should never ask the user to change an immutable feature (e.g., ethnicity, age) but only features that the user has control over (e.g., amount of income, degree of education)
- **Causal Actionability**: features are rarely independent, e.g., in order to increase the degree of education one has to age a bit. Counterfactuals should take this into account.
- **Validity**: if x is structured – i.e., if it must obey structure constraints, for instance because of molecule (chemical validity) or a solution to a Sudoku problem (Sudoku rules) – then these constraints must be taken into consideration when computing counterfactual instances x'.
- **Believability**: It is hard to trust/believe a counterfactual if it includes a combination of features which are very different from observations the classifier has seen before. So we'd like $p^*(\mathrm{x}_1)$ to be large if possible, i.e., it should lie on the data manifold. (Otherwise we'd get an **adversarial example** instead.)

Additional Properties

- **Actionability**: a counterfactual should never ask the user to change an immutable feature (e.g., ethnicity, age) but only features that the user has control over (e.g., amount of income, degree of education)
- **Causal Actionability**: features are rarely independent, e.g., in order to increase the degree of education one has to age a bit. Counterfactuals should take this into account.
- **Validity**: if x is structured – i.e., if it must obey structure constraints, for instance because of molecule (chemical validity) or a solution to a Sudoku problem (Sudoku rules) – then these constraints must be taken into consideration when computing counterfactual instances x'.
- **Believability**: It is hard to trust/believe a counterfactual if it includes a combination of features which are very different from observations the classifier has seen before. So we'd like $p^*(\mathrm{x}_1)$ to be large if possible, i.e., it should lie on the data manifold. (Otherwise we'd get an **adversarial example** instead.)

Additional Properties

- **Actionability**: a counterfactual should never ask the user to change an immutable feature (e.g., ethnicity, age) but only features that the user has control over (e.g., amount of income, degree of education)
- **Causal Actionability**: features are rarely independent, e.g., in order to increase the degree of education one has to age a bit. Counterfactuals should take this into account.
- **Validity**: if $\mathbf{x}$ is structured – i.e., if it must obey structure constraints, for instance because of molecule (chemical validity) or a solution to a Sudoku problem (Sudoku rules) – then these constraints must be taken into consideration when computing counterfactual instances $\mathbf{x}'$.
- **Believability**: It is hard to trust/believe a counterfactual if it includes a combination of features which are very different from observations the classifier has seen before. So we'd like $p^*(\mathbf{x}_1)$ to be large if possible, i.e., it should lie on the data manifold. (Otherwise we'd get an **adversarial example** instead.)

Take-away

- Counterfactuals are **human-friendly**: we use them all the time [Byrne, 2019]
- Counterfactuals support **actionable recourse**, i.e., stakeholders can decide what to change for the outcome to change
- Counterfactuals can be computed by solving **constrained optimization problem**
- Solving it can be **computationally challenging** for general models
- Cheap approximations based on gradient descent give **few guarantees**, make **interpretation tricky**

Take-away

- Many different types of explanations with different properties:
  - See [Guidotti et al., 2018]

- Many different implementations, for instance:
  - `captum` for Pytorch: `github.com/pytorch/captum`
  - `innvestigate` for Tensorflow: `github.com/albermax/innvestigate`
  - `DiCE` for counterfactuals: `github.com/albermax/innvestigate`
  - Can be used right away to find bugs & quirks in your models

- Still very much being worked out – we just scratched the surface

Adebayo, J., Gilmer, J., Muelly, M., Goodfellow, I., Hardt, M., and Kim, B. (2018).
Sanity checks for saliency maps.
In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pages 9525–9536.

Agarwal, N., Bullins, B., and Hazan, E. (2017).
Second-order stochastic optimization for machine learning in linear time.
The Journal of Machine Learning Research, 18(1):4148–4187.

Baehrens, D., Schroeter, T., Harmeling, S., Kawanabe, M., Hansen, K., and Müller, K.-R. (2010).
How to explain individual classification decisions.
The Journal of Machine Learning Research, 11:1803–1831.

Basu, S., Pope, P., and Feizi, S. (2020).
Influence functions in deep learning are fragile.
arXiv preprint arXiv:2006.14651.

Byrne, R. M. (2019).
Counterfactuals in explainable artificial intelligence (xai): Evidence from human reasoning.
In *IJCAI*, pages 6276–6282.

Dai, X., Keane, M. T., Shalloo, L., Ruelle, E., and Byrne, R. M. (2022).
Counterfactual explanations for prediction and diagnosis in xai.
In *Proceedings of the 2022 AAAI/ACM Conference on AI, Ethics, and Society*, pages 215–226.

Garreau, D. and Luxburg, U. (2020).
Explaining the explainer: A first theoretical analysis of lime.
In *International Conference on Artificial Intelligence and Statistics*, pages 1287–1296. PMLR.

Guidotti, R., Monreale, A., Giannotti, F., Pedreschi, D., Ruggieri, S., and Turini, F. (2019).
Factual and counterfactual explanations for black box decision making.
IEEE Intelligent Systems, 34(6):14–23.

Guidotti, R., Monreale, A., Ruggieri, S., Turini, F., Giannotti, F., and Pedreschi, D. (2018).
A survey of methods for explaining black box models.
ACM computing surveys (CSUR), 51(5):1–42.

Koh, P. W. and Liang, P. (2017).
Understanding black-box predictions via influence functions.
In *Proceedings of the 34th International Conference on Machine Learning*, pages 1885–1894.

Lapuschkin, S., Wäldchen, S., Binder, A., Montavon, G., Samek, W., and Müller, K.-R. (2019).
Unmasking clever hans predictors and assessing what machines really learn.
Nature communications, 10(1):1–8.

Lin, Jung, J., Goel, S., and Skeem, J. (2020).
The limits of human predictions of recidivism.
Science advances, 6(7):eaaz0652.

Lipinski, P., Brzychczy, E., and Zimroz, R. (2020).
Decision tree-based classification for planetary gearboxes' condition monitoring with the use of vibration data in multidimensional symptom space.
Sensors, 20(21):5979.

Lipton, Z. C. (2018).
The mythos of model interpretability: In machine learning, the concept of interpretability is both important and slippery.
Queue, 16(3):31–57.

Lundberg, S. M. and Lee, S.-I. (2017).
A unified approach to interpreting model predictions.
In *Proceedings of the 31st international conference on neural information processing systems*, pages 4768–4777.

Pearl, J. (2009).
Causality.
Cambridge university press.

Pearl, J. and Mackenzie, D. (2018).
The book of why: the new science of cause and effect.
Basic books.

Ribeiro, M. T., Singh, S., and Guestrin, C. (2016).

“Why should I trust you?” Explaining the predictions of any classifier.
In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1135–1144.

Rudin, C. (2019).
Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead.
Nature Machine Intelligence, 1(5):206–215.

Simonyan, K., Vedaldi, A., and Zisserman, A. (2013).
Deep inside convolutional networks: Visualising image classification models and saliency maps.
arXiv preprint arXiv:1312.6034.

Slack, D., Hilgard, S., Jia, E., Singh, S., and Lakkaraju, H. (2020).
Fooling lime and shap: Adversarial attacks on post hoc explanation methods.
In *Proceedings of the AAAI/ACM Conference on AI, Ethics, and Society*, pages 180–186.

Štrumbelj, E. and Kononenko, I. (2014).
Explaining prediction models and individual predictions with feature contributions.
Knowledge and information systems, 41(3):647–665.

Sundararajan, M., Taly, A., and Yan, Q. (2017).
Axiomatic attribution for deep networks.
In *International Conference on Machine Learning*, pages 3319–3328. PMLR.

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

Ustun, B. and Rudin, C. (2016).
Supersparse linear integer models for optimized medical scoring systems.
Machine Learning, 102(3):349–391.

Van den Broeck, G., Lykov, A., Schleich, M., and Suciu, D. (2021).
On the tractability of shap explanations.
In *Proceedings of AAAI*.