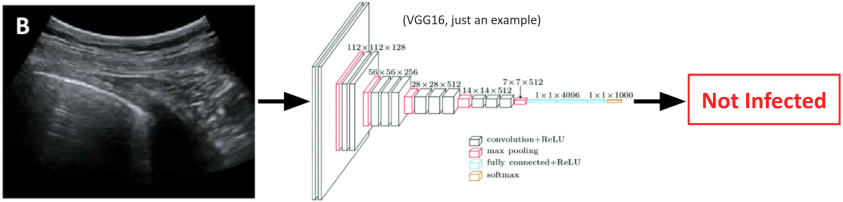
Explainable Machine Learning

Stefano Teso
Advanced Machine Learning Course

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.

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)
  - ...

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

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)
  - ...

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

- 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`

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)
  - ...

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

- 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`

- **Counter-argument**: Humans are not necessarily better and/or fairer than machines [Lin et al., 2020]

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)
  - ...

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

- 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`

- **Counter-argument**: 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?

We will mostly focus on **classification**.

Notation:

- Instances $\mathbf{x} \in \mathbb{R}^d$

We will mostly focus on **classification**.

Notation:

- Instances $\mathbf{x} \in \mathbb{R}^d$
- Labels $y \in \{1, \ldots, c\}$, often $c = 2$

We will mostly focus on **classification**.

Notation:

- Instances $\mathbf{x} \in \mathbb{R}^d$
- Labels $y \in \{1, \ldots, c\}$, often $c = 2$
- Family of classifiers $\mathcal{F}$, for instance **neural networks**, **random forests**, ...
- A classifier is a map $f : \mathcal{X} \to \{1, \ldots, c\}$

We will mostly focus on **classification**.

Notation:

- Instances $\mathbf{x} \in \mathbb{R}^d$
- Labels $y \in \{1, \ldots, c\}$, often $c = 2$
- Family of classifiers $\mathcal{F}$, for instance **neural networks**, **random forests**, ...
- A classifier is a map $f : \mathcal{X} \to \{1, \ldots, c\}$
- We often consider probabilistic classifiers defined by a conditional distribution $P(Y \mid \mathbf{X})$, in which case:

$$f(\mathbf{x}) := \operatorname*{argmax}_{y \in [c]} \; P(Y \mid \mathbf{X})$$

  Sometimes we simply use the distribution $P(Y \mid \mathbf{X})$ as a "soft prediction"

Classification

Given a family of classifiers (hypotheses) $\mathcal{F}$ and a data set $S = \{(\mathbf{x}_i, y_i) \ : \ i = 1, \ldots, m\}$ sampled i.i.d. from a ground-truth distribution $D(\mathbf{X}, Y)$, **find** a classifier $f \in \mathcal{F}$ that achieves low **true risk**.

Classification

Given a family of classifiers (hypotheses) $\mathcal{F}$ and a data set $S = \{(\mathbf{x}_i, y_i) \ : \ i = 1, \ldots, m\}$ sampled i.i.d. from a ground-truth distribution $D(\mathbf{X}, Y)$, **find** a classifier $f \in \mathcal{F}$ that achieves low **true risk**.

- Let $\ell(f, (\mathbf{x}, y))$ be a **loss** of interest: e.g., the 0–1 loss $\ell(f, (\mathbf{x}, y)) = \mathbb{1}\{f(\mathbf{x}) \neq y\}$ or the cross-entropy loss:

$$\ell(f, (\mathbf{x}, y)) = -\sum_{j \in [c]} \mathbb{1}\{j = y\} \, P(Y = j \mid \mathbf{x})$$

Classification

Given a family of classifiers (hypotheses) $\mathcal{F}$ and a data set $S = \{(\mathbf{x}_i, y_i) \ : \ i = 1, \ldots, m\}$ sampled i.i.d. from a ground-truth distribution $D(\mathbf{X}, Y)$, **find** a classifier $f \in \mathcal{F}$ that achieves low **true risk**.

- Let $\ell(f, (\mathbf{x}, y))$ be a **loss** of interest: e.g., the 0–1 loss $\ell(f, (\mathbf{x}, y)) = \mathbb{1}\{f(\mathbf{x}) \neq y\}$ or the cross-entropy loss:

$$\ell(f, (\mathbf{x}, y)) = -\sum_{j \in [c]} \mathbb{1}\{j = y\} \, P(Y = j \mid \mathbf{x})$$

- The **true risk** is the average loss w.r.t. the ground-truth distribution D:

$$L_D(f) := \mathbb{E}_{(\mathbf{x},y)\sim D}[\ell(f, (\mathbf{x}, y))] = \int_{\mathbb{R}^d} \sum_y \ell(f, (\mathbf{x}, y)) D(\mathbf{x}, y) d\mathbf{x}$$

Classification

Given a family of classifiers (hypotheses) $\mathcal{F}$ and a data set $S = \{(\mathbf{x}_i, y_i) \ : \ i = 1, \ldots, m\}$ sampled i.i.d. from a ground-truth distribution $D(\mathbf{X}, Y)$, **find** a classifier $f \in \mathcal{F}$ that achieves low **true risk**.

- Let $\ell(f, (\mathbf{x}, y))$ be a **loss** of interest: e.g., the 0–1 loss $\ell(f, (\mathbf{x}, y)) = \mathbb{1}\{f(\mathbf{x}) \neq y\}$ or the cross-entropy loss:

$$\ell(f, (\mathbf{x}, y)) = -\sum_{j \in [c]} \mathbb{1}\{j = y\} \, P(Y = j \mid \mathbf{x})$$

- The **true risk** is the average loss w.r.t. the ground-truth distribution D:

$$L_D(f) := \mathbb{E}_{(\mathbf{x},y)\sim D}[\ell(f, (\mathbf{x}, y))] = \int_{\mathbb{R}^d} \sum_y \ell(f, (\mathbf{x}, y)) D(\mathbf{x}, y) d\mathbf{x}$$

- This cannot be computed because D is unknown, so minimize **empirical risk** on the training set S:

$$\widehat{L}_S(f) := \frac{1}{|S|} \sum_{(\mathbf{x},y)\in S} \ell(f(\mathbf{x}), y)$$

obtaining $\hat{f} := \operatorname{argmin}_{f \in \mathcal{F}} \ \widehat{L}_S(f)$.

Classification

Given a family of classifiers (hypotheses) $\mathcal{F}$ and a data set $S = \{(\mathbf{x}_i, y_i) \ : \ i = 1, \ldots, m\}$ sampled i.i.d. from a ground-truth distribution $D(\mathbf{X}, Y)$, **find** a classifier $f \in \mathcal{F}$ that achieves low **true risk**.

- Let $\ell(f, (\mathbf{x}, y))$ be a **loss** of interest: e.g., the 0–1 loss $\ell(f, (\mathbf{x}, y)) = \mathbb{1}\{f(\mathbf{x}) \neq y\}$ or the cross-entropy loss:

$$\ell(f, (\mathbf{x}, y)) = -\sum_{j \in [c]} \mathbb{1}\{j = y\} \, P(Y = j \mid \mathbf{x})$$

- The **true risk** is the average loss w.r.t. the ground-truth distribution D:

$$L_D(f) := \mathbb{E}_{(\mathbf{x},y)\sim D}[\ell(f, (\mathbf{x}, y))] = \int_{\mathbb{R}^d} \sum_y \ell(f, (\mathbf{x}, y)) D(\mathbf{x}, y) d\mathbf{x}$$

- This cannot be computed because D is unknown, so minimize **empirical risk** on the training set S:

$$\widehat{L}_S(f) := \frac{1}{|S|} \sum_{(\mathbf{x},y)\in S} \ell(f(\mathbf{x}), y)$$

  obtaining $\hat{f} := \operatorname{argmin}_{f \in \mathcal{F}} \ \widehat{L}_S(f)$.

- Well-known conditions under which $|L_D(\hat{f}) - \widehat{L}_S(\hat{f})|$ decreases as the size of S increases (VC dimension, Rademacher complexity, ...)

Standard learning **pipeline**:

- Learn $\hat{f}$ on training set S.
- Evaluate $\hat{f}$ on a validation set T.

Is this enough?

Standard learning **pipeline**:

- Learn $\hat{f}$ on training set S.
- Evaluate $\hat{f}$ on a validation set T.

Is this enough? **Not always**:

> *"The demand for interpretability arises when there is a mismatch between the formal objectives of supervised learning (test set predictive performance) and the real world costs in a deployment setting." [Lipton, 2018]*

The training & validation sets are unlikely to cover all of the high-risk cases

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.

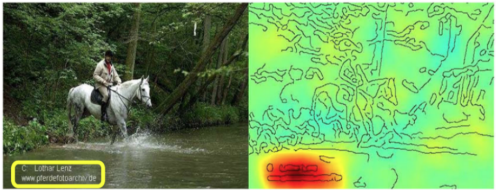
Confounders

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

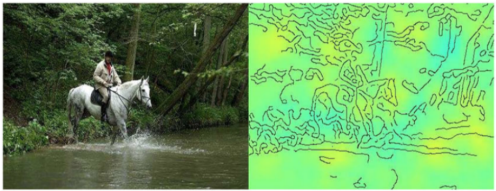
Horse-picture from Pascal VOC data set



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."

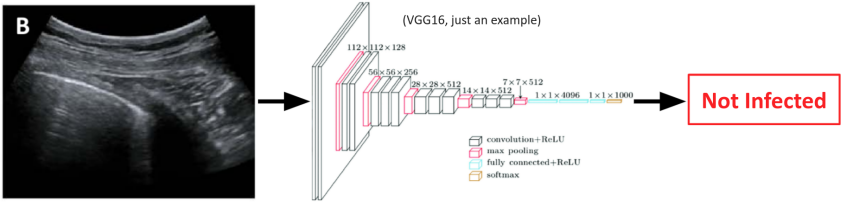
"After a formal investigation in 1907, psychologist Oskar Pfungst demonstrated that the horse was not actually performing these mental tasks, but was watching the reactions of his trainer."

Hans managed to picked up on **confounders**

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

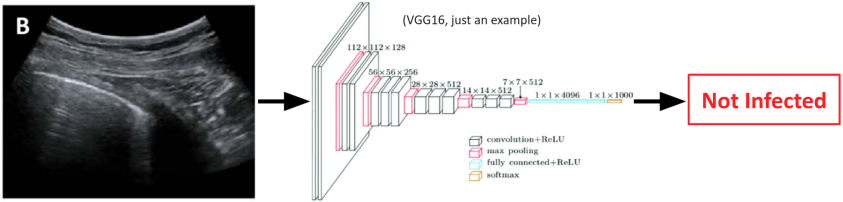
Credit: `en.wikipedia.org/wiki/Clever_Hans`

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?

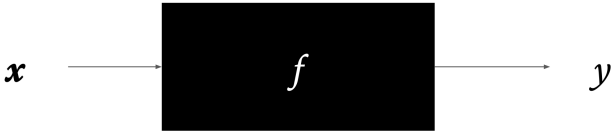
You need to be checked for COVID-19. The doctor takes a scan of your lungs and uses a state-of-the-art deep neural network to automatically compute a diagnosis. The model thinks that you are not infected.



Question: Would you trust the model's prediction?

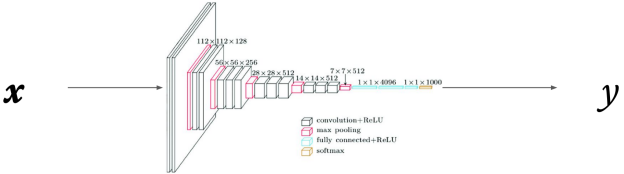
Presumably, you'll want to know **whether the model exhibits C-H behavior first** ;-)

A **black-box** classifier $f : \mathbb{R}^d \to [c]$ should like this:



Examples: neural networks, kernel 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**, but 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!

A **linear model** has the form:

$$f(\mathbf{x}) = \mathrm{sign}\big(\underbrace{\langle \mathbf{w}, \mathbf{x}\rangle + w_0}_{\text{“score” of } \mathbf{x}}\big), \qquad \langle \mathbf{w}, \mathbf{x}\rangle := \textstyle\sum_{i\in[d]} w_i x_i$$

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:[1]

- $w_i > 0 \implies x_i$ 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

[1] 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: Papayas

Does a **papaya** $\mathbf{x}$ taste good?

Consider a linear classifier:

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

Figure 1: A bunch of papaya fruits.

It is easy to read off what attributes are "for" and "against" $\mathbf{x}$ being tasty **for the model** – specifically because the model encodes independence assumptions, e.g., that the shape of $\mathbf{x}$ is unrelated to its color.[2]

[2] 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!

Example: Newsgroup Posts

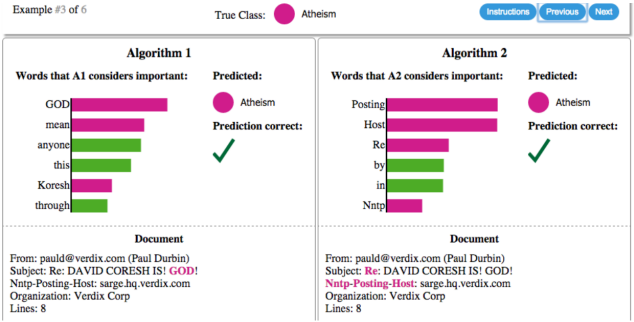


Figure 2: Explaining individual predictions of competing classifiers trying to determine if a document is about "Christianity" or "Atheism". The bar chart represents the importance given to the **most relevant words**, also highlighted in the text. Color indicates which class the word contributes to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

Caveats

- If not sparse, it may be difficult to simulate the model's reasoning in your head.

Caveats

- If not sparse, it may be difficult to simulate the model's reasoning in your head.

- The learned weights depend on the available attributes.

Example: the importance of the attribute $\mathbb{1}\{\mathbf{x} \text{ skin is yellow}\}$ 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 it does not, then color may be the only important factor.

In other words, it's best not to make *absolute* judgments based on an arbitrary selection of attributes.

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

Left: a DT for the Titanic survivors dataset. The variables include **age**, **sex**, **passenger class**, and **# of siblings** onboard.

- 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.
- The decision in each node only involves one interpretable variable (e.g., age) and is therefore easy to understand.

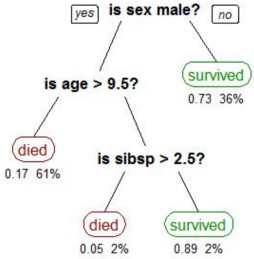


Figure 3: A shallow decision tree.

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.

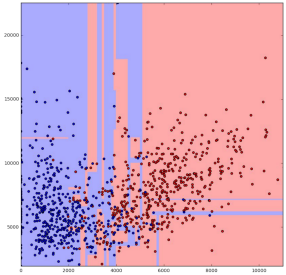
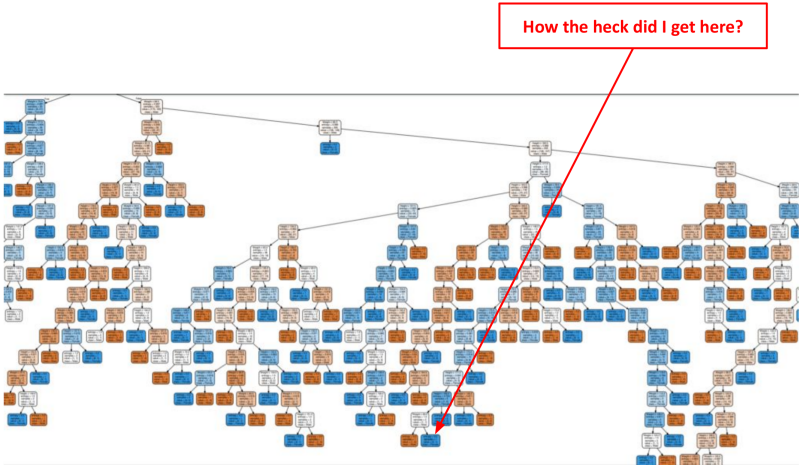


Figure 4: A shallow decision tree.

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

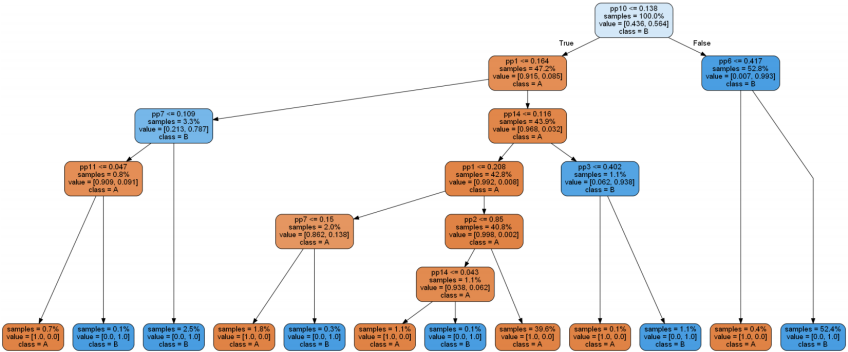
Caveats

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



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.

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.

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 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, while black-box models do (e.g., image classification)

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

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"*

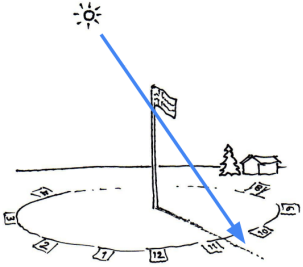


Figure 5: a flagpole and the Sun.

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!

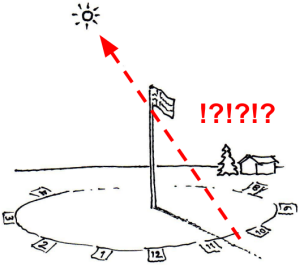


Figure 6: a flagpole and the Sun.

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.

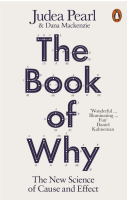
■ This is what people do in science and debugging: knocking out genes in mices or fixing the value of some variables in programs to compare the original and altered systems. Interventions are key to understand **how a mechanism works**.

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

---

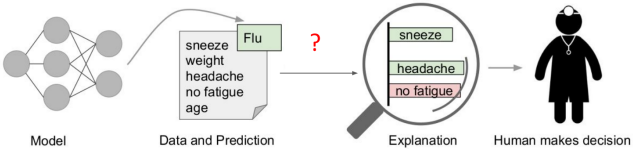
Note: causality is a fascinating topic. If you are interested, a good non-technical introduction is given by "The Book of Why" [Pearl and Mackenzie, 2018].



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?

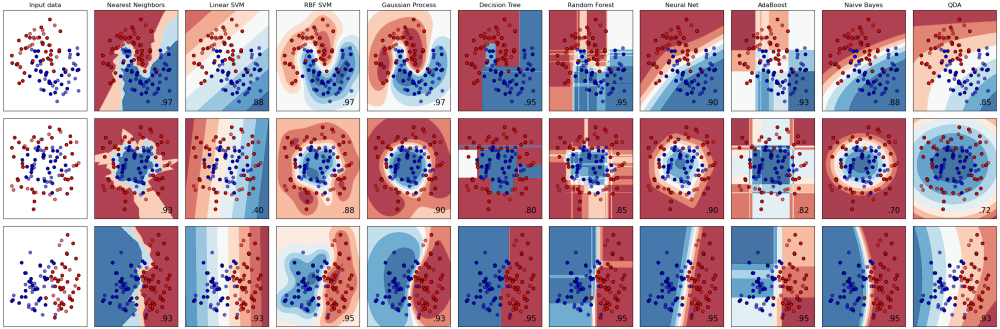
Recall that it is **easy** to answer this question for white-box models.

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

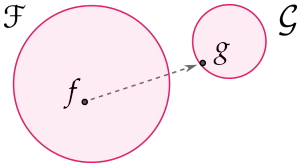
Given a classifier $f \in \mathcal{F}$ (e.g., a neural net), find a white-box classifier $g \in \mathcal{G}$ (e.g., a shallow decision tree) that approximates its predictions.

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

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

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.

[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 $\{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$

[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 $\{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$
2. Label the samples using f, obtaining $y_i := f(\mathbf{x}_i)$ for $i \in [m]$

[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 $\{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$
2. Label the samples using f, obtaining $y_i := f(\mathbf{x}_i)$ for $i \in [m]$
3. Fit $g \in \mathcal{G}$ on the the **synthetic data set** $S = \{(\mathbf{x}_i, y_i) : i \in [m]\}$

[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 $\{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$
2. Label the samples using f, obtaining $y_i := f(\mathbf{x}_i)$ for $i \in [m]$
3. Fit $g \in \mathcal{G}$ on the the **synthetic data set** $S = \{(\mathbf{x}_i, y_i) : i \in [m]\}$

In other words, model translation can be implemented as **learning**.

[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 $\{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$
2. Label the samples using f, obtaining $y_i := f(\mathbf{x}_i)$ for $i \in [m]$
3. Fit $g \in \mathcal{G}$ on the the **synthetic data set** $S = \{(\mathbf{x}_i, y_i) : i \in [m]\}$

In other words, model translation can be implemented as **learning**.

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.

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.

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.

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!

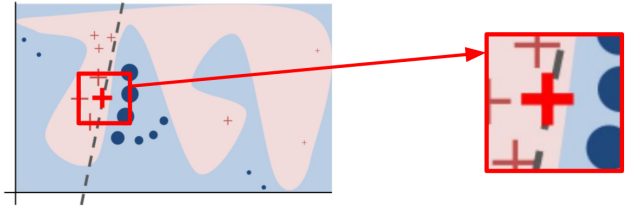
- **How large should the synthetic data set S be?**
  - Start with a small S
  - Grow S and retrain until $d(f, g) \leq \tau$, with τ controllable threshold.

- **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!

- **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.

Local Interpretable Model-agnostic Explanations (LIME)

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



- Those parts of the model that do not contribute to the decision surface around $f(\mathbf{x}_0)$ are **irrelevant** and do not need to appear in 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 $\mathbf{x}_0$, find a white-box classifier $g_0 \in \mathcal{G}$ that approximates the predictions of f **in the neighborhood of** $\mathbf{x}_0$.

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**]

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**]

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**:

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

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.

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**:

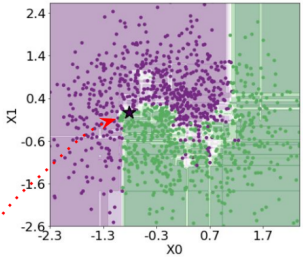
$$g_0 := \operatorname*{argmin}_{g \in \mathcal{G}} \frac{1}{m} \sum_{i \in [m]} k(\mathbf{x}_0, \mathbf{x}_i) L(g_0(\mathbf{x}_i), y_i)$$

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

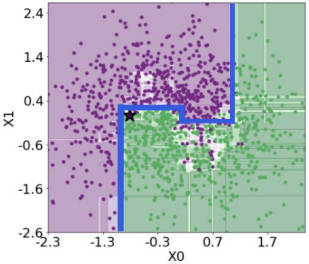
$$k(\mathbf{x}_0, \mathbf{x}_i) = \exp(-\gamma \cdot \|\mathbf{x}_0 - \mathbf{x}_i\|^2)$$

  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?

- 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.

- 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:

- 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 $\mathbf{x}_0$.

- 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.

- 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**.

■ 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.

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:

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

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.

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.

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:

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

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:

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

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

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**:

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

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

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

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

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

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

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

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

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

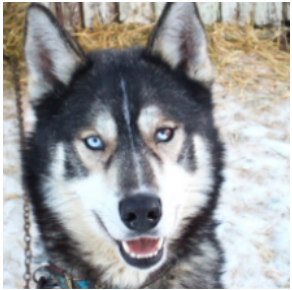
It turns out that solving this (non-combinatorial) surrogate provably solves the original problem (under assumptions).

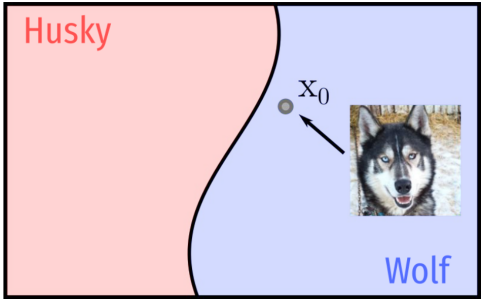
Illustration

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

Illustration

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

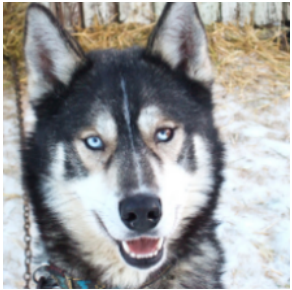


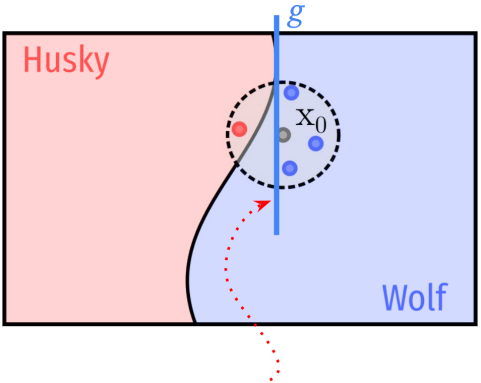


How does LIME construct an explanation for this decision?

Illustration

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

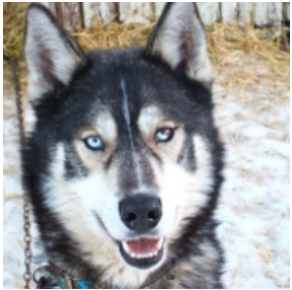


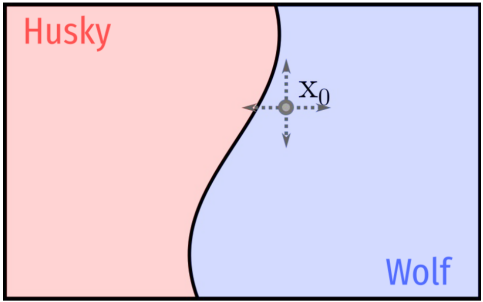


LIME samples points in the neighborhood of $\mathbf{x}_0$ and fits a **sparse linear classifier** g_0 on them

Illustration

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





Roughly equivalent to **randomly perturbing** (*aka* "wiggling") $\mathbf{x}_0$, 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?

- 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**!

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

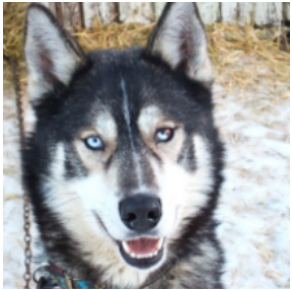
- Black-box models often rely on complex features of the inputs $\mathbf{x} = (x_1, \ldots, x_n)$:
  - **Text**: tagging documents by looking for **sequences** of words
  - **Images**: classifying pictures by leveraging **high-order correlations** between pixels

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

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**]

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**]

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:

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

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** $\psi(\mathbf{x})$ $\rightarrow$ *its explanations will also be given in terms of the interpretable concepts*

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

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

- If g_0 is a sparse linear model:

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

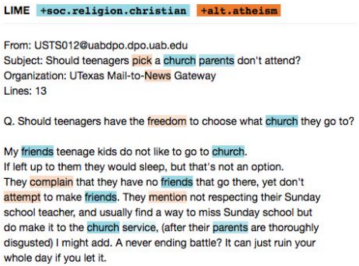
  - $w_i > 0 \implies \psi_i(\mathbf{x})$ votes for" positive class
  - $w_i < 0 \implies \psi_i(\mathbf{x})$ "votes against" positive class
  - $w_i \approx 0 \implies \psi(\mathbf{x})_i$ is irrelevant

Back to papayas

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

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

Examples



Left: LIME explains document classification by highlighting relevant words.

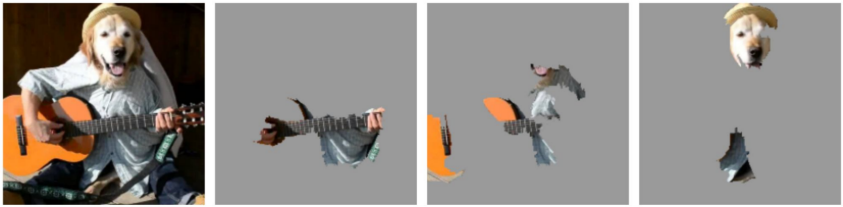
(a) Husky classified as wolf (b) Explanation

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.

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]

Shapley values

There are d **players**. The function $v(S)$ maps subsets of players $S \subseteq [d]$ to **value** $v(S) \in \mathbb{R}$, with $v(\varnothing) = 0$.

Question: *How much does ith player contribute to $v([d])$?*

This depends on the **order** in which players are added to $[d]$.

Shapley values

There are d **players**. The function $v(S)$ maps subsets of players $S \subseteq [d]$ to **value** $v(S) \in \mathbb{R}$, with $v(\varnothing) = 0$.

Question: *How much does ith player contribute to $v([d])$?*

This depends on the **order** in which players are added to $[d]$.

Fix $S \subseteq [d]$. The **marginal contribution** of i w.r.t. S is the value generated by including i in S:

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

Shapley values

There are d **players**. The function $v(S)$ maps subsets of players $S \subseteq [d]$ to **value** $v(S) \in \mathbb{R}$, with $v(\varnothing) = 0$.

Question: *How much does ith player contribute to $v([d])$?*

This depends on the **order** in which players are added to $[d]$.

Fix $S \subseteq [d]$. The **marginal contribution** of i w.r.t. S is the value generated by including i in S:

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

The **Shapley value** of i is the average marginal contribution w.r.t **every possible order**:

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

where π iterates over all **permutations** of $[d]$ and $S_{i,\pi} := \{j \ : \ \pi(j) < \pi(i)\}$ are the **players before** i **in** π.
Viewed as **the "influence" of ith player on the output of $v([d])$**.

Shapley values

There are d **players**. The function $v(S)$ maps subsets of players $S \subseteq [d]$ to **value** $v(S) \in \mathbb{R}$, with $v(\varnothing) = 0$.

Question: *How much does ith player contribute to $v([d])$?*

This depends on the **order** in which players are added to $[d]$.

Fix $S \subseteq [d]$. The **marginal contribution** of i w.r.t. S is the value generated by including i in S:

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

The **Shapley value** of i is the average marginal contribution w.r.t **every possible order**:

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

where π iterates over all **permutations** of $[d]$ and $S_{i,\pi} := \{j \ : \ \pi(j) < \pi(i)\}$ are the **players before** i **in** π.
Viewed as **the "influence" of ith player on the output of $v([d])$**.

Since the order of the elements in S does not matter for computing $\Delta(i, S)$, can rewrite:

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

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

Symmetry For any two players i, j, if $\Delta(i, S) = \Delta(j, S)$ for any $S \subseteq [d]$, 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 these properties make intuitive sense.

Idea: use Shapley values to estimate importance of ith input on the score of class y

[4] [Štrumbelj and Kononenko, 2014, Lundberg and Lee, 2017]

Idea: use Shapley values to estimate importance of ith input on the score of class y

Fix a predictor f and a decision $(\mathbf{x}, y)$. Let $\mathrm{score}(\mathbf{x})$ be the score of class y, e.g., the output of the top layer of a network or the log-likelihood.

SHAP values take:[4]

$$\begin{aligned} v(S) &= \mathbb{E}_{\mathbf{X}_{\bar{S}}}[\mathrm{score}(\mathbf{x}) \mid \mathbf{X}_S = \mathbf{x}_S] \\ &= \int_{\mathbb{R}^{|\bar{S}|}} \mathrm{score}(\mathbf{x}_S, \mathbf{x}_{\bar{S}}) d\mathbf{x}_{\bar{S}} \end{aligned}$$

where $\bar{S} = [d] \setminus S$, $\mathbf{X}_S = \{X_i : i \in S\}$, and similarly for $\mathbf{x}_S$.

This is the **expected score of class y when only features in S are known** (conditioned on $\mathbf{X}_S = \mathbf{x}_S$)

[4] [Štrumbelj and Kononenko, 2014, Lundberg and Lee, 2017]

The contribution of the ith feature is:

$$
\begin{aligned}
\phi(i) &= \sum_{S \subseteq [d]} \frac{|S|!(d-|S|-1)!}{d!} \Delta(i, S) \\
&= \sum_{S \subseteq [d]} \frac{|S|!(d-|S|-1)!}{d!} \Big(v(S \cup \{i\}) - v(S) \Big) \\
&= \sum_{S \subseteq [d]} \frac{|S|!(d-|S|-1)!}{d!} \Big(\mathbb{E}_{\mathbf{X}_{\overline{S \cup \{i\}}}}[\mathrm{score}(\mathbf{x}) \mid \mathbf{X}_{S \cup \{i\}} = \mathbf{x}_{S \cup \{i\}}] - \mathbb{E}_{\mathbf{X}_{\bar{S}}}[\mathrm{score}(\mathbf{x}) \mid \mathbf{X}_S = \mathbf{x}_S] \Big)
\end{aligned}
$$

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

The contribution of the ith feature is:

$$\begin{aligned}
\phi(i) &= \sum_{S \subseteq [d]} \frac{|S|!(d-|S|-1)!}{d!} \Delta(i, S) \\
&= \sum_{S \subseteq [d]} \frac{|S|!(d-|S|-1)!}{d!} \Big(v(S \cup \{i\}) - v(S) \Big) \\
&= \sum_{S \subseteq [d]} \frac{|S|!(d-|S|-1)!}{d!} \Big(\mathbb{E}_{\mathbf{X}_{\overline{S \cup \{i\}}}}[\mathrm{score}(\mathbf{x}) \mid \mathbf{X}_{S \cup \{i\}} = \mathbf{x}_{S \cup \{i\}}] - \mathbb{E}_{\mathbf{X}_{\bar{S}}}[\mathrm{score}(\mathbf{x}) \mid \mathbf{X}_S = \mathbf{x}_S] \Big)
\end{aligned}$$

Computing SHAP values is **highly non-trivial**:[5]

- The sum runs over 2^d subsets of variables.
- For each subset, must solve an expectation.
- Each expectation marginalizes over the model outputs, the resulting integral is often intractable.

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

Approximating SHAP

- Assume independence between features:

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

This is quite a brutal approximation in practice, but it makes the expectation independent of $\mathbf{x}_S$, i.e., it can be **cached**

- Assume that score is linear (or approximate it as such):

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

This gives an **enormous** speed-up, because we can compute the score of the average element $\mathbb{E}[\mathbf{x}]$ and we are done.

- 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

- 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)

- 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?*

Idea: typically the architecture **can be accessed**![6] Not *literally* a black-box.

[6] 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.

Idea: typically the architecture **can be accessed**![6] Not *literally* a black-box.

For instance, a neural net looks like this:

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

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.

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

and the dense layer is a linear transformation of embeddings $\phi : \mathbb{R}^d \to \mathbb{R}^q$, that is:

$$\mathbf{s}(\mathbf{x}; \theta) = W \phi(\mathbf{x})$$

[6] 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.

Idea: typically the architecture **can be accessed!**[6] Not *literally* a black-box.

For instance, a neural net looks like this:

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

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.

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

and the dense layer is a linear transformation of embeddings $\phi : \mathbb{R}^d \to \mathbb{R}^q$, that is:

$$\mathbf{s}(\mathbf{x}; \theta) = W \phi(\mathbf{x})$$

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

[6] 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 $\approx$ 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**

Gradients $\approx$ 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**

- 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 $f(\mathbf{x})$

Gradients $\approx$ Wiggling

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

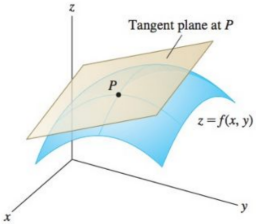
$$f'(x_0) = \left(\frac{d}{dx} f(x)\right)\Big|_{x=x_0} := \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

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

- 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 $f(\mathbf{x})$



- So, $\|\nabla_{\mathbf{x}} f(\mathbf{x}_0)\|$ measures the **sensitivity** of the output of f if we "wiggle" $\mathbf{x}_0$ around

Input Gradients

- Recall that a neural nets is:

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

The conditional distribution $p_\theta(y \mid \mathbf{x})$ is **differentiable** (almost everywhere).

Input Gradients

- Recall that a neural nets is:

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

The conditional distribution $p_\theta(y \mid \mathbf{x})$ is **differentiable** (almost everywhere).

- **Idea**: compute 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.

Input Gradients

- Recall that a neural nets is:

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

The conditional distribution $p_\theta(y \mid \mathbf{x})$ is **differentiable** (almost everywhere).

- **Idea**: compute 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]

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)

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.

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 $\mathbf{w} = (w_1, \ldots, w_d)$.

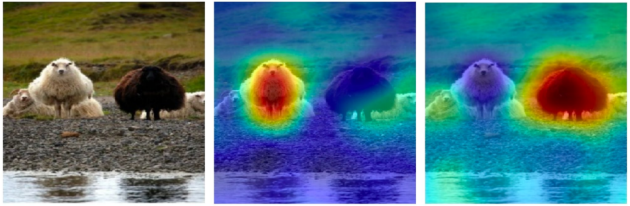
Input Gradients:

- 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 $\rightarrow$ saliency map.

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

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

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 $\mathbf{x}_0$**.

Aside: Feature interactions

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

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

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 $\mathbf{x}_0$**.

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

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

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 $x_i x_j$.

Aside: Feature interactions

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

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

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 $\mathbf{x}_0$**.

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

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

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.

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.

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.

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.

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.

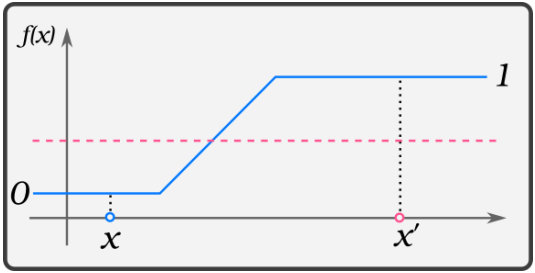
Consider a function [Sundararajan et al., 2017]:

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

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 $\mathbf{x}$, consider a **baseline** $\mathbf{x}'$ and how the gradients **change** across the two.

Idea: instead of looking at the gradient only at $\mathbf{x}$, consider a **baseline** $\mathbf{x}'$ 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 $\mathbf{x}'$ to the target point $\mathbf{x}$

Idea: instead of looking at the gradient only at $\mathbf{x}$, consider a **baseline** $\mathbf{x}'$ 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 $\mathbf{x}'$ to the target point $\mathbf{x}$

The baseline $\mathbf{x}'$ is simply:

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

Integrated gradients capture features that **account fo the change in output** between the baseline $\mathbf{x}'$ and the target point $\mathbf{x}$. 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 $\mathbf{x}$ and the baseline $\mathbf{x}'$.

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

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

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.

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.

- 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.

- Path integrals are the only methods that satisfy Completeness (and thus Sensitivity), Implementation Invariance, Dummy, and Linearity:

$$\mathrm{pathint}(\mathbf{x}, \gamma) = \int_0^1 \frac{\partial p(\gamma(\alpha))}{\partial \gamma_i(\alpha)} \frac{\partial \gamma_i(\alpha)}{\partial \alpha} d\alpha, \qquad \gamma(0) = \mathbf{x}', \gamma(1) = \mathbf{x}$$

Integrated gradients are simply path methods along a straight line γ between $\mathbf{x}'$ and $\mathbf{x}$.

What's so unique about integrated gradients?

What's so unique about integrated gradients?

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.

What's so unique about integrated gradients?

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.

- Integrated gradients are the **only** path integral that is symmetry preserving.

- 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.

- 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.

- 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.

Illustration

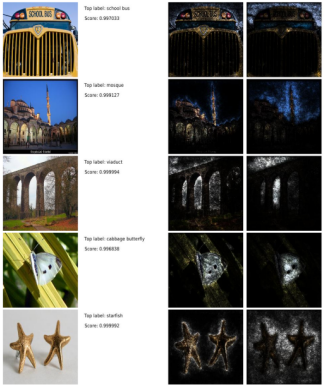


Figure 2. **Comparing integrated gradients with gradients at the image.** Left-to-right: original input image, label and softmax score for the highest scoring class, visualization of integrated gradients, visualization of gradients*image. Notice that the visualizations obtained from integrated gradients are better at reflecting distinctive features of the image.

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

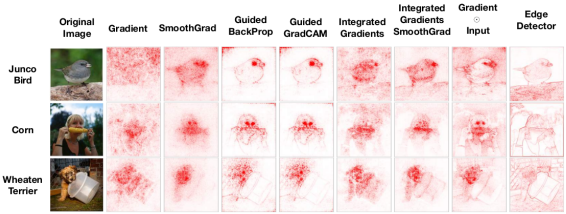


Figure 1: **Saliency maps for some common methods compared to an edge detector.** Saliency masks for 3 inputs for an Inception v3 model trained on ImageNet. We see that an edge detector produces outputs that are strikingly similar to the outputs of some saliency methods. In fact, edge detectors can also 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?)*

- 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: $\frac{1}{n} \sum_{k \in [n]} \frac{\partial}{\partial x_i} p_\theta(\mathbf{x} + \mathbf{u}_k)$
- 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.

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

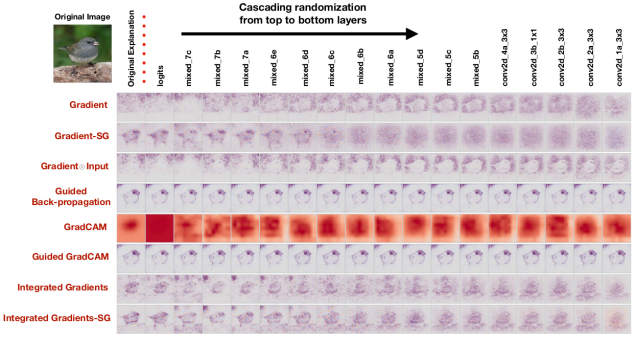


Figure 2: **Cascading randomization on Inception v3 (ImageNet).** Figure shows the original explanations (first column) for the Junco bird. Progression from left to right indicates complete randomization of network weights (and other trainable variables) up to that 'block' inclusive. We show images for 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 $p_\theta(\mathbf{x})$ to perturbations. Are they **related** somehow?

[7] 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**[7]

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

Aside: Gradients vs LIME

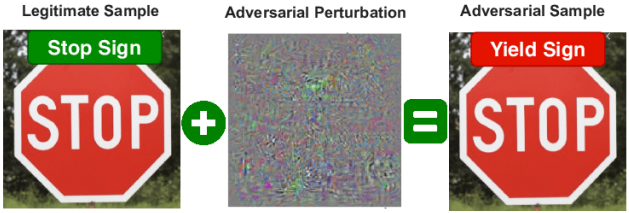
Both input gradients and LIME estimate the sensibility of the output $p_\theta(\mathbf{x})$ to perturbations. Are they **related** somehow?

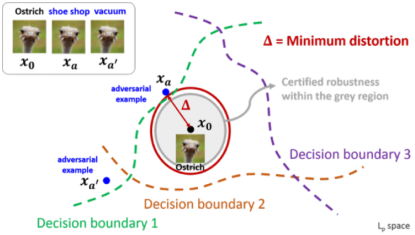
- **Yes!** Intuitively, if the kernel $k(\mathbf{x}_0, \mathbf{x}_i)$ is "pointy" enough, then LIME essentially becomes a **0-th order approximation of the input gradient**[7]

- 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 $\mathbf{x}'$.

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

Aside: Adversarial Attacks





The adversarial image $\mathbf{x}_{\text{adv}}$ is obtained by **following the gradient**:

$$\underset{\mathbf{x}_{\text{adv}}}{\text{argmin}} \; -|p_\theta(\mathbf{x}_{\text{adv}}) - p_\theta(\mathbf{x})| + \lambda \cdot \|\mathbf{x}_{\text{adv}} - \mathbf{x}\|$$

Intuition: keep $\mathbf{x}_{\text{adv}}$ close to $\mathbf{x}$, so that the difference is not perceptible to a human eye, while changing the output probability as much as possible.

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: $\mathrm{attr}(\mathbf{x}_{\text{adv}}) \approx \mathbf{a}_{\text{adv}}$

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 $\mathbf{x}$.

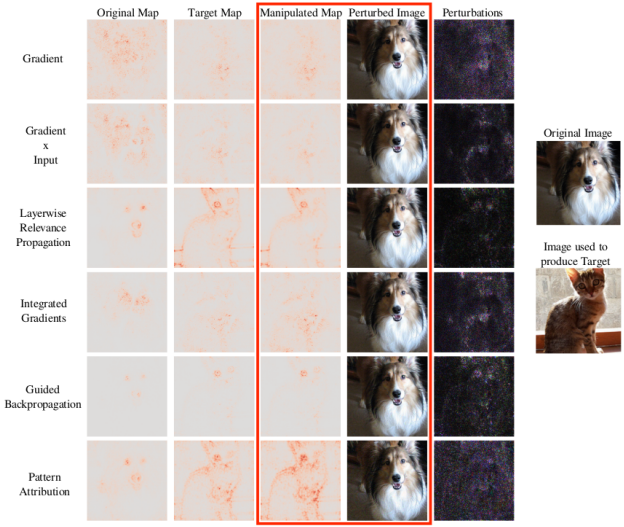


Figure 2: The explanation map of the cat is used as the target and the image of the dog is perturbed. The red box contains the manipulated images and the corresponding explanations. The first column corresponds to the original explanations of the unperturbed dog image. The target map, shown in the second 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, ...) $\rightarrow$ 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.

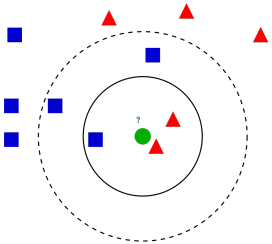
- Example attributions tell you what training examples are (indirectly) responsible for a particular prediction.

How can we explain where the model came from?

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

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 $y_0 = f(\mathbf{x})$.

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

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.

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.

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 $\varphi : \mathbb{R}^d \to \mathbb{R}^k$:

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

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:

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

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)

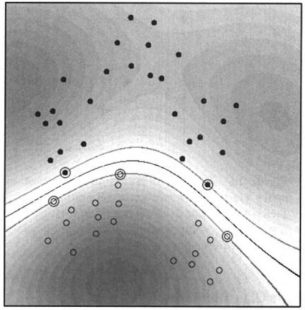


Figure 1.7 Example of an SV classifier found using a radial basis function kernel $k(x, x') = \exp(-\|x - x'\|^2)$ (here, the input space is $\mathcal{X} = [-1, 1]^2$). Circles and disks are two classes of training examples; the middle line is the decision surface; the outer lines precisely meet the 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

- 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!

What about general models?

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

- An example is a **support vector** if *removing* it from the training set and retraining changes f

- An example is a **support vector** if *removing* it from the training set and retraining changes f

- An example is a **support vector** for $\mathbf{x}_0$ if *removing* it from the training set and retraining changes $f(\mathbf{x}_0) = y_0$ (or p_θ)

- An example is a **support vector** if *removing* it from the training set and retraining changes f

- An example is a **support vector** for $\mathbf{x}_0$ if *removing* it from the training set and retraining changes $f(\mathbf{x}_0) = y_0$ (or p_θ)

- An example is **relevant** to $y_0 = f(\mathbf{x}_0)$ if *removing* it from the training set and retraining changes $f(\mathbf{x}_0)$ (or p_θ)

- An example is a **support vector** if *removing* it from the training set and retraining changes f

- An example is a **support vector** for $\mathbf{x}_0$ if *removing* it from the training set and retraining changes $f(\mathbf{x}_0) = y_0$ (or p_θ)

- An example is **relevant** to $y_0 = f(\mathbf{x}_0)$ if *removing* it from the training set and retraining changes $f(\mathbf{x}_0)$ (or p_θ)

Algorithm:

- Given:
  - A training set $S = \{(\mathbf{x}_i, y_i) : i \in [m]\}$
  - A classifier $f \in \mathcal{F}$ 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 $f_{-1} \in \mathcal{F}$
- The **relevance** of $(\mathbf{x}_i, y_i)$ is the difference between $f(\mathbf{x}_0)$ and $f(\mathbf{x}_0)$ (or p_θ)

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

Algorithm:

- Given:
  - A training set $S = \{(\mathbf{x}_i, y_i) : i \in [m]\}$
  - A classifier $f \in \mathcal{F}$ 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** $f_{-1} \in \mathcal{F} \leftarrow$ **slow**
- The **relevance** of $(\mathbf{x}_i, y_i)$ is the difference between $f(\mathbf{x}_0)$ and $f(\mathbf{x}_0)$ (or p_θ)

- Quite challenging if S is very large and/or f is a complex model (deep nets takes hours/days to retrain)
- Especially because one must retrain once for *each* i!
- Can we approximate the impact of removing $(\mathbf{x}_i, y_i)$ **without** retraining?

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].

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 and a data set $S = \{z_i = (\mathbf{x}_i, y_i) \ : \ i \in [m]\}$

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 and a data set $S = \{z_i = (\mathbf{x}_i, y_i) \ : \ i \in [m]\}$
- Let each $f \in \mathcal{F}$ be identified by parameters θ

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 and a data set $S = \{z_i = (\mathbf{x}_i, y_i) \ : \ i \in [m]\}$
- Let each $f \in \mathcal{F}$ be identified by parameters θ
- Let θ_m be the parameters of the **empirical risk minimizer** on S:

$$\theta_m \leftarrow \operatorname*{argmin}_{\theta} \ \frac{1}{m} \sum_k \ell(\theta, z_k)$$

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 and a data set $S = \{z_i = (\mathbf{x}_i, y_i) \ : \ i \in [m]\}$
- Let each $f \in \mathcal{F}$ be identified by parameters 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:

$$\theta_m(z, \epsilon) \leftarrow \operatorname*{argmin}_{\theta} \ \left(\frac{1}{m} \sum_k \ell(\theta, z_k)\right) + \epsilon \ell(\theta, z)$$

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 and a data set $S = \{z_i = (\mathbf{x}_i, y_i) \ : \ i \in [m]\}$
- Let each $f \in \mathcal{F}$ be identified by parameters 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:

$$\theta_m(z, \epsilon) \leftarrow \operatorname*{argmin}_{\theta} \ \left(\frac{1}{m} \sum_k \ell(\theta, z_k)\right) + \epsilon \ell(\theta, z)$$

Remark: $\theta_m = \theta_m(z, 0)$.

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 and a data set $S = \{z_i = (\mathbf{x}_i, y_i) \ : \ i \in [m]\}$
- Let each $f \in \mathcal{F}$ be identified by parameters 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:

$$\theta_m(z, \epsilon) \leftarrow \operatorname*{argmin}_{\theta} \ \left(\frac{1}{m} \sum_k \ell(\theta, z_k)\right) + \epsilon \ell(\theta, z)$$

Remark: $\theta_m = \theta_m(z, 0)$.
Remark: $\epsilon = \frac{1}{t}$ is equivalent to deleting z.

- Take a first-order Taylor expansion:

$$\theta_m(z,\epsilon) - \theta_m(z,0) \approx \epsilon \cdot \underbrace{\left(\frac{d}{d\epsilon}\theta_m(z,\epsilon)\Big|_{\epsilon=0} \right)}_{\text{influence function } \mathcal{I}(z)} \tag{1}$$

- Take a first-order Taylor expansion:

$$\theta_m(z,\epsilon) - \theta_m(z,0) \approx \epsilon \cdot \underbrace{\left(\frac{d}{d\epsilon}\theta_m(z,\epsilon)\Big|_{\epsilon=0} \right)}_{\text{influence function } \mathcal{I}(z)} \tag{1}$$

- 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:

$$\approx -\frac{1}{t} \cdot \mathcal{I}(z)$$

- Take a first-order Taylor expansion:

$$\theta_m(z,\epsilon) - \theta_m(z,0) \approx \epsilon \cdot \underbrace{\left(\frac{d}{d\epsilon}\theta_m(z,\epsilon)\Big|_{\epsilon=0} \right)}_{\text{influence function } \mathcal{I}(z)} \tag{1}$$

  - 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:

$$H(\theta_m) := \frac{1}{t} \sum_{k=1}^{t} \nabla_\theta^2 \ell(z_k, \theta_m), \qquad \nabla_\theta^2 \ell(z_k, \theta_m) = \left[\frac{\partial}{\partial \theta_s \partial \theta_t} \ell(z_k, \theta) \Big|_{\theta=\theta_m} \right]_{st}$$

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: $\|H(\theta_m)^{-1} \nabla_\theta \ell(z, \theta_m)\|$.

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!

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^*?

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:

$$\begin{aligned}
\frac{d}{d\epsilon} P(y^* \mid \mathbf{x}^*; \theta_m(z_k, \epsilon))\Big|_{\epsilon=0} &= \nabla_\theta P(y^* \mid \mathbf{x}^*; \theta_m)^\top \frac{d}{d\epsilon}\theta_m(z_k, \epsilon)\Big|_{\epsilon=0} \\
&= \nabla_\theta P(y^* \mid \mathbf{x}^*; \theta_m)^\top \mathcal{I}(z_k) \\
&= -\nabla_\theta P(y^* \mid \mathbf{x}^*; \theta_m)^\top H(\theta_m)^{-1}\nabla_\theta \ell(z, \theta_m)
\end{aligned}$$

This is a **scalar**, it approximates the **change in likelihood** at z^* by upscaling z by ϵ.

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:

$$\begin{aligned}
\frac{d}{d\epsilon} P(y^* \mid \mathbf{x}^*; \theta_m(z_k, \epsilon))\Big|_{\epsilon=0} &= \nabla_\theta P(y^* \mid \mathbf{x}^*; \theta_m)^\top \frac{d}{d\epsilon}\theta_m(z_k, \epsilon)\Big|_{\epsilon=0} \\
&= \nabla_\theta P(y^* \mid \mathbf{x}^*; \theta_m)^\top \mathcal{I}(z_k) \\
&= -\nabla_\theta P(y^* \mid \mathbf{x}^*; \theta_m)^\top H(\theta_m)^{-1}\nabla_\theta \ell(z, \theta_m)
\end{aligned}$$

This is a **scalar**, it approximates the **change in likelihood** at z^* by upscaling z by ϵ.

- The same trick works for other functions of θ_m, like the loss, the input gradients, *etc.*

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:

$$H(\theta_m) := \frac{1}{t} \sum_{k=1}^{t} \nabla_\theta^2 \ell(z_k, \theta_m), \qquad \nabla_\theta^2 \ell(z_k, \theta_m) = \left[\frac{\partial}{\partial \theta_s \partial \theta_t} \ell(z_k, \theta) \Big|_{\theta=\theta_m} \right]_{st}$$

The **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:

$$H(\theta_m) := \frac{1}{t} \sum_{k=1}^{t} \nabla_\theta^2 \ell(z_k, \theta_m), \qquad \nabla_\theta^2 \ell(z_k, \theta_m) = \left[\frac{\partial}{\partial \theta_s \partial \theta_t} \ell(z_k, \theta) \Big|_{\theta=\theta_m} \right]_{st}$$

- 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)

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 $-s^* \cdot \nabla_\theta \ell(z, \theta_m)$

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**

- HVP via **stochastich estimation** (LISSA) [Agarwal et al., 2017]

- HVP via **stochastich estimation** (LISSA) [Agarwal et al., 2017]

- Fix $j \in \mathbb{N}_0$ and consider:

$$H_j^{-1} = \sum_{i=0}^{j} (I - H)^i$$

This is the jth order **Taylor expansion** of H^{-1}, and $H_j^{-1} \to H^{-1}$ as $j \to \infty$.

- HVP via **stochastich estimation** (LISSA) [Agarwal et al., 2017]

- Fix $j \in \mathbb{N}_0$ and consider:

$$H_j^{-1} = \sum_{i=0}^{j} (I - H)^i$$

This is the jth order **Taylor expansion** of H^{-1}, and $H_j^{-1} \to H^{-1}$ as $j \to \infty$.

- This can be written **recursively** as:

$$H_j^{-1} = I + (I - H) H_{j-1}^{-1}$$

Can be showng by plugging the definition into the RHS we get:

$$\begin{aligned} I + (I - H) H_{j-1}^{-1} &= I + (I - H) \sum_{i=0}^{j-1} (I - H)^i = I + \sum_{i=1}^{j} (I - H)^i \\ &= (I - H)^0 + \sum_{i=1}^{j} (I - H)^i = \sum_{i=0}^{j} (I - H)^i = H_j^{-1} \end{aligned}$$

- HVP via **stochastich estimation** (LISSA) [Agarwal et al., 2017]

- HVP via **stochastich estimation** (LISSA) [Agarwal et al., 2017]

Idea: $\nabla^2_\theta \ell(\theta, z_i)$, where z_i is a single training point, is an **unbiased estimator** of H! This means that its average matches that of H.

- HVP via **stochastich estimation** (LISSA) [Agarwal et al., 2017]

Idea: $\nabla^2_\theta \ell(\theta, z_i)$, where z_i is a single training point, is an **unbiased estimator** of H! This means that its average matches that of H.

Algorithm for stochastic approximation of $H^{-1}\mathbf{v}$:

- Initialize $\tilde{H}_0^{-1}\mathbf{v} \leftarrow \mathbf{v}$
- Repeat:

$$\tilde{H}_j^{-1}\mathbf{v} \leftarrow \mathbf{v}I + (I - \nabla^2_\theta \ell(\theta, z_s))\tilde{H}_{j-1}^{-1}$$

  where $z_s \sim S$ is a single, random training set example.

- Then the average of $H_j^{-1}\mathbf{v}$ 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

How do IF compare to leave-one-out retraining?

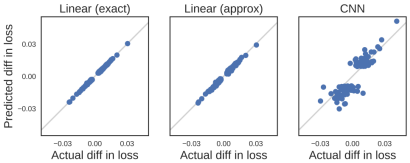


Figure 2. **Influence matches leave-one-out retraining.** We arbitrarily picked a wrongly-classified test point z_{test}, but this trend held more broadly. These results are from 10-class MNIST. **Left:** For each of the 500 training points z with largest $\left|\mathcal{I}_{\text{up,loss}}(z, z_{\text{test}})\right|$, 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"

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** [?]: the recursion can **diverge**!

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** [**?**]: the recursion can **diverge**!

Solutions: standard remedies include:

- **Fine-tuning** θ using a second-order method like L-BFGS [Koh and Liang, 2017] $\rightarrow$ this is "cheating", second-order methods are quite slow (sometimes comparably to retraining)
- Implicitly preconditioning H^{-1} $\rightarrow$ this smooths out the curvature, may be *insufficient*
- Weight decay [**?**] keeps $\|\theta\|$ small, only indirectly affects 2nd order derivatives, may be *insufficient*

Idea: replace Hessian with Fisher information matrix $F(\theta)$:

$$F(\theta) := \frac{1}{t-1} \sum_{k=1}^{t-1} \mathbb{E}_{y \sim P(Y \mid \mathbf{x}_k, \theta)} \left[\nabla_\theta \log P(y \mid \mathbf{x}_k, \theta) \nabla_\theta \log P(y \mid \mathbf{x}_k, \theta)^\top \right]$$

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

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.

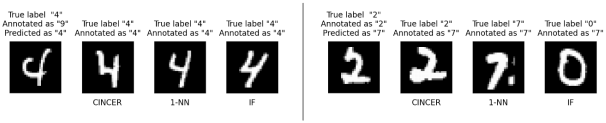


Figure 1: Suspicious example and counter-examples selected using (from left to right) CINCER, 1-NN and influence functions (IF), on noisy MNIST. **Left**: the suspicious example is mislabeled, the machine's suspicion is supported by a clean counter-example. **Right**: the suspicious example is not mislabeled, 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:

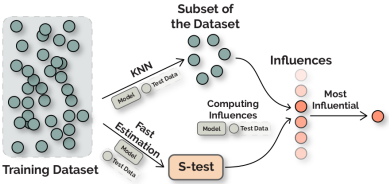


Figure 1: Workflow of our FASTIF w.r.t. a test data-point. First a subset of data-points are selected from the entire training set using kNN to reduce search space, then the inverse Hessian-vector product (s_{test}) is estimated based on Sec. 5.2. The influence values of data-points are computed using 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

- Factual explanations explain why a particular decision $y_0 = f(\mathbf{x}_0)$ 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**

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?

Enter **counterfactual** explanations:

- They explain why a particular outcome $y0$ was obtained instead of a (more desirable) alternative $y1$

Enter **counterfactual** explanations:

- They explain why a particular outcome $y0$ was obtained instead of a (more desirable) alternative $y1$

Intuition:

1. Given $\mathbf{x}_0$, look for the "**closest**" instance $\mathbf{x}_1 \in \mathbb{R}^d$ 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. Summarize the difference between $\mathbf{x}_0$ and $\mathbf{x}_1$ by, for instance, identifying the variables that differ between them:

$$\{i \in [d] \ : \ x_{0i} \neq x_{1i}\}$$

Enter **counterfactual** explanations:

- They explain why a particular outcome $y0$ was obtained instead of a (more desirable) alternative $y1$

Intuition:

1. Given $\mathbf{x}_0$, look for the "**closest**" instance $\mathbf{x}_1 \in \mathbb{R}^d$ 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. Summarize the difference between $\mathbf{x}_0$ and $\mathbf{x}_1$ by, for instance, identifying the variables that differ between them:

$$\{i \in [d] \ : \ x_{0i} \neq x_{1i}\}$$

■ Picking $\mathbf{x}_1$ to be "close" to $\mathbf{x}_0$ encourages the difference to be minimal/sparse and easy to summarize, thus **interpretable**

Algorithm: it is easy to see that counterfactual examples $\mathbf{x}_1$ can be obtained by solving:

$$\mathbf{x}_1 \leftarrow \underset{\mathbf{x} \in \mathbb{R}^d}{\text{argmax}} \; \|\mathbf{x}_0 - \mathbf{x}_1\|_0$$

$$\text{s.t. } f(\mathbf{x}_1) = y_1 \qquad (\text{or } f(\mathbf{x}_1) \neq f(\mathbf{x}_0))$$

Algorithm: it is easy to see that counterfactual examples $\mathbf{x}_1$ can be obtained by solving:

$$\mathbf{x}_1 \leftarrow \underset{\mathbf{x} \in \mathbb{R}^d}{\text{argmax}} \; \|\mathbf{x}_0 - \mathbf{x}_1\|_0$$
$$\text{s.t. } f(\mathbf{x}_1) = y_1 \qquad (\text{or } f(\mathbf{x}_1) \neq f(\mathbf{x}_0))$$

- The pseudo-norm $\|\cdot\|_0$ counts the number of non-zero entries, i.e., we **minimize the number of entries that differ between $\mathbf{x}_0$ and $\mathbf{x}_1$** $\rightarrow$ *sparse explanation*

Algorithm: it is easy to see that counterfactual examples $\mathbf{x}_1$ can be obtained by solving:

$$\mathbf{x}_1 \leftarrow \underset{\mathbf{x} \in \mathbb{R}^d}{\operatorname{argmax}} \; \|\mathbf{x}_0 - \mathbf{x}_1\|_0$$
$$\text{s.t. } f(\mathbf{x}_1) = y_1 \qquad (\text{or } f(\mathbf{x}_1) \neq f(\mathbf{x}_0))$$

- The pseudo-norm $\|\cdot\|_0$ counts the number of non-zero entries, i.e., we **minimize the number of entries that differ between $\mathbf{x}_0$ and $\mathbf{x}_1$** $\rightarrow$ *sparse explanation*

- Once again, replace with L_1 norm $\|\cdot\|_1$ to obtain a more tractable optimization problem (as for LIME earlier).

Algorithm (Updated):

$$\mathbf{x}_1 \leftarrow \underset{\mathbf{x} \in \mathbb{R}^d}{\operatorname{argmax}} \; \|\mathbf{x}_0 - \mathbf{x}_1\|_1$$

$$\text{s.t. } f(\mathbf{x}_1) = y_1 \qquad (\text{or } f(\mathbf{x}_1) \neq f(\mathbf{x}_0))$$

Algorithm (Updated):

$$\mathbf{x}_1 \leftarrow \underset{\mathbf{x} \in \mathbb{R}^d}{\operatorname{argmax}} \; \|\mathbf{x}_0 - \mathbf{x}_1\|_1$$
$$\text{s.t. } f(\mathbf{x}_1) = y_1 \qquad (\text{or } f(\mathbf{x}_1) \neq f(\mathbf{x}_0))$$

- How do we solve this?
  - Use **gradient descent** (*aka*, "if you have a hammer, every problem you see looks like a nail")

    Start from $\mathbf{x}_0$ and follow the gradient of p_θ. This will usually give you *a* solution – but not necessarily, and not necessarily the closest one.

    This strategy sports **no guaranteees**.

Algorithm (Updated):

$$\mathbf{x}_1 \leftarrow \underset{\mathbf{x} \in \mathbb{R}^d}{\operatorname{argmax}} \; \|\mathbf{x}_0 - \mathbf{x}_1\|_1$$
$$\text{s.t. } f(\mathbf{x}_1) = y_1 \qquad (\text{or } f(\mathbf{x}_1) \neq f(\mathbf{x}_0))$$

- How do we solve this?
  - Use **gradient descent** (*aka*, "if you have a hammer, every problem you see looks like a nail")

    Start from $\mathbf{x}_0$ and follow the gradient of p_θ. This will usually give you *a* solution – but not necessarily, and not necessarily the closest one.

    This strategy sports **no guaranteees**.
  - Use **model-specific procedures**.

Algorithm (Updated):

$$\mathbf{x}_1 \leftarrow \underset{\mathbf{x} \in \mathbb{R}^d}{\operatorname{argmax}} \; \|\mathbf{x}_0 - \mathbf{x}_1\|_1$$
$$\text{s.t. } f(\mathbf{x}_1) = y_1 \qquad (\text{or } f(\mathbf{x}_1) \neq f(\mathbf{x}_0))$$

- How do we solve this?
  - Use **gradient descent** (*aka*, "if you have a hammer, every problem you see looks like a nail")

    Start from $\mathbf{x}_0$ and follow the gradient of p_θ. This will usually give you *a* solution – but not necessarily, and not necessarily the closest one.

    This strategy sports **no guaranteees**.
  - Use **model-specific procedures**.
  - Use **mathematical programming**.

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 $y_\ell \in [c]$

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]

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 $y_{\ell'} = y_1$.

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 $\min_{\mathbf{x}' \models \phi_{\ell'}} \|\mathbf{x}_0 - \mathbf{x}'\|_1$

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 $\mathbf{x}_1$.

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)

Alternative: simply encode the whole problem using, e.g., **mixed-integer linear programming** (MILP)

Mixed-integer Linear Program

An optimization program is a MILP if it can be written as:

$$\min_{\mathbf{x}} \mathbf{c}^\top \mathbf{x} \tag{2}$$

$$\text{s.t. } A\mathbf{x} \leq \mathbf{b} \qquad (\text{equiv. } \forall j \ \mathbf{a}_j^\top \mathbf{x} \leq b_j) \tag{3}$$

$$\forall i \in \mathcal{I}_\mathsf{C} \ x_i \in \mathbb{R} \tag{4}$$

$$\forall i \in \mathcal{I}_\mathsf{I} \ x_i \in \mathbb{Z} \tag{5}$$

$$\mathcal{I}_\mathsf{C} \cup \mathcal{I}_\mathsf{I} = [d] \tag{6}$$

$$\mathcal{I}_\mathsf{C} \cap \mathcal{I}_\mathsf{I} = \varnothing \tag{7}$$

In other words, (i) the cost is a **linear function** of the input $\mathbf{x}$, (ii) the feasible space is a **conjunction of hyperplanes** (i.e., a convex polytope)

Notice that some of the variables are continuous while the others are integral.

Alternative: simply encode the whole problem using, e.g., **mixed-integer linear programming** (MILP)

Mixed-integer Linear Program

An optimization program is a MILP if it can be written as:

$$\min_{\mathbf{x}} \mathbf{c}^\top \mathbf{x} \tag{2}$$

$$\text{s.t. } A\mathbf{x} \leq \mathbf{b} \qquad (\text{equiv. } \forall j \ \mathbf{a}_j^\top \mathbf{x} \leq b_j) \tag{3}$$

$$\forall i \in \mathcal{I}_\mathsf{C} \ x_i \in \mathbb{R} \tag{4}$$

$$\forall i \in \mathcal{I}_\mathsf{I} \ x_i \in \mathbb{Z} \tag{5}$$

$$\mathcal{I}_\mathsf{C} \cup \mathcal{I}_\mathsf{I} = [d] \tag{6}$$

$$\mathcal{I}_\mathsf{C} \cap \mathcal{I}_\mathsf{I} = \varnothing \tag{7}$$

In other words, (i) the cost is a **linear function** of the input $\mathbf{x}$, (ii) the feasible space is a **conjunction of hyperplanes** (i.e., a convex polytope)

Notice that some of the variables are continuous while the others are integral.

- Can be solved with excellent **off-the-shelf solver** like Gurobi, CPLEX, SCIP, ...

Alternative: simply encode the whole problem using, e.g., **mixed-integer linear programming** (MILP)

Mixed-integer Linear Program

An optimization program is a MILP if it can be written as:

$$\min_{\mathbf{x}} \mathbf{c}^\top \mathbf{x} \tag{2}$$

$$\text{s.t. } A\mathbf{x} \leq \mathbf{b} \qquad (\text{equiv. } \forall j \ \mathbf{a}_j^\top \mathbf{x} \leq b_j) \tag{3}$$

$$\forall i \in \mathcal{I}_\mathsf{C} \ x_i \in \mathbb{R} \tag{4}$$

$$\forall i \in \mathcal{I}_\mathsf{I} \ x_i \in \mathbb{Z} \tag{5}$$

$$\mathcal{I}_\mathsf{C} \cup \mathcal{I}_\mathsf{I} = [d] \tag{6}$$

$$\mathcal{I}_\mathsf{C} \cap \mathcal{I}_\mathsf{I} = \varnothing \tag{7}$$

In other words, (i) the cost is a **linear function** of the input $\mathbf{x}$, (ii) the feasible space is a **conjunction of hyperplanes** (i.e., a convex polytope)

Notice that some of the variables are continuous while the others are integral.

- Can be solved with excellent **off-the-shelf solver** like Gurobi, CPLEX, SCIP, ...
- Can we encode the counterfactual search problem as MILP?

Encoding: finding a counterfactual example for a DT:

$$\underset{\mathbf{x}_1}{\text{argmin}} \sum_{j \in [d]} |x_{0j} - x_{1j}| \tag{8}$$

$$\text{s.t. } \mathbf{a}_{\ell,f}^\top \mathbf{x} - b_{\ell,f} \leq 0 \qquad \forall \ell : y_\ell = y_1, \text{face } f \tag{9}$$

Encoding: finding a counterfactual example for a DT:

$$\underset{\mathbf{x}_1}{\text{argmin}} \sum_{j \in [d]} |x_{0j} - x_{1j}| \tag{8}$$

$$\text{s.t. } \mathbf{a}_{\ell,f}^\top \mathbf{x} - b_{\ell,f} \leq 0 \qquad \forall \ell : y_\ell = y_1, \text{face } f \tag{9}$$

- Wait, what?

Encoding: finding a counterfactual example for a DT:

$$\underset{\mathbf{x}_1}{\text{argmin}} \sum_{j \in [d]} |x_{0j} - x_{1j}| \tag{8}$$

$$\text{s.t. } \mathbf{a}_{\ell,f}^\top \mathbf{x} - b_{\ell,f} \leq 0 \qquad \forall \ell : y_\ell = y_1, \text{face } f \tag{9}$$

- Wait, what?

- This is **wrong**!

Encoding: finding a counterfactual example for a DT:

$$\underset{\mathbf{x}_1}{\text{argmin}} \sum_{j \in [d]} |x_{0j} - x_{1j}| \tag{8}$$

$$\text{s.t. } \mathbf{a}_{\ell,f}^\top \mathbf{x} - b_{\ell,f} \leq 0 \qquad \forall \ell : y_\ell = y_1, \text{face } f \tag{9}$$

- Wait, what?
- This is **wrong**!
- **Whoops**!

Encoding: finding a counterfactual example for a DT:

$$\underset{\mathbf{x}_1}{\text{argmin}} \sum_{j \in [d]} |x_{0j} - x_{1j}| \tag{10}$$

$$\text{s.t. } \mathbf{a}_{\ell,f}^\top \mathbf{x} - b_{\ell,f} \leq \epsilon_\ell \qquad \forall \ell : y_\ell = y_1, \text{face } f \tag{11}$$

$$\epsilon \leq \epsilon_\ell \qquad \forall \ell : y_\ell = y_1 \tag{12}$$

$$\epsilon \leq 0 \tag{13}$$

Encoding: finding a counterfactual example for a DT:

$$\underset{\mathbf{x}_1}{\text{argmin}} \sum_{j \in [d]} |x_{0j} - x_{1j}| \tag{10}$$

$$\text{s.t. } \mathbf{a}_{\ell,f}^\top \mathbf{x} - b_{\ell,f} \leq \epsilon_\ell \qquad \forall \ell : y_\ell = y_1, \text{face } f \tag{11}$$

$$\epsilon \leq \epsilon_\ell \qquad \forall \ell : y_\ell = y_1 \tag{12}$$

$$\epsilon \leq 0 \tag{13}$$

This strategy:

+ works for all models with a **piecewise-linear** decision surface. This includes: DTs, random forest classifiers and regressors, kernel machines with piecewise-linear kernels, neural nets with ReLU activations, ...
- the encoding can be non-trivial and lead to a **practically hard** optimization problem

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)

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.

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 $\mathbf{x}'$.

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.

Byrne, R. M. (2019).
Counterfactuals in explainable artificial intelligence (xai): Evidence from human reasoning.
In *IJCAI*, pages 6276–6282.

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*.