

## Introduction

Advances in technology have typically gone hand in hand with an outsourcing of labour from humans to machines: the printing press succeeded human scribes centuries ago, ATMs replaced bank tellers decades ago and today robots are swarming factory floors. While these transitions involved a substitution of manual or repetitive tasks, recent advances in computing and artificial intelligence (AI) have accelerated a new type of transformation: from human to data-driven decision-making. Today, it is more likely than not that your digital loan or employment application will be handled by an algorithm, at least in the first instance. This can in theory be beneficial to you: automation typically leads to increased efficiency and has the potential to remove human bias and error. In reality though, state-of-the-art algorithms are often instable ([?]), encode existing biases ([?]) and learn representations that are surprising or even counter-intuitive from a human perspective (**REFERENES?** []).

This is made more problematic by the fact that many modern machine learning algorithms tend to be so complex and underspecified in the data, that they are essentially black boxes. While this is a known issue, such models are still used to guide decision-making and research in industry as well as academia. At the time of writing, the largest artificial neural networks are made up of several hundreds of billion neurons. In the context of high-stake decision-making systems, black-box models create undesirable dynamics: the human operators in charge of the system have to rely on it blindly, while those individuals subject to it generally have no way to challenge an outcome. If your digital loan or employment application gets rejected, for example, that is typically the end of the story.

“You cannot appeal to (algorithms). They do not listen. Nor do they bend.”

— Cathy O’Neil in *Weapons of Math Destruction*, 2016

While the inappropriate abuse of such technologies is arguably the biggest issue, we should also be concerned about missed opportunities. The lack of trustworthiness in machine learning prevents it from being adopted in other fields of research, which might actually benefit from its adoption. Economics and financial markets, for example, are full of complexities and non-linearities that machine learning algorithms are well-equipped to model. But financial practitioners and policy makers are understandably wary of using tools they cannot fully understand ([?],[?]).

In light of all this, a quickly growing body of literature on explainable artificial intelligence has emerged. Counterfactual explanations (CE) and algorithmic recourse (AR) fall into this broader category. Counterfactual explanations can help programmers make sense of the systems they build: they explain how inputs into a system need to change for it to produce different decisions. Explanations that involve realistic and actionable changes can be used for the purpose of algorithmic recourse (AR): they offer individuals subject to algorithms a way to turn a negative decision into positive one. Through the **CounterfactualExplanations** package we aim to contribute a scalable and versatile implementation of CE and AR to the Julia community. The

remainder of this article is structured as follows: Section ?? presents related work on explainable AI, Section ?? provides a brief overview of the methodological framework, Section ?? presents the package functionality and Section ?? concludes.

## Related work

### Literature on explainable AI

The field of explainable artificial intelligence (xAI) is still relatively young and made up of a variety of subdomains, definitions, concepts and taxonomies. Covering all of these is beyond the scope of this article, so we will focus only on high-level concepts. The following literature surveys provide more detail: [?] provide a broad overview of xAI; [?] focus on explainability in the context of deep learning; and finally, [?] offer a detailed review of the literature on counterfactual explanations and algorithmic recourse.<sup>1</sup>

The first broad distinction we want to make here is between **interpretable** and **explainable** AI. These terms are often used interchangeably, but this can lead to confusion. We find the distinction made in [?] useful: interpretable AI involves models that are inherently interpretable such as general additive models (GAM), decision trees and rule-based models; explainable AI involves models that are not inherently interpretable, but require additional tools to be explainable to humans. Examples of the latter include ensembles, support vector machines and deep neural networks. Some would argue that we best avoid the second category of models [[?]] and instead focus solely on interpretable AI. While we agree that initial efforts should always be geared towards interpretable models, avoiding black boxes altogether would entail missed opportunities and anyway is probably not very realistic at this point. For that reason, we expect the need for explainable AI to persist in the near future.

Explainable AI can further be broadly divided into **global** and **local** explainability.

### Existing software

## Methodological background

Counterfactual search happens in the feature space: we are interested in understanding how we need to change individual attributes in order to change the model output to a desired value or label ([?]). Typically the underlying methodology is presented in the context of binary classification:  $M : \mathcal{X} \mapsto y$  where  $y \in \{0, 1\}$ . Let  $t = 1$  be the target class and let  $\bar{x}$  denote the factual feature vector of some individual outside of the target class, so  $\bar{y} = M(\bar{x}) = 0$ . We follow this convention here, though it should be noted that the ideas presented here also carry over to multi-class problems and regression ([?]).

---

<sup>1</sup>Readers who prefer a text-book approach may also want to consider [?] and [?]

## Generic framework

Then the counterfactual search objective originally proposed by [?] is as follows

$$\min_{\underline{x} \in \mathcal{X}} h(\underline{x}) \quad \text{s. t.} \quad M(\underline{x}) = t \quad (1)$$

where  $h(\cdot)$  quantifies how complex or costly it is to go from the factual  $\bar{x}$  to the counterfactual  $\underline{x}$ . To simplify things we can restate this constrained objective (Equation ??) as the following unconstrained and differentiable problem:

$$\underline{x} = \arg \min_{\underline{x}} \ell(M(\underline{x}), t) + \lambda h(\underline{x}) \quad (2)$$

Here  $\ell$  denotes some loss function targeting the deviation between the target label and the predicted label and  $\lambda$  governs the strength of the complexity penalty. Provided we have gradient access for the black-box model  $M$  the solution to this problem (Equation ??) can be found through gradient descent. This generic framework lays the foundation for most state-of-the-art approaches to counterfactual search and is also used as the baseline approach - **GenericGenerator** - in our package. The hyperparameter  $\lambda$  is typically tuned through grid search. Conventional choices for  $\ell$  include margin-based losses like cross-entropy loss and hinge loss. It is worth pointing out that the loss function is typically computed with respect to logits rather than predicted probabilities, a convention that we have chosen to follow.<sup>2</sup>

Numerous - and in some cases competing - extensions to this simple approach have been developed since counterfactual explanations were first proposed in 2017 (see [?] and [?] for surveys). The various approaches largely differ in how they define the complexity penalty. In [?], for example,  $h(\cdot)$  is defined in terms of the Manhattan distance between factual and counterfactual feature values. While this is an intuitive choice, it is too simple to address many of the desirable properties of effective counterfactual explanations that have been set out. These desiderata include: **closeness** - the average distance between factual and counterfactual features should be small ([?]); **actionability** - the proposed feature perturbation should actually be actionable ([?], [?]); **plausibility** - the counterfactual explanation should be plausible to a human ([?]); **unambiguity** - a human should have no trouble assigning a label to the counterfactual ([?]); **sparsity** - the counterfactual explanation should involve as few individual feature changes as possible ([?]); **robustness** - the counterfactual explanation should be robust to domain and model shifts ([?]); **diversity** - ideally multiple diverse counterfactual explanations should be provided ([?]); and **causality** - counterfactual explanations reflect the structural causal model underlying the data generating process ([?],[?]).

---

<sup>2</sup>While the rationale for this convention is not entirely obvious, implementations of loss functions with respect to logits are often numerically more stable. For example, the `logitbinarycrossentropy( $\hat{y}$ ,  $y$ )` implementation in `Flux.Losses` (used here) is more stable than the mathematically equivalent `binarycrossentropy( $\hat{y}$ ,  $y$ )`.

## Counterfactuals for Bayesian models

For what follows it is worth elaborating on the approach proposed in [?]. The authors demonstrate that many of the abovementioned desiderata can be addressed very easily, if the classifier  $M$  is Bayesian. In particular, they show that close, realistic, sparse and unambiguous counterfactuals can be generated by implicitly minimizing the classifier’s predictive uncertainty through a greedy counterfactual search. Formally, they define  $h(\cdot)$  as the predictive entropy of the classifier, which captures both **epistemic** and **aleatoric** uncertainty: the former is high on points far away from the training data while the latter is high in regions of the input space that are inherently noisy. Both are regions we want to steer clear off in our counterfactual search and hence predictive entropy is an intuitive choice for a complexity penalty. The authors further point out that any solution that minimizes cross-entropy loss (Equation ??) also minimizes predictive entropy:  $\arg \min_{\underline{x}} \ell(M(\underline{x}), t) \in \arg \min_{\underline{x}} h(\underline{x})$ . Let  $\widetilde{\mathcal{M}}$  denote the class of binary classifiers that incorporate predictive uncertainty, then the previous observation implies that the optimal solution to counterfactual search (Equation ??) can be restated as follows:

$$\underline{x} = \arg \min_{\underline{x}} \ell(M(\underline{x}), t) \quad , \quad \forall M \in \widetilde{\mathcal{M}} \quad (3)$$

We can drop the complexity penalty altogether and still generate effective counterfactual explanations. As we will see below, even a fast and greedy counterfactual search proposed in [?] yields good results in this setting. The approach has been implemented as **GreedyGenerator** in our package and should only be used with classifiers of type  $\widetilde{\mathcal{M}}$ . It is worth noting that the findings in [?] are not mutually exclusive of many of the other methodologies that have been put forward. On the contrary, we believe that they are complementary: the generic counterfactual search proposed in [?], for example, can be shown to produce more plausible counterfactuals in the Bayesian setting. Similarly, there is no obvious reason why recent work on diversity ([?]), robustness ([?]) and causality ([?],[?]) could not be complemented by the findings in [?]. For this reason we are highlighting [?] here and have prioritized it in the development of **CounterfactualExplanations**. While there is no free lunch and  $M \in \widetilde{\mathcal{M}}$  may seem like a hard constraint, recent advances in probabilistic machine learning have shown that the computational cost involved in Bayesian model averaging is lower than we may have thought ([?], [?], [?], [?]).

## Using CounterfactualExplanations

The package is built around two modules that are designed to be as scalable as possible through multiple dispatch: 1) **Models** is concerned with making any arbitrary model compatible with the package; 2) **Generators** is used to implement arbitrary counterfactual search algorithms.<sup>3</sup>

---

<sup>3</sup>We have made an effort to keep the code base as flexible and scalable as possible, but cannot guarantee at this point that really any counterfactual generator can be implemented without further adaptation.

The core function of the package `generate_counterfactual` uses an instance of type `T <: FittedModel` produced by the `Models` module (Figure ??) and an instance of type `T <: Generator` produced by the `Generators` module (Figure ??). Relating this back the methodology outlined in Section ??, the former instance corresponds to the model  $M$  while the latter defines the rules for the counterfactual search (Equation ?? and Equation ??). In the following we will demonstrate how to use and extend the package architecture through a few examples.

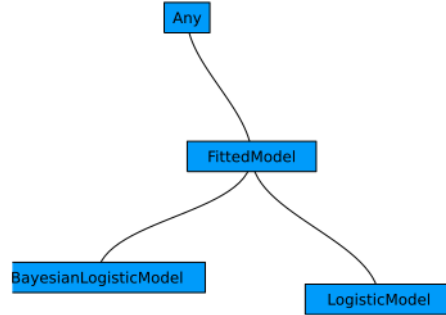


Figure 1: Schematic overview of the `FittedModel` base type and its descendants.

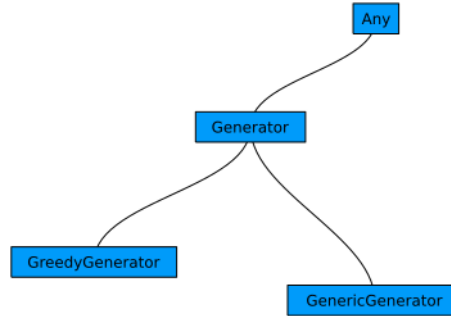


Figure 2: Schematic overview of the `Generator` base type and its descendants.

## Getting started

The code below provides a complete example demonstrating how the framework presented in Section ?? can be implemented in Julia using the `CounterfactualExplanations` package: using a synthetic data set with linearly separable samples we firstly define our model and then generate a counterfactual for a randomly selected sample. Figure ?? shows the resulting counterfactual path in the two-dimensional feature space: features go through iterative perturbations until the desired confidence level is reached as illustrated by the contour in the background, which indicates the classifier's predicted probability that the label is equal to 1.

It may help to go through the relevant parts of the code in some more detail starting from the part involving the model. For illustrative purposes the `Models` module ships with a constructor for a logistic regression model: `LogisticModel(W::Matrix, b::AbstractArray) <: FittedModel`. This constructor does not fit the regression model, but rather takes its underlying parameters as given. In other words, it is generally assumed that the user has already estimated a model. Based on the provided estimates two functions are already implemented that compute logits and probabilities for the model, respectively. Below we will see how users can use multiple dispatch to extend these functions for use with arbitrary models. For now it is enough to note that those methods define how the model makes its predictions  $M(x)$  and hence they form an integral part of the counterfactual search.

With the model  $M$  defined in the code below we go on to set up the counterfactual search as follows: 1) choose a random sample `x_factual`; 2) compute its factual label `y_factual` as predicted by the model ( $M(\bar{x}) = 0$ ); and 3) specify the other class as our `target` label ( $t = 1$ ) along with a desired level of `confidence` in the final prediction  $M(x) = t$ .

The last two lines of the code below define the counterfactual generator and finally run the counterfactual search. The first three fields of the `GenericGenerator` are reserved for hyperparameters governing the strength of the complexity penalty, the step size for gradient descent and the tolerance for convergence. The fourth field accepts a `Symbol` defining the type of loss function  $\ell$  to be used. Since we are dealing with a binary classification problem logit binary cross-entropy is an appropriate choice.<sup>4</sup> The fifth and last field can be used to define mutability constraints for the features.

```
[language = Julia] Data: using CounterfactualExplanations, Random Random.seed!(1234) N = 100
number of data points using CounterfactualExplanations.Data x, y = toy_data_linear(N)
```

```
Model: using CounterfactualExplanations.Models w = [1.0 1.0] true coefficients b = 0 M = LogisticModel(w, [b])
```

```
Setup: x_factual = x[rand(1 : length(x))] y_factual = round(probs(M, x_factual)[1]) target =
ifelse(y_factual == 1.0, 0.0, 1.0) confidence = 0.75
```

---

<sup>4</sup>As mentioned earlier, the loss function is computed with respect to logits and hence it is important to use logit binary cross-entropy loss as opposed to just binary cross-entropy.

```

Counterfactual search: generator = GenericGenerator( 0.1,0.1,1e-5,:logitbinarycrossentropy,nothing)
counterfactual = generate_counterfactual(generator, x_actual, M, target, confidence)

```

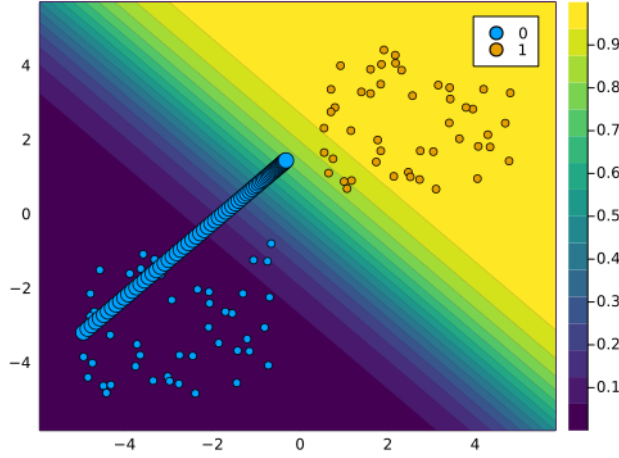


Figure 3: Counterfactual path using generic counterfactual generator for conventional binary classifier.

In this simple example the generic generator produces an effective counterfactual: the decision boundary is crossed (i.e. the counterfactual explanation is valid) and upon visual inspection the counterfactual seems plausible (Figure ??). Still, the example also illustrates that things may well go wrong: since the underlying model produces high-confidence predictions in regions free of any data, it is easy to think of scenarios that involve valid but unrealistic or ambiguous counterfactuals. Consider, for example, the scenario illustrated in Figure ??, which involves the same logistic classifier albeit massively overfitted. In this case generic search may yield an unrealistic counterfactual that is well into the yellow region and yet far away from all other samples (red marker) or an ambiguous counterfactual near the decision boundary (black marker).

Among the different approaches that have recently been put forward to deal with such issues is the greedy generator for Bayesian models proposed by [?]. For reasons discussed in Section ??, we have chosen to prioritize this approach in the development of **CounterfactualExplanations**. The code below shows how this approach can be implemented. Figure ?? shows the resulting counterfactual path through the feature space along with the predicted probabilities from the Bayesian classifier.

Once again it is worth dwelling on the code for a moment. We have used the same synthetic toy data as before, but this time we use assume that we have fit a Bayesian logistic regression model through Laplace approximation. This approximation uses the fact the second-order Taylor expansion of the logit binary cross-entropy function evaluated at the maximum-a-posteriori (MAP) estimate amounts to a multivariate Gaussian distribution ([?]).<sup>5</sup> The

<sup>5</sup>See also this [blog post](#) for a gentle introduction and implementation in Julia.

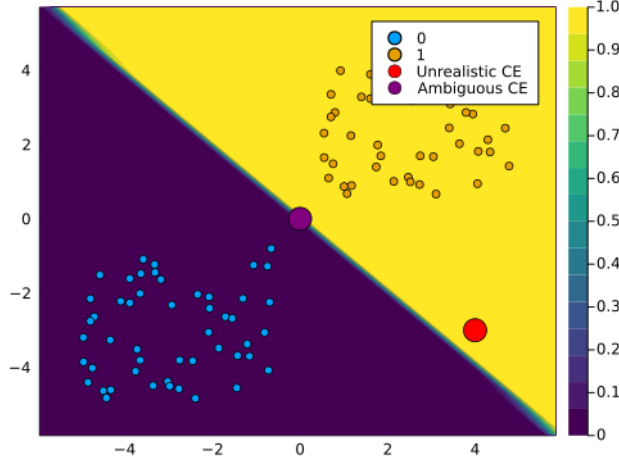


Figure 4: Unrealistic and ambiguous counterfactuals that may be produced by generic counterfactual search for an overfitted conventional binary classifier.

`BayesianLogisticModel` <: `FittedModel` constructor takes the two moments defining that distribution as its arguments: firstly, the MAP estimate, i.e. the vector of parameters  $\hat{\mu}$  including the constant term and, secondly, the corresponding covariance matrix  $\hat{\Sigma}$ . As with logistic regression above, the package ships with methods to compute predictions from instances of type `BayesianLogisticModel`.<sup>6</sup> Contrary to the simple logistic regression model above, predictions from the Bayesian logistic model incorporate uncertainty and hence predicted probabilities fan out in regions free of any training data (Figure ??).

For the counterfactual search we use a greedy approach following [?]. The approach is greedy in the sense that in each iteration it selects the most salient feature with respect to our objective (Equation ??) and perturbs it by some predetermined step size  $\delta$ . Since the gradient  $\nabla_{\underline{x}} \ell(M(\underline{x}, t))$  is proportional to the MAP estimate  $\hat{\mu}$ , the same feature is chosen until a predefined maximum number of perturbations  $n$  has been exhausted. Those two hyperparameters,  $\delta$  and  $n$ , are defined in the first two fields of `GreedyGenerator` <: `Generator` in the code below. The third and fourth field are reserved for the loss function and mutability constraints. Since we are making use of multiple dispatch, the final command that actually runs the counterfactual search is the same as before.

```
[language = Julia] Model: using LinearAlgebra I = UniformScaling(1) cov = Symmetric(reshape(randn(9),3,3).*0.01 + I) w = [1 1] params = hcat(b, w) M = BayesianLogisticModel(params, cov)
```

```
Counterfactual search: generator = GreedyGenerator( 0.25,20,;logitbinarycrossentropy,nothing)
counterfactual = generate_counterfactual(generator, x_factual, M, target, confidence)
```

<sup>6</sup>Predictions are computed using a probit approximation.



The counterfactual in Figure ?? is not only valid, but also realistic and unambiguous. In this case it is more difficult to imagine adverse scenarios like in Figure ?. Evidently it is easier to avoid pitfalls when generating counterfactual explanations for models that incorporate predictive uncertainty.

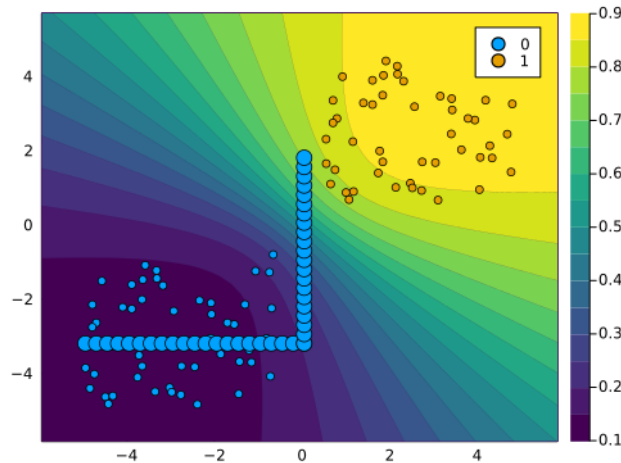


Figure 5: Counterfactual path using greedy counterfactual generator for Bayesian binary classifier.

## Custom models

One of our priorities has been to make `CounterfactualExplanations` scalable and versatile. In the long term we aim to add support for more default models and counterfactual generators. In the short term it is designed to allow users to integrate models and generators themselves. Ideally, these community efforts will facilitate our long-term goals. Only two steps are necessary to make any supervised-learning model compatible with our package<sup>7</sup>:

To demonstrate how this can be done in practice we will now consider another synthetic example. Once again samples are two-dimensional for illustration purposes, but this time they are grouped into four different classes and not linearly separable. To predict class labels based on features we use a simple deep-learning model trained in `Flux.jl` ([?]). The code below shows the simple model architecture. Note how outputs from the final layer are not passed through a softmax activation function, since counterfactual loss is evaluated with respect to logits as we discussed earlier. The model is trained with dropout for ten training epochs.

```
[language = Julia] n_hidden = 32 output_dim = length(unique(y)) input_dim = 2 model = Chain(Dense(input_dim, n_hidden, activation), Dropout(0.1), Dense(n_hidden, output_dim))
```

<sup>7</sup>In order for the model to be compatible with the gradient-based default generators presented in Section ?? gradient access is also necessary, but any model can also be complemented with a custom generator.

The code below implements the two steps that are necessary to make the trained neural network compatible with the package: subtyping and multiple dispatch. Computing logits amounts to just calling the Flux.jl model on inputs. Predicted probabilities for labels can then be computed through softmax.

```
[language = Julia] Step 1) struct NeuralNetwork <: Models.FittedModel model::Any end
```

```
Step 2) import functions in order to extend import CounterfactualExplanations.Models: logits
import CounterfactualExplanations.Models: probs logits(M::NeuralNetwork, X::AbstractArray)
= M.model(X) probs(M::NeuralNetwork, X::AbstractArray) = softmax(logits(M, X)) M =
NeuralNetwork(model)
```

Finally, the code below draws a random sample and generates a counterfactual in a different target class through generic search. The code very much resembles the earlier examples, with the only notable difference that for the counterfactual loss function we are now using the multi-class logit cross-entropy loss. The resulting counterfactual path is shown in Figure ???. In this case the contour shows the predicted probability that the input is in the target class ( $t = 4$ ). Generic search yields a valid, realistic and unambiguous counterfactual.

```
[language = Julia] Randomly selected factual: using Random Random.seed!(42) x_factual =
x[rand(1 : length(x))] y_factual = Flux.onecold(probs(M, x_factual), unique(y)) target =
rand(unique(y)[1 : end] != y_factual) confidence = 0.75
```

```
Counterfactual search: generator = GenericGenerator( 0.1,0.1,1e-5,:logitcrossentropy,nothing)
counterfactual = generate_counterfactual(generator, x_factual, M, target, confidence)
```

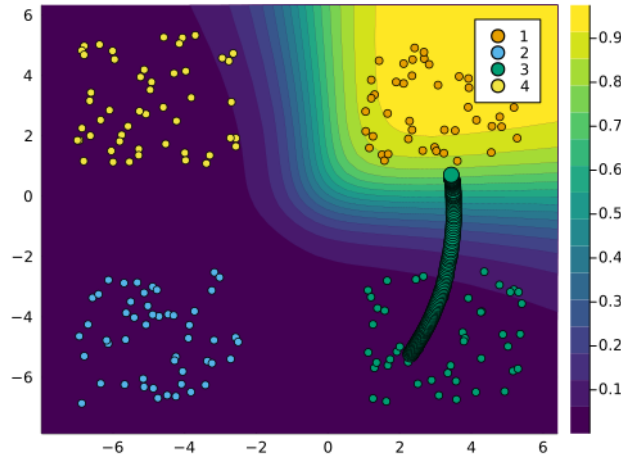


Figure 6: Counterfactual path using generic counterfactual generator for multi-class classifier.

As before we will also look at the Bayesian setting. Using Laplace approximation (LA) much in the same way as above we can recover a Bayesian representation of our neural network in a post-hoc fashion ([?]). Alternatively, we could have considered using a deep ensemble

([?]), Monte Carlo dropout ([?]) or variational inference. Using the greedy generator yields the counterfactual path in Figure ?? . The code that produces these results follows below.

Contrary to the example involving binary classification above, it is less clear that counterfactuals for the Bayesian classifier are more effective in this case. While predictions from this simple Bayesian neural network are overall more conservative, the model fails to only produce high-confidence predictions in regions that are abundant with training samples. This illustrates that the quality of counterfactual explanations may ultimately depend to some degree on the quality of the classifier. Put differently, if the quality of the classifier is poor, we may expect this to come through in the counterfactual explanation.

Fitting the Laplace approximation: using BayesLaplace `la = laplace(model) fit!(la, data)`

Model: Step 1) `struct LaplaceNeuralNetwork <: Models.FittedModel la::BayesLaplace.LaplaceRedux`  
`end`

Step 2) `logits(M::LaplaceNeuralNetwork, X::AbstractArray) = M.la.model(X)` `probs(M::LaplaceNeuralNetwork, X::AbstractArray) = BayesLaplace.predict(M.la, X)` `M = LaplaceNeuralNetwork(la)`

Counterfactual search: `generator = GreedyGenerator( 0.25,30,:logitcrossentropy,nothing)` `counterfactual = generate_counterfactual(generator, x_factual, M, target, confidence)`

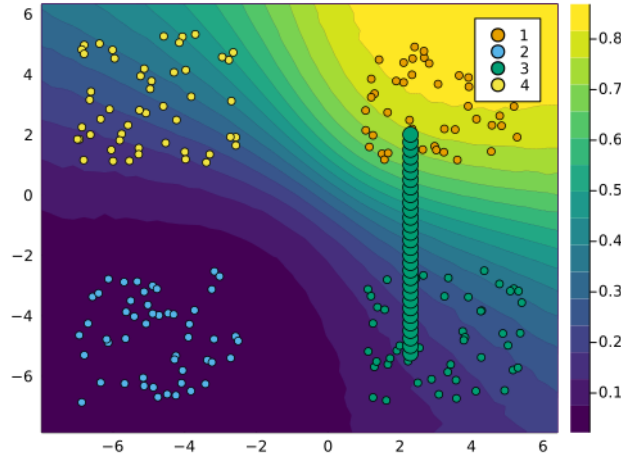


Figure 7: Counterfactual path using generic counterfactual generator for multi-class classifier with Laplace approximation.

## Empirical example

Now that we have explained the basic functionality of `CounterfactualExplanations` through a few illustrative toy examples, it is time to consider some real data. The MNIST dataset contains 60,000 training samples of handwritten digits in the form of 28x28 pixel grey-scale

images ([?]). Each image is associated with a label indicating the digit (0-9) that the image represents. The data makes for an interesting case-study of counterfactual explanations, because humans have a good idea of what realistic counterfactuals of digits look like. For example, if you were asked to pick up an eraser and turn the digit in Figure ?? into a four (4) you would know exactly what to do: just erase the top part. In [?] leverage this idea to illustrate to the reader that their methodology produces effective counterfactuals. In what follows we replicate some of their findings. You as the reader are therefore the perfect judge to evaluate the quality of the counterfactual explanations presented here.

On the model side we will use two pre-trained classifiers<sup>8</sup>: firstly, a simple multi-layer perceptron (MLP) and, secondly, a deep ensemble composed of five such MLPs following [?]. Deep ensembles are approximate Bayesian model averages that have been shown to yield high-quality estimates of predictive uncertainty for neural networks ([?], [?])). In the previous section we already created the necessary subtype and methods to make the multi-output MLP compatible with our package. The code below implements the two necessary steps for the deep ensemble.

```
using Flux: stack
Step 1) struct FittedEnsemble <: Models.FittedModel ensemble::AbstractArray end
Step 2) using Statistics logits(M::FittedEnsemble, X::AbstractArray)
= mean( stack([m(X) for m in M.ensemble],3), dims=3) probs(M::FittedEnsemble,
X::AbstractArray) = mean( stack([softmax(m(X)) for m in M.ensemble],3), dims=3)
M_ensemble = FittedEnsemble(ensemble)
```

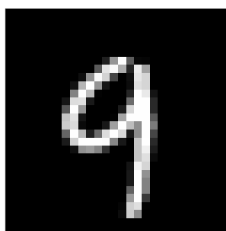


Figure 8: A handwritten nine (9) randomly drawn from the MNIST dataset.

For the counterfactual search we will use four different combinations of classifiers and generators: firstly, the generic approach for the MLP; secondly, the greedy approach for the MLP; thirdly, the generic approach for the deep ensemble; and finally, the greedy approach for the deep ensemble.

**TBD**

## Concluding remarks

---

<sup>8</sup>The pre-trained models were stored as package artifacts and loaded through helper functions.