Introduction

Machine Learning models like Deep Neural Networks have become so complex and opaque over recent years that they are generally considered Black Boxes. This lack of transparency of modern machine learning models like deep neural networks exacerbates a number of other problems typically associated with them: they tend to be unstable [**?**], encode existing biases [**?**] and learn representations that are surprising or even counter-intuitive from a human perspective [**?**]. Nonetheless, they often form the basis for data-driven decision-making systems in real-world applications.

As others have pointed out, this scenario gives rise to an undesirable principal-agent problem involving a group of principals—i.e. human stakeholders—that fail to understand the behaviour of their agent—i.e. the black-box system [**?**]. The group of principals may include programmers, product managers and other decision-makers who develop and operate the system as well as those individuals ultimately subject to the decisions made by the system. In practice, decisions made by black-box systems are typically left unchallenged since the principals cannot scrutinize them:

> "You cannot appeal to (algorithms). They do not listen. Nor do they bend." [**?**]

In light of all this, a quickly growing body of literature on Explainable Artificial Intelligence (XAI) has emerged. Counterfactual Explanations (CE) fall into this broader category. They can help human stakeholders make sense of the systems they develop, use or endure: they explain how inputs into a system need to change for it to produce different decisions. Explainability benefits internal as well as external quality assurance. Explanations that involve plausible and actionable changes can be used for Algorithmic Recourse (AR): they offer the group of principals a way to not only understand their agent's behaviour but also adjust or react to it.

The availability of open-source software to explain black-box models through counterfactuals is still limited. Most existing implementations are specific to particular methodologies. They are also exclusively built in Python and for Python models. The only existing unifying software approach, for example, is tailored to models built in the two most popular Python libraries for deep learning. The Julia ecosystem has so far lacked an open-source implementation of CE.

Through the work presented here, we aim to close that gap and thereby contribute to broader community efforts towards explainable AI. We envision this package to one day be the go-to place

for Counterfactual Explanations in Julia. Thanks to Julia's unique support for interoperability with foreign programming languages we believe that this library may ultimately also benefit the broader machine learning and data science community.

Our package provides a simple and intuitive interface to generate Counterfactual Explanations for differentiable classification models trained in Julia. It comes with detailed documentation involving various illustrative example datasets, linear and deep learning classifiers and counterfactual generators for binary and multi-class prediction tasks. A carefully designed package architecture allows for a seamless extension of the package functionality through custom generators and models. Through simple examples, we also demonstrate how to use our package to explain models that were built and trained in `Python` and `R`, although at the time of writing this feature is still experimental.

The remainder of this article is structured as follows: Section **??** presents related work on Explainable AI as well as a brief overview of the methodological framework underlying CE. Section **??** introduces the Julia package and its high-level architecture. Section **??** then presents a number of basic and advanced usage examples. In Section **??** we demonstrate how the package functionality can be customized and extended. To provide a flavour of its practical use, we use the package to explain models trained on MNIST data in Section **??**. Finally, we also discuss the current limitations of our package, as well as its future outlook in Section **??**. Section **??** concludes.

Background and related work

In this section, we first briefly introduce the broad field of Explainable Artificial Intelligence (XAI), before narrowing it down to Counterfactual Explanations. We introduce the methodological framework and finally point to existing open-source software.

Literature on Explainable AI

The field of Explainable AI 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: Arrieta et al. (2020) provide a broad overview of XAI [**?**]; Fan et al. (2020) focus on explainability in the context of deep learning [**?**]; and finally, Karimi et al. (2020) [**?**] and Verma et al. (2020) [**?**] offer detailed reviews of the literature on Counterfactual Explanations and Algorithmic Recourse.[1] Finally, Miller (2019) explicitly discusses the concept of explainability from the perspective of a social scientist [**?**].

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

[1] Readers who prefer a text-book approach may also want to consider [**?**] and [**?**]

distinction made in [**?**] useful: interpretable AI involves models that are inherently interpretable and transparent such as general additive models (GAM), decision trees and rule-based models; explainable AI may involve 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 altogether 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 medium term. Explainable AI can further be broadly divided into **global** and **local** explainability: the former is concerned with explaining the average behaviour of a model, while the latter involves explanations for individual predictions [**?**]. Tools for global explainability include partial dependence plots (PDP), which involve the computation of marginal effects through Monte Carlo, and global surrogates. A surrogate model is an interpretable model that is trained to explain the predictions of a black-box model.

Counterfactual Explanations fall into the category of local methods: they explain how individual predictions change in response to individual feature perturbations. Among the most popular alternatives to Counterfactual Explanations are local surrogate explainers including local interpretable model-agnostic explanations (LIME) and Shapley additive explanations (SHAP). Since explanations produced by LIME and SHAP typically involve simple feature importance plots, they arguably rely on reasonably interpretable features at the very least. Contrary to Counterfactual Explanations, for example, it is not obvious how to apply LIME and SHAP to visual or audio data. Nonetheless, local surrogate explainers are among the most widely used XAI tools today, potentially because they are easily understood, relatively fast and implemented in popular programming languages. Proponents of surrogate explainers also commonly mention that there is a straightforward way to assess their reliability: a surrogate model that generates predictions in line with those produced by the black-box model is said to have high **fidelity** and therefore considered reliable. As intuitive as this notion may be, it also points to an obvious shortfall of surrogate explainers: even a high-fidelity surrogate model that produces the same predictions as the black-box model 99 per cent of the time is useless and potentially misleading for every 1 out of 100 individual predictions. A recent study has shown that even experienced data scientists tend to put too much trust in explanations produced by LIME and SHAP [**?**]. Another recent work has shown that both LIME and SHAP can be easily fooled: both methods depend on random input perturbations, a property that can be abused by adverse agents to essentially whitewash strongly biased black-box models [**?**]. In a related work the same authors find that while gradient-based Counterfactual Explanations can also be manipulated, there is a straightforward way to protect against this in practice [**?**]. In the context of quality assessment, it is also worth noting that—contrary to surrogate explainers—Counterfactual Explanations always achieve full fidelity by construction: counterfactuals are searched with respect to the black-box classifier, not some proxy for it. That being said, Counterfactual Explanations should also be used with care and research around them is still in its early stages. We shall discuss this in more detail in the following.

A framework for Counterfactual Explanations

Counterfactual search happens in the feature space[2]: 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: $M : \mathcal{X} \mapsto \mathcal{Y}$ where $\mathcal{X} \subset \mathbb{R}^D$ and $\mathcal{Y} = \{0, 1\}$. Further, let $t = 1$ be the target class and let x denote the factual feature vector of some individual sample outside of the target class, so $y = M(x) = 0$. We follow this convention here, though it should be noted that the ideas presented here also carry over to multi-class problems and regression [**?**].

The counterfactual search objective originally proposed by Wachter et al. (2017) [**?**] is as follows

$$\min_{x' \in \mathcal{X}} h(x') \quad \text{s. t.} \quad M(x') = t \tag{1}$$

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

$$x' = \arg\min_{x'} \ell(M(x'), t) + \lambda h(x') \tag{2}$$

Here ℓ denotes some loss function targeting the deviation between the target label and the predicted label and λ 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 in our package. The hyperparameter λ is typically tuned through grid search or in some sense pre-determined by the nature of the problem. Conventional choices for ℓ 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.[3]

Numerous 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 the baseline paper [**?**], 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

[2] Or, in the case of Latent Space generators, some latent representation of the feature space.

[3] 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(ŷ, y)` implementation in `Flux.Losses` (used here) is more stable than the mathematically equivalent `binarycrossentropy(ŷ, y)`.

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 should respect the structural causal model underlying the data generating process ([**?**],[**?**]).

Existing software

To the best of our knowledge, the package introduced here provides the first implementation of Counterfactual Explanations in Julia and therefore represents a novel contribution to the community. As for other programming languages, we are only aware of one other unifying framework: the recently introduced Python library CARLA [**?**].[4] In addition to that, there exists open-source code for some specific approaches to Counterfactual Explanations that have been proposed in recent years. The approach-specific implementations that we have been able to find are generally well-documented, but exclusively in Python. For example, a PyTorch implementation of a greedy generator for Bayesian models proposed in [**?**] has been released. As another example, the popular InterpretML library includes an implementation of a diverse counterfactual generator proposed by [**?**].

Generally speaking, software development in the space of XAI has largely focused on various global methods and surrogate explainers: implementations of PDP, LIME and SHAP are available for both Python (e.g. `lime`, `shap`) and R (e.g. `lime`, `iml`, `shapper`, `fastshap`). In the Julia space we have only been able to identify one package that falls into the broader scope of XAI, namely `ShapML.jl` which provides a fast implementation of SHAP.[5] We also should not fail to mention the comprehensive Interpretable AI infrastructure, which focuses exclusively on interpretable models. Arguably the current availability of tools for explaining black-box models in Julia is limited, but it appears that the community is invested in changing that. The team behind `MLJ.jl`, for example, recruited contributors for a project about both Interpretable and Explainable AI in 2022.[6] With our work on Counterfactual Explanations we hope to contribute to these efforts. We think that because of its unique transparency the Julia language naturally lends itself towards building a greater degree of trust in Machine Learning and Artificial Intelligence.

[4] While we were writing this paper, the `R` package `counterfactuals` was released [**?**]. The developers seem to also envision a unifying framework, but the project appears to still be in its early stages.

[5] See here: https://github.com/nredell/ShapML.jl

[6] For details, see the Google Summer of Code 2022 project proposal: https://julialang.org/jsoc/gsoc/MLJ/#interpretable_machine_learning_in_julia.

Introducing: `CounterfactualExplanations.jl`

Figure ?? provides an overview of the package architecture. It is built around two core modules that are designed to be as extensible as possible through dispatch: 1) `Models` is concerned with making any arbitrary model compatible with the package; 2) `Generators` is used to implement arbitrary counterfactual search algorithms. The core function of the package `generate_counterfactual` uses an instance of type `<:AbstractFittedModel` produced by the `Models` module and an instance of type `<:AbstractGenerator` produced by the `Generators` module. Relating this to the methodology outlined in Section ??, the former instance corresponds to the model M, while the latter defines the rules for the counterfactual search (Equation ??).

Generators

At the time of writing the following counterfactual generators have been implemented in the package:

- ClaPROAR [?]
- CLUE [?]
- DiCE [?]
- FeatureTweak [?]
- Gravitational [?]
- Greedy [?]
- PROBE [?]
- REVISE [?]
- Wachter [?]

Models

The package currently offers native support for models built and trained in Flux as well as a small subset of models made available through MLJ [?]. While in general it is assumed that users will use this package to explain their pre-trained models, we provide a simple API call to train the following simple default models:

- Linear Classifier (Logistic Regression)
- Multi-Layer Perceptron (Deep Neural Network)

- Deep Ensemble [**?**]
- Decision Tree, Random Forest, Gradient Boosted Trees

As we demonstrate below, it is straightforward to extend the package through custom models. Support for `torch` models trained in Python is also available.

Data and Benchmarking

To allow researchers and practitioners to test and compare counterfactual generators, the package ships with pre-processed synthetic and real-world benchmark datasets from different domains. Real-world datasets include:

- Adult Census [**?**]
- California Housing [**?**]
- CIFAIR10 [**?**]
- German Credit [**?**]
- Give Me Some Credit [**?**]
- MNIST [**?**] and Fashion MNIST [**?**]
- UCI defaultCredit [**?**]

Plotting

The package also extends common `Plots.jl` methods to facilitate the visualization of results. Calling the generic `plot()` method on an instance of type `<:CounterfactualExplanation`, for example, generates a plot visualizing the entire counterfactual path in the feature space[7]. We will see several examples of this below.

Basic Usage

In the following, we begin our exploration of the package functionality with a simple example. We then turn to a more advanced usage example and show how users can impose mutability constraints on features.

[7] For multi-dimensional input data, standard dimensionality reduction techniques are used to compress the data. In this case, the classifier's decision boundary is approximated through a Nearest Neighbour model. This is still somewhat experimental and will be improved in the future.

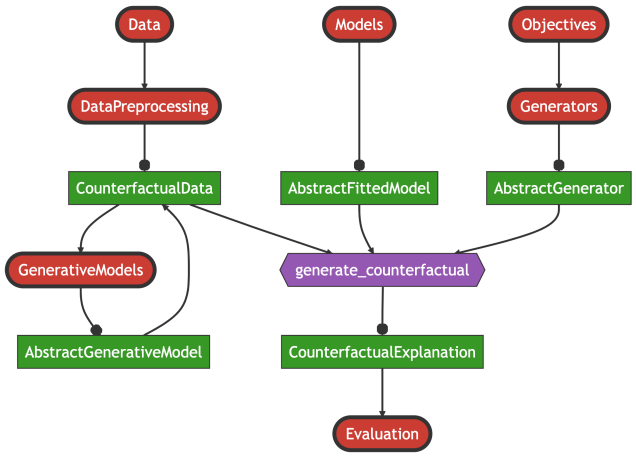

Figure 1: High-level schematic overview of package architecture. Modules are shown in red, structs in green and functions in purple.

A Simple Generic Generator

Code **??** below provides a complete example demonstrating how the framework presented in Section **??** can be implemented in Julia with our package. Using a synthetic data set with linearly separable samples we first 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) <: AbstractFittedModel`. 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 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) specify the other class as our `target` label ($t = 1$) in line **??**; 2) choose a random sample `x` from the non-target class in line **??**; 3) define the counterfactual generator in line **??**; and finally run the counterfactual search in line **??**. Gradient-based generators like the `GenericGenerator` take several optional arguments that can be used to define the objective function and the desired optimiser. This will be discussed in some more detail when looking at the advanced usage example below.

[language=Julia, escapechar=@, numbers=left, label=lst:simple, caption=Standard workflow for generating counterfactuals.] Data and Classifier: counterfactual$_data$ = $load_linearly_separable()M = fit_model(counterfactual_data, : Linear)$

Factual and Target: yhat = predict$_label(M, counterfactual_data)target = 2targetlabel@@candidates = findall(vec(yhat).! = target)chosen = rand(candidates)x = select_factual(counterfactual_data, chosen)@@$

Counterfactual search: generator = GenericGenerator() @@ ce = generate$_counterfactual(x, target, counterfactu$

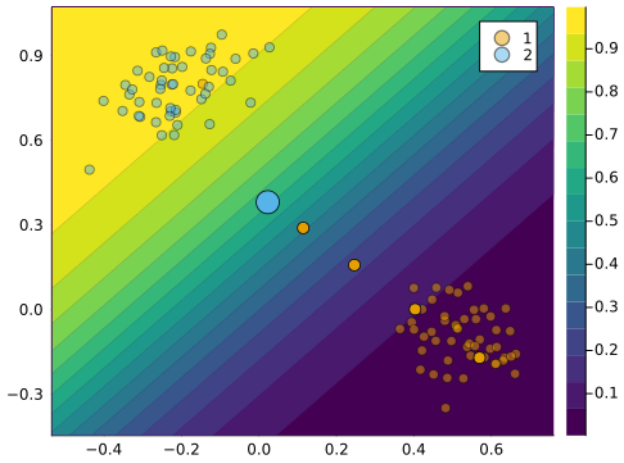

Figure 2: Counterfactual path using generic counterfactual generator for conventional binary classifier.

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 - that is regions with high epistemic uncertainty - it is easy to think of scenarios that involve valid but implausible counterfactuals. Similarly, any degree of overfitting can be expected to result in more ambiguous Counterfactual Explanations, since it reduces the classifier's sensitivity to regions with high aleatoric uncertainty. Consider, for example, the scenario illustrated in Figure **??**, which involves the same logistic classifier, but a massively overfitted version of it. In this case, generic search may yield an entirely ambiguous counterfactual near the decision boundary (purple marker) or an implausible counterfactual that has moved well into the target domain, but remains far away from all other samples (red marker).

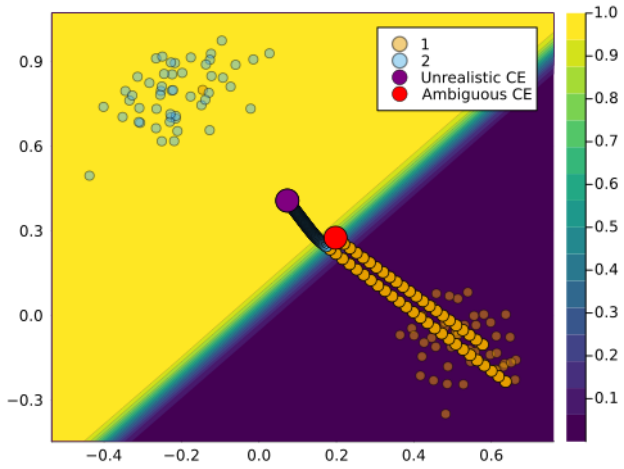

Figure 3: Implausible and ambiguous counterfactuals that may be produced by generic counterfactual search for an overfitted conventional binary classifier.

More Advanced Generators

The more advanced generators currently implemented in `CounterfactualExplanations.jl` are designed to address the desiderata mentioned above, in particular plausibility. In this context, 'plausibility' is defined in the sense that counterfactuals ought to be generated by the same data-generating process (DGP) that generates the actual data points. To this end, **Latent Space** generators like REVISE [**?**] use a separate generative model to learn the DGP. We refer to them as Latent Space generators, because they search counterfactuals in the latent embedding learned by the generative model.[8]. The **Greedy** approach [**?**] instead relies on minimizing predictive uncertainty in order to generate plausible counterfactuals. **CLUE** [**?**] can be thought of as a combination of these two ideas. The other generator currently implemented, **DiCE** [**?**], generates multiple counterfactuals at once that are as diverse as possible. This strategy is based on the intuition that a wide variety of diverse explanations may be suitable depending on the practical context.

Code **??** below shows a more advanced usage example involving the DiCE generator. Once again it is worth dwelling on this for a moment. The `DiCEGenerator` is instantiated in line **??** with a custom Flux optimizer and decision threshold specified in lines **??** and **??**, respectively[9]. The main API call to generate counterfactuals is the same as before, but note that in line **??** we have specified an optional key argument that determines how many counterfactuals are generated. For the DiCE generator it naturally makes sense to generate multiple counterfactuals,

[8]Currently our implementation relies on a Variational Autoencoder (VAE)

[9]Note that all differentiable generators except the `GreedyGenerator` work with Flux optimizers and accept them as an optional key argument.

but note that this is in principle also possible for all other generators.[10] Figure **??** shows the resulting output. It was generated by calling the generic `plot` method directly on the object returned by `generate_counterfactual`.

[language=Julia, escapechar=@, numbers=left, label=lst:binary-advanced, caption=Using the DiCE generator.] Generator: generator = DiCEGenerator(@@ opt = Flux.Optimise.Descent(0.01), @@ decision$_t hreshold$ = 0.7, @@ = [0.1, 5])$Counterfactualsearch$: $counterfactuals$ = $generate_counterfactual(x, target, counterfactual_data, M, generator; num_counterfactuals$ = 5@@)

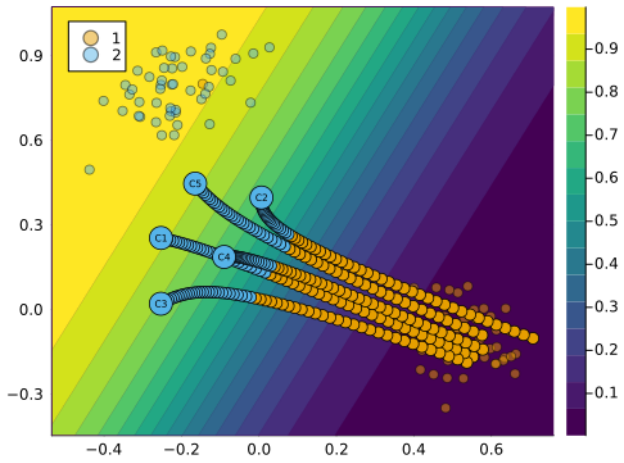

Figure 4: Counterfactual path using the DiCE generator.

Mutability Constraints

In practice, features usually cannot be perturbed arbitrarily. Suppose, for example, that one of the features used by a bank to predict the creditworthiness of its clients is *gender*. If a counterfactual explanation for the prediction model indicates that female clients should change their gender to improve their creditworthiness, then this is an interesting insight (it reveals gender bias), but it is not usually an actionable transformation in practice. In such cases, we may want to constrain the mutability of features to ensure actionable and plausible recourse. To illustrate how this can be implemented in `CounterfactualExplanations.jl` we will look at the linearly separable toy dataset again.

Mutability of features can be defined in terms of four different options: 1) the feature is mutable in both directions, 2) the feature can only increase (e.g. *age*), 3) the feature can only decrease (e.g. *time left* until your next deadline) and 4) the feature is not mutable (e.g. *skin colour*,

[10] By default counterfactuals are initialized by adding a small, random perturbation, as this improves adversarial robustness [**?**]. Therefore, generating multiple counterfactuals will yield multiple distinct outcomes even without an explicit diversity constraint.

ethnicity, ...). To specify which category a feature belongs to, users can pass a vector of symbols containing the mutability constraints at the pre-processing stage. For each feature one can choose from these four options: `:both` (mutable in both directions), `:increase` (only up), `:decrease` (only down) and `:none` (immutable). By default, `nothing` is passed to that keyword argument and it is assumed that all features are mutable in both directions.[11]

Below we impose that the second feature is immutable.

[language=Julia, escapechar=@, numbers=left, label=lst:mutability, caption=Adding mutability constraints.] $\text{counterfactual}_data.mutability = [:both, :none]$

The resulting counterfactual path is shown in Figure **??** below. Since only the first feature can be perturbed, the sample can only move along the horizontal axis.

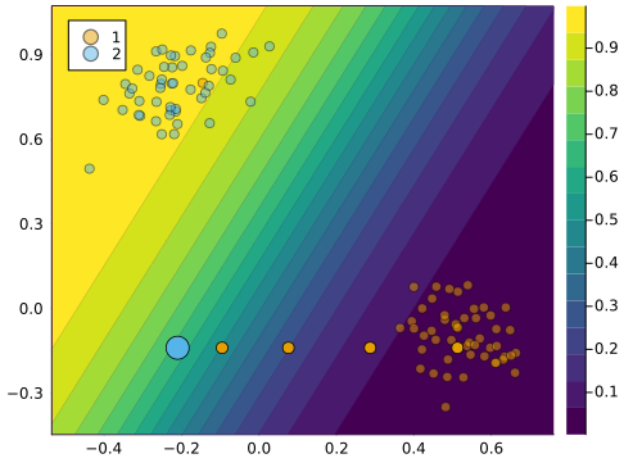

Figure 5: Counterfactual path with immutable feature.

Customization and Extensibility

One of our priorities has been to make `CounterfactualExplanations` customizable and extensible. 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.

Adding Custom Models

At the high level, only two steps are necessary to make any supervised learning model compatible with our package:

[11] Mutability constraints are currently not yet implemented for Latent Space generators.

Subtyping: the model needs to be declared as a subtype of `AbstractFittedModel`.

Dispatch: the functions `logits` and `probs` need to be extended through custom methods for the model in question.

To demonstrate how this can be done in practice, we will reiterate here how native support for `Flux.jl` ([**?**]) deep learning models was enabled.[12] Once again we use synthetic data for an illustrative example. Code **??** below builds a simple model architecture that can be used for a multi-class prediction task. Note how outputs from the final layer are not passed through a softmax activation function, since the counterfactual loss is evaluated with respect to logits as we discussed earlier. The model is trained with dropout.

[language=Julia, escapechar=@, numbers=left, label=lst:nn, caption=A simple neural network model.] $n_hidden = 32 output_dim = length(unique(y)) input_dim = 2 model = Chain(Dense(input_dim, n_hidden, activation), Dropout(0.1), Dense(n_hidden, output_dim))$

Code **??** below implements the two steps that were necessary to make Flux models compatible with the package. In line **??** we declare our new struct as a subtype of `AbstractDifferentiableModel`, which itself is an abstract subtype of `AbstractFittedModel`.[13] Computing logits amounts to just calling the model on inputs. Predicted probabilities for labels can then be computed by passing predicted logits through the softmax function.

[language=Julia, escapechar=@, numbers=left, label=lst:mymodel, caption=A wrapper for Flux models.] Step 1) struct MyFluxModel <: AbstractDifferentiableModel @@ model::Any likelihood::Symbol @@ end

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

The API call for actually generating counterfactuals for our new model is the same as before. Figure **??** shows the resulting counterfactual path for a randomly chosen sample. In this case, the contour shows the predicted probability that the input is in the target class ($t = 2$). Generic search yields a valid, plausible and largely unambiguous counterfactual.

Adding Custom Generators

To illustrate how custom generators can be implemented we will consider a simple example of a generator that extends the functionality of our `GenericGenerator`. We have noted elsewhere that the effectiveness of Counterfactual Explanations depends to some degree on the quality of

[12] Flux models are now natively supported by our package and can be instantiated by calling `FluxModel()`.

[13] Note that in line **??** we also provide a field determining the likelihood. This is optional and only used internally to determine which loss function to use in the counterfactual search. If this field is not provided to the model, the loss function needs to be explicitly supplied to the generator.

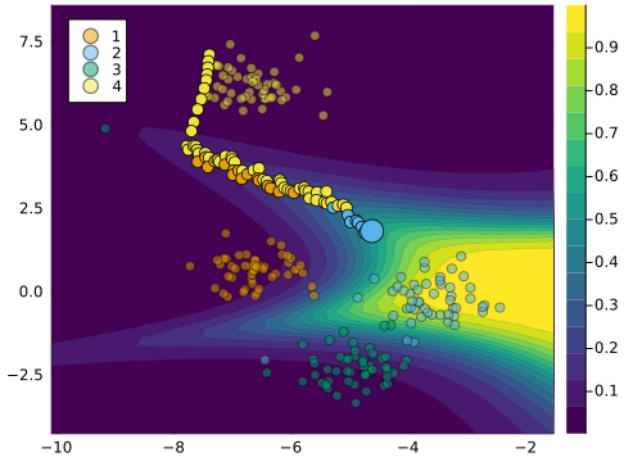

Figure 6: Counterfactual path using generic counterfactual generator for multi-class classifier.

the fitted model. Another, perhaps trivial, thing to note is that Counterfactual Explanations are not unique: there are potentially many valid counterfactual paths. One idea building on these two observations might be to introduce some form of regularization in the counterfactual search. For example, we could use dropout to randomly switch features on and off in each iteration. Without dwelling further on the merit of this idea, we will now briefly show how this can be implemented.

A Generator with Dropout

Code **??** below implements two important steps: 1) create an abstract subtype of the `AbstractGradientBasedGenerator` and 2) create a constructor similar to the `GenericConstructor`, but with one additional field for the probability of dropout.

[language=Julia, escapechar=@, numbers=left, label=lst:dropout, caption=Building a custom generator with dropout.] Abstract suptype: abstract type AbstractDropoutGenerator <: AbstractGradientBasedGenerator end Constructor: struct DropoutGenerator <: AbstractDropoutGenerator loss::Symbol loss function complexity::Function complexity function @λ@::AbstractFloat strength of penalty decision$_t$*hreshold* :: $UnionNothing, AbstractFloat$*opt* :: *Anyoptimizer*@$\tau$@::AbstractFloat tolerance for convergence p$_d$*ropout* :: *AbstractFloatdropoutrateend*

Next, in listing **??** we define how feature perturbations are generated for our custom dropout generator: in particular, we extend the relevant function through a method that implements the dropout logic.

[language=Julia, escapechar=@, numbers=left, label=lst:generate, caption=Generating feature perturbations with dropout.] using CounterfactualExplanations.Generators

function Generators.generate$_p erturbations(generator :: AbstractDropoutGenerator, ce ::$
$Couterfactual Explanation)$@s′@ = deepcopy(ce.@$s′$@) new$_{@}$s′@ = Generators.propose$_s tate(generator, ce)$@$\Delta s′$
= new$_{@}$s′@ - @$s′$@ gradient step Dropout: set$_t o_z ero$ = $sample(1 : length($@$\Delta s′$@$)$,
Int(round(generator.p$_d ropout * length($@$\Delta s′$@$)))$, replace=false) @$\Delta s′$@[set$_t o_z ero$]. =
$0return$@$\Delta s′$@ end

Finally, we proceed to generate counterfactuals in the same way we always do. The resulting counterfactual path is shown in Figure **??**.

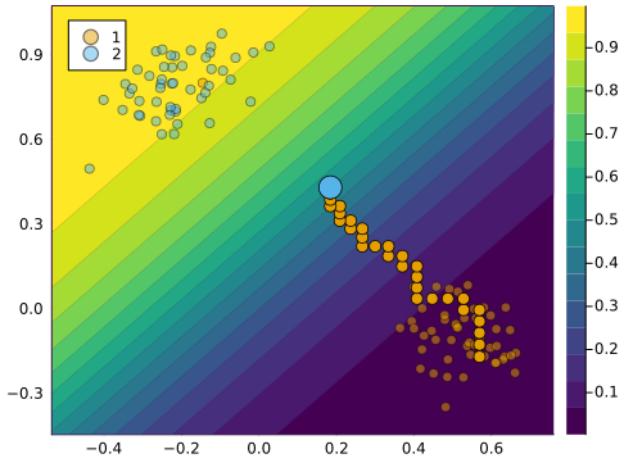

Figure 7: Counterfactual path for a generator with dropout.

A Real-World Examples

Now that we have explained the basic functionality of `CounterfactualExplanations.jl` through some synthetic examples, it is time to work through examples involving real data.

Give Me Some Credit

The **Give Me Some Credit** dataset is one of the tabular real-world datasets that ship with the package [**?**]. It can be used to train a binary classifier to predict whether a borrower is likely to experience financial difficulties in the next two years. In particular, we have an output variable $y \in \{0 = \texttt{no stress}, 1 = \texttt{stress}\}$ and a feature matrix X that includes socio-demographic variables like `age` and `income` X. A retail bank might use such a classifier to determine if potential borrowers should receive credit or not.

For the classification task we use a Multi-Layer Perceptron with dropout regularization.

Using the Gravitational generator [**?**] we will generate counterfactuals for ten randomly chosen individuals that would be denied credit based on our pre-trained model. Concerning the mutability of features, we only impose that the `age` cannot be decreased.

Figure **??** shows the resulting counterfactuals proposed by Wachter in the two-dimensional feature space spanned by the `age` and `income` variables. An increase in income and age is recommended for the majority of individuals, which seems plausible: both age and income are typically positively related to creditworthiness.

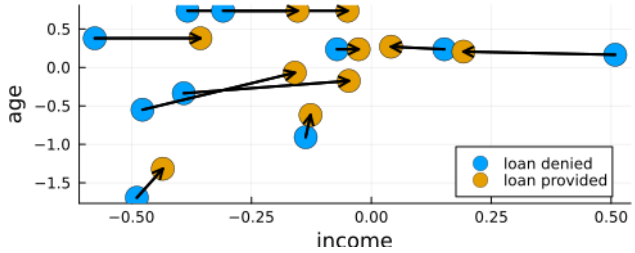

Figure 8: Give Me Some Credit: counterfactuals for would-be borrowers proposed by the Gravitational Generator.

MNIST

For our second example, we will look at image 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 plausible counterfactuals of digits look like. For example, if you were asked to pick up an eraser and turn the digit in the left panel of Figure **??** into a four (4) you would know exactly what to do: just erase the top part. Schut et al. (2021) [**?**] leverage this idea to illustrate to the reader that their methodology produces plausible 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 below.

On the model side, we will use a simple multi-layer perceptron (MLP). Code **??** loads the data and the pre-trained MLP. It also loads two pre-trained Variational Auto-Encoders, which will be used by our counterfactual generator of choice for this task: REVISE.

[language=Julia, escapechar=@, numbers=left, label=lst:mnist-setup, caption=Loading pre-trained models and data for MNIST.] $\text{counterfactual}_data = load_mnist()X, y = unpack_data(counterfactual_data)input_dim, n_obs = size(counterfactual_data.X)M = load_mnist_mlp()vae = load_mnist_vae()vae_weak = load_mnist_vae(; strong = false)$

The proposed counterfactuals are shown in Figure **??**. In the case in which REVISE has access to an expressive VAE (centre), the result looks convincing: the perturbed image does look like

it represents a four (4). In terms of explainability, we may conclude that removing the top part of the handwritten nine (9) leads the black-box model to predict that the perturbed image represents a four (4). We should note, however, that the quality of counterfactuals produced by REVISE hinges on the performance of the underlying generative model, as demonstrated by the result on the right. In this case, REVISE uses a weak VAE and the resulting counterfactual is invalid. In light of this, we recommend using Latent Space search with care.

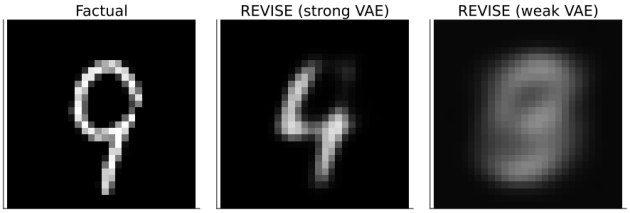

Figure 9: Counterfactual explanations for MNIST using a Latent Space generator: turning a nine (9) into a four (4).

Discussion and Outlook

We believe that this package in its current form offers a valuable contribution to ongoing efforts towards XAI in Julia. That being said, there is significant scope for future developments, which we briefly outline in this final section.

Candidate models and generators

The package supports various models and generators either natively or through minimal augmentation. In future work, we would like to prioritize the addition of further predictive models and generators. Concerning the former, it would be useful to add native support for any supervised models built in `MLJ.jl`, an extensive Machine Learning framework for Julia [**?**]. This may also involve adding support for regression models as well as additional non-differentiable models. In terms of counterfactual generators, there is a list of recent methodologies that we would like to implement including MINT [**?**] and ROAR [**?**].

Additional datasets

For benchmarking and testing purposes it will be crucial to add more datasets to our library. We have so far prioritized tabular datasets that have typically been used in the literature on counterfactual explanations including Adult, Give Me Some Credit and German Credit [**?**]. There is scope for adding data sources that have so far not been explored much in this context including additional image datasets as well as audio, natural language and time-series data.

Adding Foreign Language Support

The Julia language offers unique support for programming language interoperability. For example, calling R or Python is made remarkably easy through auxiliary packages like `RCall.jl`, `PythonCall.jl` and `PyCall.jl`, respectively. Early experimentation has shown that this functionality can be leveraged to make `CounterfactualExplanations.jl` compatible with models that were developed in foreign programming languages. At the time of writing, we have added native support for explaining torch models trained in R and Python.

Concluding remarks

The goal of this paper was to illustrate the need for explainability in machine learning and the promise of Counterfactual Explanations in this context. To this end, we introduced `CounterfactualExplanation.jl`: a package for generating Counterfactual Explanations and Algorithmic Recourse in Julia. Through various synthetic and real-world examples, we have demonstrated the basic usage of the package and shown how it can be easily customized and extended. We envision this package to one day constitute the go-to place for explaining arbitrary predictive models through a diverse suite of counterfactual generators. As a major next step, we would therefore like to interface our library with the popular `MLJ.jl` package for machine learning in Julia. The package can also serve as a testing ground for new and existing methodological approaches to Counterfactual Explanations and Algorithmic Recourse. We invite the Julia community to contribute to these goals through usage, open challenge and active development.

Acknowledgements

We are immensely grateful to the group of TU Delft students who contributed huge improvements to this package as part of a school project in 2023: Rauno Arike, Simon Kasdorp, Lauri Keskküll, Mariusz Kicior, Vincent Pikand. We also want to thank the broader Julia community for being welcoming and open and for supporting research contributions like this one. Some of the members of TU Delft were partially funded by ICAI AI for Fintech Research, an ING—TU Delft collaboration.