Introduction

Machine learning models like deep neural networks have become so complex, opaque and underspecified in the data that they are generally considered as black boxes. This lack of transparency exacerbates a number of other problems typically associated with these models: they tend to be instable ([**?**]), 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.

As others have pointed out, this scenario gives rise to and 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 has emerged. Counterfactual explanations (CE) and algorithmic recourse (AR) fall into this broader category. Counterfactual explanations 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 realistic and actionable changes can be used for the purpose of 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 for the purpose of explaining black-box models through counterfactuals is still limited. Most existing implmentations 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 counterfactual explanations.

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 be a go-to place for counterfactual explanations in Julia. Thanks to its applicability to systems built in other programming languages we believe that this library may ultimately also benefit the broader community engaged in data-driven decision making.

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 seamless extension of the package functionality through custom generators and models. By leveraging Julia's unique support for language interoperability, we also demonstrate how to easily use our package to explain models that were built and trained in `Python` and `R`.

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. To demonstrate its practical use, Section **??** involves an application to MNIST data. Finally, we also discuss current limitations of our package, as well as its future outlook in Section **??**. Section **??** concludes.

Background and 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, [**?**] and [**?**] offer detailed reviews of the literature on counterfactual explanations and algorithmic recourse.[1] Finally, [**?**] explicitly takes the social sciences take on explanation into account.

The first broad distinction we want to make here is between **interpretable** and **explainable** AI. These terms are often used interchangeably, but this can cause confusion. We find the 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 [[**?**]] 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: the former is concerned with explaining the average behavior of a model, while the latter involves explanations for individual predictions [**?**]. Tools for global explainability include partial dependence plots (PDP), which involves 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.

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

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 at the very least on reasonably interpretable features. 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 straight-forward 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 percent of the time is useless and potentially misleading for every 1 out 100 individual predictions. In fact, 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 straight-forward 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 at its early stages. We shall discuss this in more detail in Section **??**.

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 introduce Python library CARLA ([**?**]). 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.[2]

[2]See here: https://github.com/oscarkey/explanations-by-minimizing-uncertainty

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.[3] 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, is currently recruiting contributors for a project about both interpretable and explainable AI.[4] 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.

Counterfactual explanations

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: $M : \mathcal{X} \mapsto \mathcal{Y}$ 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 ([**?**]).

A framework for Counterfactual Explanations

The counterfactual search objective originally proposed by [**?**] is as follows

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

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

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

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

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

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. 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.[5]

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 realistic 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 ([**?**],[**?**]).

The `CounterfactualExplanations.jl` Package

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.[6] The core function of the package `generate_counterfactual` uses an instance of type `T <: AbstractFittedModel` produced by the `Models` module and an instance of type `T <: AbstractGenerator` produced by the

[5] 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)`.

[6] We have made an effort to keep the code base a flexible and extensible as possible, but cannot guarantee at this point that really any counterfactual generator can be implemented without further adaptation.

`Generators` module. Relating this back 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 **??**). At the time of writing the following counterfactual generators have been implemented in the package:

- Generic [**?**]
- Greedy [**?**]
- DiCE [**?**]
- Latent Space Search as in REVISE [**?**] and CLUE [**?**]

The package currently offers native support for models built in the following libraries: Flux; Torch for R; PyTorch. In the following section we will present usage examples and explain how the package can be extended through custom generators and models.

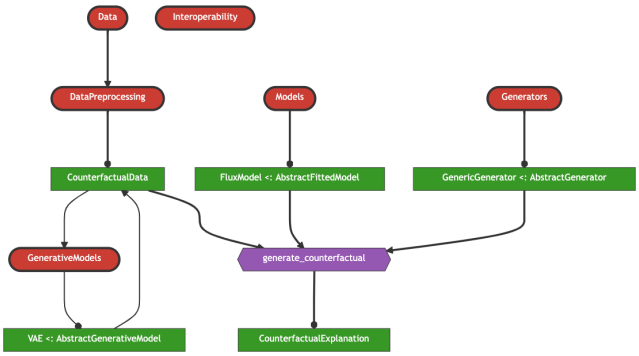

Figure 1: Overview of package architecture. Modules are shown in red, structs in green and functions in blue.

`CounterfactualExplanations.jl`: Basic Usage

A Simple Generic Generator

The 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 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) <: 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) choose a random sample `x`; 2) compute its factual label `y` as predicted by the model ($M(x) = 0$); and 3) specify the other class as our `target` label ($t = 1$) along with a desired level of `confidence` in the final prediction $M(x\prime) = t$.

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 ℓ to be used. Since we are dealing with a binary classification problem, logit binary cross-entropy is an appropriate choice.[7] The fifth and last field can be used to define mutability constraints for the features.

[language=Julia, escapechar=@, numbers=left] Data: using CounterfactualExplanations, Random Random.seed!(1234) N = 100 number of data points xs, ys = toy$_d ata_l inear(N) X = hcat(xs...) counterfactual_d ata = CounterfactualData(X, ys\prime)$

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

Setup: x = select$_f actual(counterfactual_d ata, rand(1 : length(xs)))y = round(probs(M, x)[1])target = ifelse(y == 1.0, 0.0, 1.0)$

Counterfactual search: generator = GenericGenerator() counterfactual = generate$_c ounterfactual(x, target, counte$

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 unrealistic counterfactuals. Similarly, any degree of overfitting can be expected to result in more ambiguous counterfactual explanations, since it reduces the classifiers sensitivity to regions with high aleatoric uncertainty. Consider, for example, the

[7] 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.

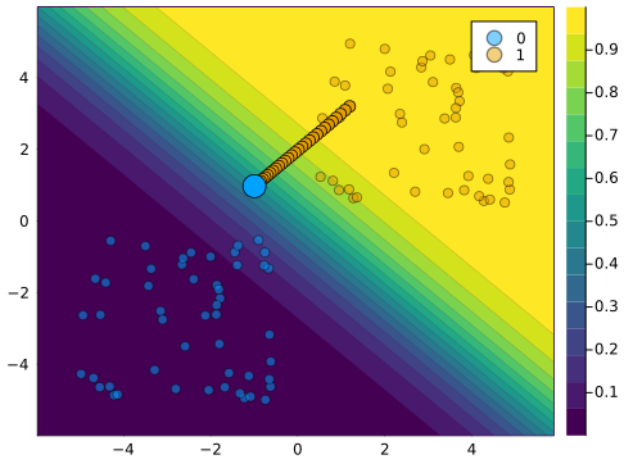

Figure 2: Counterfactual path using generic counterfactual generator for conventional binary classifier.

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 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).

More Advanced Generators

The more advanced generators currently implemented in `CounterfactualExplanations.jl` are designed to generate more realistic counterfactuals. In this context, 'realistic' 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 minimizing predictive uncertainty in order to generate realistic 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. In line **??** we instantiate a Flux optimizer that will determine how exactly the counterfactual search objective is optimized. That optimizer is then

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

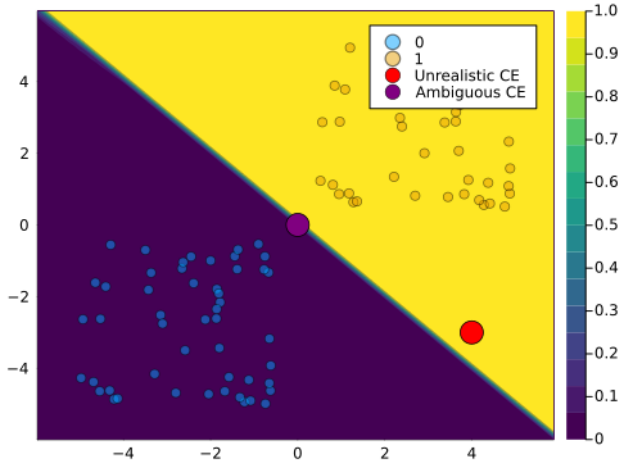

Figure 3: Unrealistic and ambiguous counterfactuals that may be produced by generic counterfactual search for an overfitted conventional binary classifier.

fed to the `DiCEGenerator` in line **??**.[9]. The main API call to actually 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, but note that this is in principal 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=] Counterfactual search: opt = Flux.Optimise.Descent(1.0) @@ generator = DiCEGenerator(;opt = opt) @@ counterfactuals = generate$_counterfactual(x, target, counterfactual_data, M, generator; num_counterfactuals$ 5@@)$Plottingplt = plot(counterfactuals)$

Adding Custom Models

One of our priorities has been to make `CounterfactualExplanations` extensible 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. At the high level, only two

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

[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.

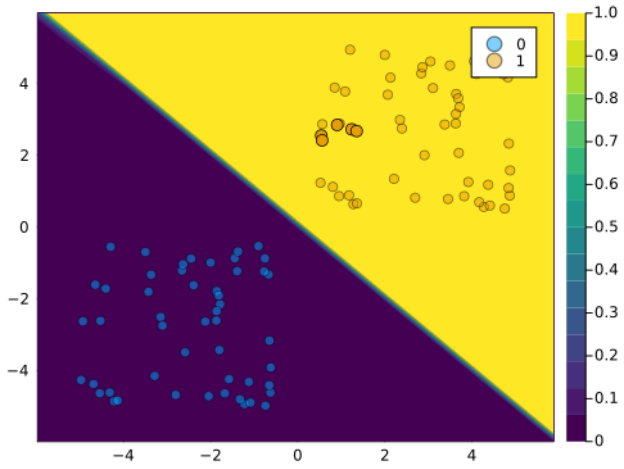

Figure 4: Counterfactual path using the DiCE generator.

steps are necessary to make any supervised learning model compatible with our package[11]:

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 counterfactual loss is evaluated with respect to logits as we discussed earlier. The model is trained with dropout for ten training epochs.

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

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 than be computed by passing predicted logits through the softmax function.

[11] 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.

[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.

[language=Julia, escapechar=@, numbers=left, label=lst:mymodel, caption=] 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 ($t =$ '*juliatarget*'). Generic search yields a valid, realistic and unambiguous counterfactual.

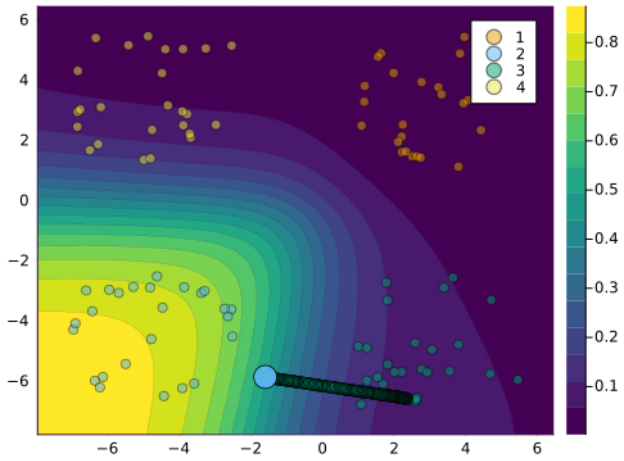

Figure 5: Counterfactual path using generic counterfactual generator for multi-class classifier.

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 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=] 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* :: $UnionNothing, AbstractFloatprobabilitythresholdopt$ :: $Anyoptimizer$@τ@::AbstractFloat tolerance for convergence p$_d$*ropout* :: $AbstractFloatdropoutrateend$

Instantiate: using LinearAlgebra generator = DropoutGenerator(:logitbinarycrossentropy, norm, 0.1, 0.5, Flux.Optimise.Descent(), 0.1, 0.5)

Next, in code **??** 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=] using CounterfactualExplanations.Generators import Generators: generate$_p$*erturbations*, $propose_stateusingStatsBasefunctionge$ $AbstractDropoutGenerator, counterfactual_state :: State)$@s′@ = deepcopy(counterfactual$_s$*tate*.@s′@) new$_{@}$s′@ = propose$_s$*tate*$(generator, counterfactual_state)$@$\Delta s'$@ = 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 **??**.

Feature constraints

In practice, features usually cannot be perturbed arbitrarily. Suppose, for example, that one of the features used by a bank to predict the credit worthiness of its clients is *gender*. If a counterfactual explanation for the prediction model indicates that female clients should change their gender to improve their credit worthiness, 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 realistic recourse. To illustrate how this can be implemented in `CounterfactualExplanations.jl` we will look at the linearly separable toy dataset again.

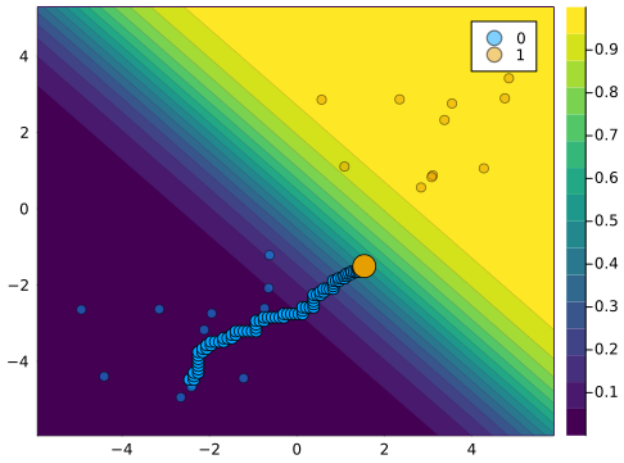

Figure 6: Counterfactual path for a generator with dropout.

Mutability

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*, *ethnicity*, ...). To specify which category a feature belongs to, you can pass a vector of symbols containing the mutability constraints at the pre-processing stage. For each feature you 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.

Below we impose that the second feature is immutable. 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.

[language=Julia, escapechar=@, numbers=left] counterfactual$_data = CounterfactualData(X, ys'; mutability = [: both, : none])$

Language interoperability

The Julia language offers unique support for programming language interoperability. For example, calling R or Python is made remarkably easy through `RCall.jl` and `PyCall.jl`, respectively. This functionality can be leveraged to use `CounterfactualExplanations.jl` to generate explanations for models that were developed in other programming languages. While at the time of writing we have not yet implemented out-of-the-box support for foreign

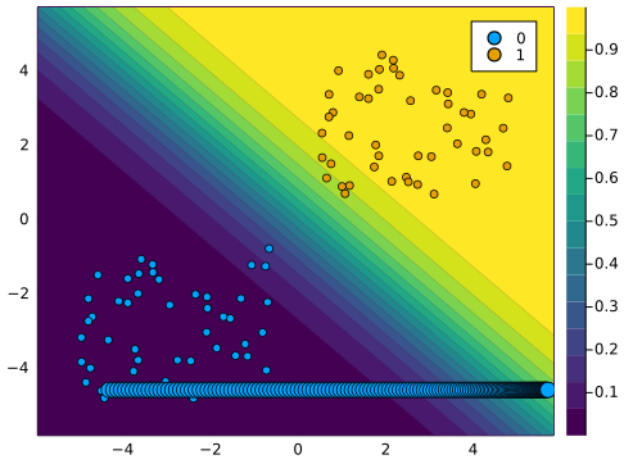

Figure 7: Counterfactual path with immutable feature.

programming languages, the following example involving a `torch` neural network trained in `R` demonstrates how versatile our package is.[14]

Explaining a model trained in R

We have trained a simple MLP for binary classification task involving a synthetic data set using the R library `torch`. Inside the R working environment the fitted `torch` model is stored as an object called `model`. That R object can be accessed from Julia using `RCall.jl` by simply calling `R"model"`. As in Section `??` and Section `??` the first thing necessary to make this model compatible with our package is to declare it as a subtype of `Model.AbstractFittedModel`. As always we also need to extend the `logits` and `probs` functions to make the model compatible with `CounterfactualExplanations.jl`. The code below shows how this can be done. Logits are returned by the `torch` model and copied from R into the Julia environment. Probabilities are then computed in Julia by passing the logits through the sigmoid function.

[language=Julia] Step 1) struct TorchNetwork <: Models.AbstractFittedModel nn::Any end

Step 2) function logits(M::TorchNetwork, X::AbstractArray) nn = M.nn y = rcopy(R"as$_a$rray(nn(torch$_t$ensor(t(X) y = isa(y, AbstractArray) ? y : [y] return y' end function probs(M::TorchNetwork, X::AbstractArray) return .(logits(M, X)) end M = TorchNetwork(R"model")

Next, we need to do a tiny bit of work on the `AbstractGenerator` side. The default methods underlying the counterfactual generators are desiged to work with models that have gradient access through `Zygote.jl`, one of Julia's main autodifferentiation packages. Of course, `Zygote.jl` cannot access the gradients of our `torch` model, so we need to adapt the code

[14] The corresponding example involving `PyTorch` is analagous and therefore not included here. You may find it here: https://www.paltmeyer.com/CounterfactualExplanations.jl/dev/tutorials/interop/

slightly. Fortunately, it turns out that all we need to do is extend the function that computes the gradient with respect to the loss function for the generic counterfactual search. In particular, we will extend the function by a method that is specific to the `TorchNetwork` type we defined above. The code below implements this: our new method calls R in order to use `torch`'s autodifferentiation functionality for computing the gradient. The method itself is then used by the core function `generate_counterfactuals` introduced earlier. From here on onwards the `CounterfactualExplanations.jl` functionality can be used as always. Figure **??** shows the counterfactual path for a randomly chosen sample with respect to the MLP trained in R.

[language=Julia, escapechar=@, numbers=left] import CounterfactualExplanations.Generators: @$\partial\ell$@ using LinearAlgebra

Countefactual loss: function @$\partial\ell$@(generator::AbstractGradientBasedGenerator, counterfactual$_state$ :: $CounterfactualState$)$M = counterfactual_state.Mnn = M.nn$@x$\prime$@ = counterfactual$_state$.@x$\prime$@ t = counterfactual$_state.target_encodedR$"""$x < -torch_tensor$(@$x\prime$@, requires$_grad = TRUE$)$output < -$nn(x) loss$_fun < -nnf_binary_cross_entropy_with_logits obj_loss < -loss_fun$($output$,t) obj$_loss$backward() """ grad = rcopy(R"as$_array$($x$grad)") return grad end

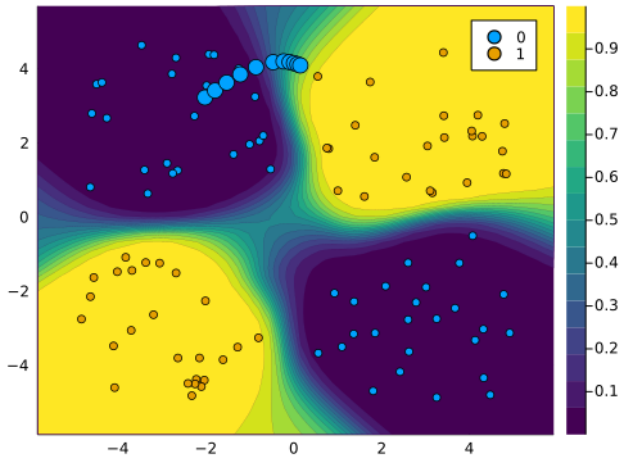

Figure 8: Counterfactual path using the generic counterfactual generator for a model trained in R.

Application to MNIST

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 methodolgy 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[15]: 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 esimtates of predictve 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.

[language=Julia, escapechar=@, numbers=left] using Flux: stack Step 1) struct FittedEnsemble <: Models.AbstractFittedModel 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) $M_ensemble = FittedEnsemble(ensemble)$

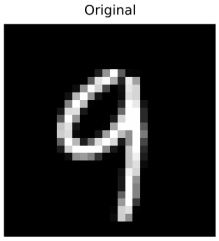

Figure 9: A handwritten nine (9) randomly drawn from the MNIST dataset.

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.

We begin by turning the nine in Figure **??** into a four. Figure **??** shows the resulting counterfactuals. In every case the desired label switch is in fact achieved, but arguably from a human perspective only the counterfactuals for the deep ensemble look like a four. The generic generator produces mild perturbations in regions that seem irrelevant from a human perspective, but nonetheless yields a coutnerfactual that can pass as a four. The greedy approach ([**?**]) clearly targets pixels at the top of the handwritten nine and yields the best result overall. For

[15]The pre-trained models were stored as package artifacts and loaded through helper functions.

the non-bayesian MLP, both the generic and the greedy approach generate counterfactuals that look much like adversarial examples: they perturb pixels in seemingly random regions on the image. Figure **??** shows another example. This time the goal is to turn a randomly chosen three (3) into an eight (8). Onve again the outcomes for the deep ensemble look more realistic, but overall the generated counterfactuals look less effective than those in Figure **??**. The results could likely be improved by using adversarial training for the classifiers as recommended in [**?**].

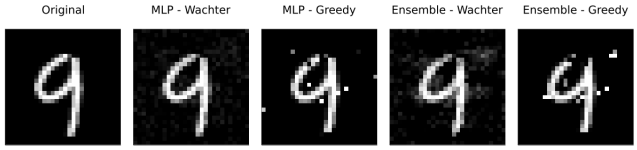

Figure 10: Counterfactual explanations for MNIST: turning a nine (9) into a four (4).

Overall, the examples in this section demonstrate two points that we have already made earlier: firstly, the findings in [**?**] can indeed complement other existing approaches to counterfactual generation; and secondly, the quality of the classifier is clearly reflected in the quality of the counterfactual explanations. In other words, we cannot generate effective counterfactual explanations for a poorly trained model. That is actually desirable: if a model bases its predictions on representations that are not intuitive to a human, we would like that to be evident from the counterfactual explanation. From that perspective, counterfactual explanations can help us to not only understand a black-box model, but potentially also guide us in improving it.

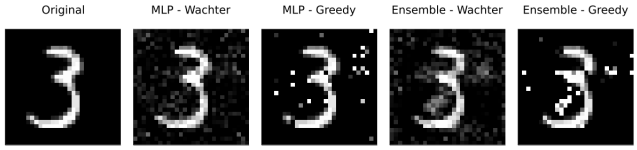

Figure 11: Counterfactual explanations for MNIST: turning a three (3) into an eight (8).

Discussiong and Outlook

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

Candidate models and generators

At the time of writing the package supports a handful of default models and generators either natively or through minimal augmentation. In future work we would like to prioritize the addition of further predictive models and especially generators. With respect to the former, it would be useful to add native support for any arbitrary Flux model, as well as predictive models built in other popular libraries including `MLJ.jl`, `ScikitLearn.jl`, `GLM.jl` and `Turing.jl`. This may also involve adding support for regression models as well as non-differentiable models. In terms of counterfactual generators, we are particularly interested in having the following approaches added: CLUE [**?**], DiCE [**?**], MINT [**?**], REVISE [**?**] and ROAR [**?**]. Through its composable nature, our package may also allow for combining different approaches.

Candidate datasets

For benchmarking and testing purposes it will be crucial to add more datasets to our library. We would like to prioritize datasets that have typically been used in the literature on counterfaction explanations including: Adult [**?**], Boston Housing [**?**], COMPAS [**?**] and German Credit [**?**]. That being said, there is also scope for adding data sources that have so far not been explored much in this context including image, audio, natural language and timeseries data.

Improved data preprocessing

Support for data preprocessing is currently limited to adding mutabilitiy and domain constraints. For practical use, the package should ideally be able to natively handle categorical data. It should also offer support for scale independence. The basic module for this is already in place and should be realtively easily extended.

Concluding remarks

The goal of this paper is 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. We envision this package to be a go-to place for explaining arbitrary predictive models through a diverse suite of counterfactual generators. It 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.