Flow-based counterfactuals for interpretable graph node classification

Lorenz Ohly

Fachbereich Mathematik und Informatik
Freie Universität Berlin

This dissertation is submitted for the degree of
Bachelor Informatik

February 2022

Declaration

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Lorenz Ohly
February 2022

Abstract

As more deep learning models are deployed for high-stakes use cases, explaining the predictions of a model is becoming more important. One class of methods for explainability are counterfactual examples. A counterfactual modifies a model input in such a way that the model output, for example a classification, changes in a target direction. In this work, we apply an efficient method for generating such counterfactuals [15] (ECINN) to graph node classification. We introduce a synthetic graph dataset with ground-truth explanation labels. Using this dataset, we quantitatively compare the model-specific ECINN method against a model-agnostic counterfactual generation method by Wachter et al. [36] on explanation size and correctness. We find that ECINN [15] produces higher-quality counterfactuals and discuss the trade-offs between it and model-agnostic methods.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

1.1 Overview

In recent years, deep learning has been used to great success at many learning tasks, achieving state-of-the-art results across various domains. One of these domains is graph data, which lends itself to modeling structures like citation networks, social networks, chemical molecules, and point clouds. There, deep learning is advancing the state-of-the-art for many tasks like node classification [37], link prediction [21], graph classification [39], and graph generation [38], thereby improving the feasibility of real-world use cases like molecule discovery [16] and self-driving cars [3]. Graphs can represent data from many domains, and improved learning algorithms for graphs enable us to make use of and understand this data better.

Even if a model is performing well, interpreting the predictions of deep learning models is critical to deploy these models responsibly. Recent work in adversarial attacks on image classification models has shown that slight, humanly imperceptible changes to an input image can change the classifier's prediction dramatically [34]. Deep neural networks are not intuitively interpretable due to the high dimensionality of their parameters, so other methods have been devised to interpret model outputs.

Counterfactual explanations [36][5] form a family of explanation methods. Given an input and a target model output, a counterfactual example answers the question: "What changes in input features lead to a different, target prediction?". The counterfactual is generated by modifying the orginal input's features such that the model output changes to a target value. By comparing the input and the counterfactual, we can identify relevant features that the model uses to differentiate between the original and the target prediction.

Applied to a classifier, a counterfactual should change an input's features so that the model predicts a different class. As Dandl et al. [5] exemplify, consider a person denied a bank loan by an algorithm that classifies whether applicants should receive a loan. A counterfactual indicates which variables need to change for the loan's approval and hence provides an actionable explanation, as long as the variables are under the applicant's control.

To be useful, counterfactuals should reliably change the model output in the target direction (correctness), only change semantically important features (minimality), and be actionable [36] [15]. Different methods of generation optimize for different properties. For example, Wachter et al. [36] optimize a two-objective loss function that minimizes the distance to the target value (correctness) while maximizing the similarity to the original sample (minimality). Dandl et al. [5] use two additional terms, optimizing for sparse feature changes (minimality and actionability) and the likelihood of the generated counterfactual (realism).

The methods by Wachter et al. [36] and Dandl et al. [5] are counterfactual methods that work with black-box models, as they do not make assumptions about the model and use general optimization algorithms like gradient descent to iteratively refine a counterfactual.

There are also methods that assume some model structure and hence work only with a specific class of models. Hvilshøj et al. [15] propose such a method called *Efficient Counterfactuals from Invertible Neural Networks* (ECINN) that works on a so-called generative classifier. By making use of the structure of this known, so-called white-box model, it can generate counterfactuals in only two model evaluations, in contrast to most other methods which require expensive, iterative optimization.

In this thesis, we apply counterfactuals to graph data by generating counterfactuals at the node level. Given a model trained for node classification, our node counterfactuals change the node features such that the model's prediction changes to a target class. While the graph's edge information is used in the model, the counterfactual modifies only the node features, not the connectivity. These node feature counterfactuals can be useful in many scenarios. For example, consider a researcher who wants to get their paper accepted by a journal. Given a graph of papers with a binary acceptance label and word features for every paper, a counterfactual could identify words (or topics) the researcher should use less or more often to have the paper accepted.

We compare the performance of the model-agnostic counterfactual method of Wachter et al. [36] and the model-specific ECINN Hvilshøj et al. [15] method by generating such node counterfactuals. We expect to find ECINN to be not just more computationally efficient, but also to produce higher-quality counterfactuals because the underlying generative classifier explicitly learns to model the distribution of data.

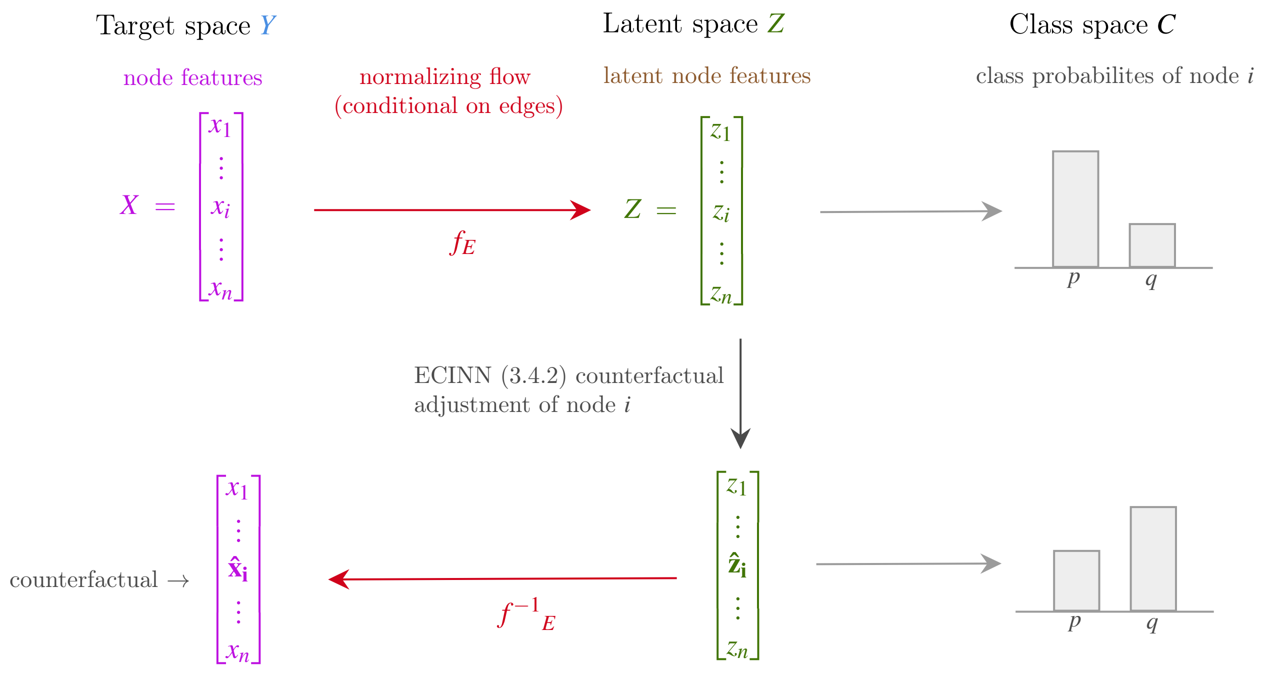

Target space Y
node features
$X = \begin{bmatrix} x_1 \\ \vdots \\ x_i \\ \vdots \\ x_n \end{bmatrix}$
normalizing flow
(conditional on edges)
f_E
Latent space Z
latent node features
$Z = \begin{bmatrix} z_1 \\ \vdots \\ z_i \\ \vdots \\ z_n \end{bmatrix}$
Class space C
class probabilites of node i
p
q
ECINN (3.4.2) counterfactual
adjustment of node i
counterfactual →
$\begin{bmatrix} x_1 \\ \vdots \\ \hat{\mathbf{x}}_\mathbf{i} \\ \vdots \\ x_n \end{bmatrix}$
${f^{-1}}_E$
$\begin{bmatrix} z_1 \\ \vdots \\ \hat{\mathbf{z}}_\mathbf{i} \\ \vdots \\ z_n \end{bmatrix}$
p
q

Fig. 1.1 Overview of our approach to generating graph node counterfactuals

To generate a counterfactual for node with features $x = X_i$ all node features X are embedded into the latent space Z with a normalizing flow. Then only the node's latent features are modified using ECINN, and then the modified batch of latent features is cast back to the target space using the inverse of the flow. Both directions of the flow are conditional on the graph's edges E. Best viewed in color.

1.2 Contributions

We first demonstrate that generative classifiers are a viable alternative to the regular, discriminative classifier on graph node classification tasks (5.1). For this, we compare their classification peformance on the widely-used standard benchmark datasets Cora [27], PubMed [25], and CiteSeer [10].

Next, we show how to create node counterfactuals that can be used to explain predictions made by graph node classification models. Our approach is detailed in section 4.4.

Finally, we compare the performance of the counterfactual generation methods by Wachter et al. [36] (model-agnostic) and Hvilshøj et al. [15] (model-specific). To quantify the quality of counterfactuals, we introduce a synthetic graph dataset (4.2) with ground-truth explanation labels. Using this dataset, we evaluate both methods on explanation magnitude and correctness (5.2.1).

1.3 Outline

In Chapter 2, we cover related work. Chapter 3 provides the theoretical background for the methods we use. Chapter 4 introduces reference datasets, our proposed dataset, and model architectures for the experiments detailed in chapter 5. Chapter 6 discusses the results of the experiments. Finally, chapter 7 summarizes our findings and gives directions for future work.

Chapter 2

Related Work

In this chapter, we give an overview of research relevant to this thesis. We first reference work on deep learning and, in some detail, deep generative models. We then examine prior art on normalizing flows, a particular family of deep generative models. Finally, we summarize work done on counterfactuals. We give more theoretical background in chapter 3.

2.1 Deep learning

Deep learning models, also called deep neural networks, are a class of learning algorithms that have again become popular in the last decade. Neural networks learn to approximate complex mappings and are hence useful in many domains, even those with very high-dimensional data like images. Particularly, deep learning models have achieved state-of-the-art results on learning tasks in domains such as computer vision [23] and natural language processing [4]. Theoretical background on the construction and training of deep neural networks can be found in section 3.1.

2.1.1 Graph neural networks

Following the success of deep learning in computer vision, there has also been more research done on applying neural networks to graphs. Yang et al. [37] approach node classification and link prediction by embedding node features with deep models and thereby improve on the previous state-of-the-art. Shortly after, Kipf and Welling [22] introduced the graph convolutional layer that generalizes the widely used convolution to the irregular neighborhoods of graphs. We use graph convolutions in our work and give further background on learning on graphs and graph neural networks in section 3.3.

2.1.2 Generative models

Aside from common supervised learning tasks like classification and regression, neural networks are also being applied in a generative setting. Generative models learn some aspects of the dataset they are trained on, allowing one to score the likelihood of a sample or to generate new samples. Variational autoencoders [20] embed samples in a lower-dimensional latent space of distribution parameters that can be used to generate many distinct samples or processed further by unsupervised clustering algorithms. Introducing Generative Adversarial Networks (GAN), Goodfellow et al. [12] train a generative model adversarially by having it fool a discriminator that is trained in tandem to differentiate between real and generated samples. GANs can generate high-quality samples without explicitly modeling a distribution. Diffusion models [31][32][14] learn to add (and subtract) Gaussian noise through a Markov chain of diffusion steps. Like GANs, they generate high-quality samples but, while GANs model a distribution implicitly, a diffusion model's distribution can also be evaluated analytically. However, due to the iterative nature of sampling, generation is slow[32] compared to other generative architectures.

In this work, we apply and investigate another family of generative models, so-called normalizing flows.

2.2 Normalizing flows

Normalizing flows [6] [7] are a family of deep generative models that explicitly model the data distribution. A flow is an invertible mapping trained to transform a simple base distribution like a Gaussian to a more complex target distribution. Given the right choice of mapping, this enables efficient training, sampling, and density estimation. Like other deep learning models, normalizing flows are made up of a sequence of parameterized, invertible layers. We use affine coupling layers [7] (described in detail in 3.2.2) as they are efficient and can be parameterized by arbitrary neural networks. We give a more detailed background in chapter 3.2.

2.2.1 Normalizing flows on graphs

Normalizing flows have already been applied to graph data, for example, to generate molecule graphs [38] or synthesize human motion [13]. To create molecule graphs, Zang and Wang [38] model the node features and adjacency matrix with a flow. These molecule graphs are usually small, but in general, representing a graph's edges densely with an adjacency matrix is costly since its size scales quadratically with the number of nodes. Liu et al. [24] also create molecular graphs using flows, but learn a low-dimensional representation of the adjacency matrix using an autoencoder to tackle this problem. In this work, we also apply normalizing flows to graph data, but model only the node features with the flow. The static edge information is incorporated through graph convolutional layers used inside the flow. Instead of graph generation, we focus on making node classification models explainable. The practicalities of our approach are described in section 4.1.

2.2.2 Generative classifiers

Ardizzone et al. [2], Dinh et al. [7] and Kingma and Dhariwal [19] use normalizing flows for classification tasks by employing a mixture of Gaussians as the flow's base distribution. Besides classification, this generative classifier also allows class-conditional sampling and density estimation. To optimize both generative and classification abilities during training, they use an objective called Information Bottleneck Loss [35] that balances both terms. In this work, we train generative classifiers for graph node classification. We cover the theory of constructing and training generative classifiers in section 3.2.3.

2.3 Counterfactuals

Between deep learning models being widely deployed and their black-box nature, it is becoming more important to explain the predictions they produce.

Large parts of the explainability literature have focused on attribution-based models [30, 29] that score the importance of each input feature for making the prediction, often using gradient information. These are widely applicable but fragile [9] and can be hard to interpret.

A different family of interpretability methods are example-based: Prototypes and criticisms [17] are especially representative or outlier data points in a (sub)set of data. These can be used to better understand the dataset and ensure that a model behaves predictably even on edge cases.

Another class of example-based methods are counterfactuals [36]: these are modified samples that explain which features need to be changed to obtain a target change in the model's output, as explained in chapter 1.

Wachter et al. [36] generate counterfactuals from an input through optimization. They construct a loss function that rewards change toward the target output but also similarity to the original input. Dandl et al. [5] take a similar approach but add two additional terms to the loss function to also optimize for sparsity and likelihood. Their methods are both iterative and optimization-based, requiring many queries to the model to obtain a single counterfactual example. Further, they do not make assumptions about the structure of the model and thus can be used with any black-box model.

Recently, Hvilshøj et al. [15] proposed *Efficient Counterfactuals based on Invertible Neural Networks* (ECINN), a method that generates counterfactuals without requiring iterative optimization. Unlike the previous methods, it works on a specific model architecture, the generative classifer. ECINN leverages the latent space to compute counterfactuals efficiently.

In our work, we compare the model-agnostic method by Wachter et al. [36] to the model-specific ECINN [15] method. We describe both methods in detail in section 3.4.

2.3.1 Counterfactuals on graphs

There has also been some prior work on generating counterfactuals in the context of graph data. Abrate and Bonchi [1] create whole-graph counterfactuals that change the output of a classifier trained to distinguish brain structures represented as graphs. Zhao et al. [40] approach link prediction by forging counterfactual edges. In contrast to these approaches, we focus on creating counterfactual node features for node classification (see 4.4 for details).

Chapter 3

Theoretical background

In this chapter, we provide background to the methods we apply in our experiments, covering deep learning, normalizing flows, graph neural networks, and counterfactuals.

3.1 Deep learning

Deep learning models form a family of algorithms for learning from data. We start off by defining the supervised learning setting in which these models are often used. Then, we explain how deep models are built by sequencing smaller components and how to train them. We also touch on the importance of generalization and finally cover classification, a common task for learning algorithms that is central to our work.

Supervised learning

The supervised learning setting poses the following problem: given a dataset of input and target pairs $(x,y) \in X \times Y$, find a mapping $f(x) = y : X \to Y$ that maps each input to a target. For example, the inputs could be images containing either a dog or a cat and the target a binary variable that indicates whether a dog or a cat is present in the image. The goal is to use a set of available, labeled data to learn a mapping that generalizes to new, unseen data.

We call such a learnable mapping a model parameterized by parameters Θ. Training the model amounts to finding the realization of parameters Θ' that optimizes an objective function. This objective comes in the form of a *loss function* $\ell(f(X),Y)$. The lower the loss, the better the model's prediction $f_\Theta(x)$, so to train the model we minimize the loss function.

Deep models

Deep learning models, also called neural networks, are generally made up of a sequence of smaller models, or layers, that successively transform the input. The choice of layer determines how well a model can approximate the mapping $X \to Y$. More layers generally result in a model with more representational power. One simple layer is an affine transformation:

$$A(x) = W * x + b \tag{3.1}$$

$$\Theta_A = (W, b) \tag{3.2}$$

It consists of a matrix multiplication and a vector addition with its parameters W and b. By varying the sizes of these, the layer becomes a mapping $\mathbb{R}^n \to \mathbb{R}^m$ for any n, m. We can create a model with better representational power by sequencing many of these layers after each other. Sequencing affine layers naively, however, does not result in improved representational power because every composition of affine transformations is itself a single affine transformation. Instead, it is common practice to add an activation function at the end of each layer. This activation is a non-linear function applied element-wise to every layer's output. An activation that is simple and computationally efficient is the Rectified Linear Unit, defined as $ReLU(x) = max(0, x)$. Interleaving linear layers with these non-linear activation functions results in expressive models.

Training deep models

The standard way of training deep models is through gradient descent. For this, both model and loss function are chosen such that the partial derivative in respect to the weights

$$\frac{\partial \ell(f_\Theta(X), Y)}{\partial \Theta} \tag{3.3}$$

is tractable.

This derivative, also called a gradient, indicates the direction and magnitude in which the model's parameters needs to be adjusted to decrease $L(f(X), Y))$. We can then update each parameter value, given an update rule (also called an optimizer). A simple update rule is

$$\Theta = \Theta - \eta G \tag{3.4}$$

where $\eta \in \mathbb{R}^+$ is the so-called learning rate and controls the magnitude of the update. η is usually smaller than 1 but must be finetuned for every task. To perform gradient descent, we repeatedly compute the loss and derivatives and update Θ. Commonly, we partition the dataset into equally-sized batches and perform the gradient descent step on each batch, which we call stochastic gradient descent. This is done for multiple epochs, with every sample being seen once per epoch.

In our experiments, we use the ADAM [18] optimizer, a variant of gradient descent. It adds an exponential moving-average term to the update rule and generally requires less finetuning of the learning rate η.

Generalization

A model that memorizes every sample it sees during training would achieve a perfect loss. However, this is not desirable as the model would not generalize to unseen data. We can measure generalization as the difference between the model's performance on training data and unseen data. Specifically, we partition the dataset into a training set and a validation set. Only the training set is used during training, so we can measure whether the model generalizes to data in the validation set.

Generalization is a complex topic with many techniques aiming to improve it, like (1) using a model with domain-specific architectural priors, e.g convolutions for image processing, (2) stopping training once the loss on the validation dataset starts increasing, and (3) employing regularization techniques like batching, weight decay, or Dropout [33].

Classification

A common supervised learning task is classification. Here, one wants to map an input to one of a number of possible classes. For example, classifying whether an image contains a cat or a dog can be represented as a 2-class classification task $\mathbb{R}^{h \times w} \to C$ with $C = \{\text{cat}, \text{dog}\}$ and $c = |C| = 2$.

The target space C is discrete, but neural networks are easiest to optimize for continuous mappings. To circumvent this, we construct a model to output a continuous c-dimensional space of class probabilities instead of a discrete class label. As a result, we have a continuous objective during training, but the predicted class can still be recovered by selecting the index of the output with the highest predicted probability. The targets for training are constructed from the class label y by one-hot encoding, resulting in a vector with c elements where the y-th element is 1, and all others are 0.

To optimize this classifier, we also need an appropriate differentiable loss function $\ell(y', y) : \mathbb{R}^c \times \mathbb{R}^c \to \mathbb{R}$. Commonly used for classification with multiple classes is categorical crossentropy:

$$\ell(y', y) = -\sum_{c=1}^{C} \log \frac{\exp(y'_c)}{\sum_{i=1}^{C} \exp(y'_i)} y_c \tag{3.5}$$

Since the magnitude of loss functions tends to be hard to interpret on its own, it is common to use additional metrics to assess model performance. Metrics need not be differentiable because they are not used for optimization.

The standard metric for classification tasks is accuracy, defined as the fraction of correctly classified samples. However, accuracy will be biased in the presence of class frequency imbalances. An alternative metric that handles class imbalances better is the Macro F1-score, defined as the mean of the accuracies of each class' samples. Formally,

$$\text{accuracy}(Y', Y) = \frac{1}{n} \sum_{i=1}^{n} 1_{\text{argmax}(Y'_i) = \text{argmax}(Y'_i)} \tag{3.6}$$

$$\text{MacroF1}(Y', Y) = \frac{1}{c} \sum \text{accuracy}(Y'_c, Y_c) \tag{3.7}$$

for n model outputs Y' and targets Y and Y_c the subset of targets with class labels c.

3.2 Normalizing Flows

Normalizing flows [7] are one family of generative deep learning models which learn to model a probability distribution. A flow is an invertible mapping from a simple distribution to a more complex target distribution.

We formally define a flow $f : Y \to Z$ as follows. Let $Z \in \mathbb{R}^n$ be a base distribution with a known probability density function $p_Z : \mathbb{R}^n \to \mathbb{R}$ and $Y = f^{-1}(Z)$ a target distribution. Notably, due to the required invertibility, the base and target distribution must be of the same dimensionality. Using the change of variables formula, the density of Y can be evaluated as:

$$p_Y(y) = p_Z(f(y)) |\det Df(y)| \tag{3.8}$$

$$= p_Z(f(y)) |\det Df^{-1}(f(y))|^{-1} \tag{3.9}$$

where f and f^{-1} are a bijective function and its inverse, $Df(y) = \frac{\partial f}{\partial y}$ is the Jacobian of f and $Df^{-1}(z) = \frac{\partial f^{-1}}{\partial z}$ is the Jacobian of f^{-1}.

We can sample from a flow's distribution by drawing a sample z from the known base distribution and passing it through $f^{-1} : Z \to Y$. Hence f^{-1} is called the *generative* direction of the flow while f is the *normalizing direction*. A common choice for the base distribution p_Z is a standard Gaussian distribution $\mathscr{N}_n(0, 1)$ [7].

3.2.1 Density estimation

The flow f and base distribution p_Z are parameterized by parameters Θ and Φ, respectively. Adjusting these allows training a flow for density estimation. Given a dataset $\mathscr{D} = \{y_i\}_{i=1}^N$ from a target distribution, we want to maximize the log-likelihood

$$logp(\mathscr{D}|(\Theta,\Phi)) = logp_Z(f(\mathscr{D}|\Theta)|\Phi) + log|\det Df(\mathscr{D}|\Theta)| \tag{3.10}$$

which is the sum of the log-likelihood of the data under the base distribution and the log Jacobian determinant of the flow. To maximize this term, we choose as loss function the negative log-likelihood. During training, we must evaluate both the normalizing direction f and its log determinant, while evaluating f^{-1} is necessary for sampling. As a result, the tractability of computing both directions is critical when constructing a flow f.

3.2.2 Layers

Composition

An important observation for building flows is that a sequence of invertible bijections is itself an invertible bijection with tractable Jacobian determinant. If we construct the generative direction of the flow $f^{-1} = f_n \circ f_{n-1} \circ ... \circ f_2 \circ f_1$, then the normalizing direction is $f = f_1 \circ f_2 \circ ... \circ f_{n-1} \circ f_n$. The determinant of the Jacobian of f is the product of each function's Jacobian determinant. So if we define $y_i = f_{i+1} \circ ... \circ f_n(y)$, then

$$\det Df(y) = \prod_{i=1}^{n} \det Df_i(y_i)$$

Using this property, we can construct deeper normalizing flows with more representational power than a single layer.

Affine coupling layers

One choice of building block for normalizing flows is the affine coupling layer [7]. An affine coupling layer is a function $f : \mathbb{R}^D \to \mathbb{R}^D$ that achieves invertibility by splitting an input x into two parts $x_{1:d}$ and $x_{d+1:D}$ with $d < D$. $f(x) = y$ then transforms both parts as follows:

$$y_{1:d} = x_{1:d} \tag{3.11}$$

$$y_{d+1:D} = x_{d+1:D} \odot \exp(s(x_{1:d})) + t(x_{1:d}) \tag{3.12}$$

Here, $t : \mathbb{R}^d \to \mathbb{R}^d$ and $s : \mathbb{R}^d \to \mathbb{R}^d$ are two functions with parameters θ_1 and θ_2, and $\odot$ is element-wise multiplication. Since part of the input space is left unchanged, we can calculate the inverse of $f(x) = y$ as

$$x_{1:d} = y_{1:d} \tag{3.13}$$

$$x_{d+1:D} = (y_{d+1:d} - t(y_{1:d})) \odot \exp(-s(y_{1:d})) \tag{3.14}$$

While the Jacobian for bijective functions is, in general, expensive to compute, Dinh et al. [7] show that the affine coupling layer's Jacobian is triangular. As a result, the determinant of the Jacobian can be efficiently computed as $\exp\left[\sum_j s(x_{1:d})_j\right]$. Importantly, no computation of s or t's Jacobians is needed, enabling the use of arbitrary differentiable functions $s, t : \mathbb{R}^n \to \mathbb{R}^n$. In practice, we model these subnets s and t using deep neural networks allowing us to represent complex functions and optimize the subnets using gradient descent.

It is also common to use the same network structure for both s and t. Additionally, since a single affine coupling layer only transforms part of the input space, two coupling layers are often sequenced such that each transforms one-half of the feature space.

We adopt affine coupling layers for constructing normalizing flows in our work because they have (1) efficient forward and inverse evaluation, (2) tractable Jacobian determinants, and (3) expressive parameterization.

3.2.3 Generative classifiers

We can also use normalizing flows for classification with an architecture called a generative classifier, as shown by Ardizzone et al. [2] and Mackowiak et al. [26]. To do so, the authors use a Gaussian Mixture Model (GMM) with class means μ_c and standard deviations fixed to 1 as base distribution of a normalizing flow. This allows us to model conditional likelihoods: the class-conditional base distribution is defined through

$$p(Z|Y) = \mathscr{N}(Z;\mu_c,1) \tag{3.15}$$

$$p(Z) = \sum_y p(y)p(Z|y) = \sum_y p(y)\mathscr{N}(Z;\mu_y,1) \tag{3.16}$$

As such, we can calculate the class-conditional likelihood by putting the class' mixture component into Eq. 3.9. To classify a sample, we find the mixture component which scores its likelihood highest. Since the mixture components are Gaussians with constant standard deviations, this is equivalent to selecting the nearest neighbor of the sample in latent space.

Neither the log-likelihood loss 3.2.1 nor the categorical cross-entropy loss are suitable loss functions to train a generative classifier since they penalize only generation *or* classification error. A proposed loss function to train generative classifiers is the Information Bottleneck [35][2][26], which consists of two terms that encourage a model to improve both its generative and classification abilities.

$$\mathscr{L}_{IB}(x,y) = \mathscr{L}_X(x) + \beta\mathscr{L}_Y(y) \tag{3.17}$$

$$\mathscr{L}_X(x) = -\log|detD_f(x)| + \text{logsumexp}_y(v_y^2 - 2w_y) \tag{3.18}$$

$$\mathscr{L}_Y(x,y) = \text{onehot}(y)\cdot\text{logsoftmax}_y(\frac{v_y^2}{2} - w_y) \tag{3.19}$$

where $w_y = \log(1/c)$ (uniform class priors), $v_y = f(x) - \mu_y$, and logsumexp and logsoftmax sum over the class dimension. L_X penalizes a low likelihood of the samples regardless of class label, while L_Y acts similarly to categorical cross-entropy to ensure samples are classified correctly. The factor β trades off the weight given to either loss component.

3.3 Learning on graphs

Graphs lend themselves to modeling data in various domains, from social networks and chemical molecules to geospatial data. A graph $G = (V,E)$ consists of nodes V and edges $E \subseteq V \times V$, $n = |V|$ connecting these nodes. For example, we can represent a social network with persons as nodes and friendship status as edges.

Various supervised learning tasks involving graphs exist. In *graph classification*, we have a dataset of graphs $\mathscr{G} = \{G_1, G_2, ..., G_i\}$ and want to learn the mapping $\mathscr{G} \rightarrow C$ from graph to class. For *node classification*, we want predict a class for each node instead, while in *Link prediction* we try to predict a node's edges.

In each of these tasks, the nodes can carry additional features which we represent as a $|V| \times f$-matrix X with f features per node.

3.3.1 Graph neural networks

To use neural networks effectively on graphs, we need a model that incorporates both node and edge information which we write as $X' = \text{model}(X|E)$.

Kipf and Welling [22] propose a graph convolution layer that can be used to build such graph neural networks, defined as

$$\text{GraphConv}(X|E) = \hat{D}^{-\frac{1}{2}} X \hat{D}^{-\frac{1}{2}} X \Theta \tag{3.20}$$

where $\hat{A} = A + I$ is the adjacency matrix with self-loops, $\hat{D}_{ii} = \sum_{j=0} \hat{A}_{ij}$ is the diagonal degree matrix, and Θ is a learnable weight parameter. Like in other deep learning models, we stack this layer to build a model that successively transforms the node features. The resulting features can then be classified individually for node classification, or aggregated for graph classification.

3.4 Counterfactuals

Counterfactuals [36] are a method to explain the predictions of a model. A counterfactual answers the question of how to change a given input in order to change the model output in a desired direction. Formally, given an input x and corresponding model output y, we want to construct an input $\hat{x}$ for which the model outputs a desired target $\hat{y}$. In the case of a classifier, the original output is a class p, and the desired target a different class q. Comparing the features that differ between input x and counterfactual $\hat{x}$ identifies relevant features. While attribution-based methods find features that are relevant for arriving at a prediction, counterfactuals additionally indicate features that differentiate between different classes of outputs.

To be able to compare model-specific and model-agnostic counterfactual generation methods, we employ two methods in our experiments. ECINN [15] allows for particularly efficient counterfactual generation by making use of normalizing flows, while Wachter et al. [36] is representative of the model-agnostic methods that do not make assumptions about the model they are used on.

3.4.1 Wachter et al.

Wachter et al. [36] introduce an iterative optimization-based counterfactual generation method. They propose a 2-term loss function consisting of (1) a term of similarity to the original sample and (2) a task-specific loss that rewards changing the model output in the right direction. Given a sample x, a counterfactual $\hat{x}$, and a target model output y, the loss function is defined as

$$L(x,\hat{x},y) = \lambda L_{\text{task}}(\hat{x},y) + L_{\text{sim}}(\hat{x},x) \tag{3.21}$$

$$L_{\text{sim}}(\hat{x},x) = \sum_{i=1}^{p} \frac{|x_i - \hat{x}_i}{MAD_i} \tag{3.22}$$

$$MAD_i = \text{median}(|X_i - \text{median}(X_i)|) \tag{3.23}$$

with p the number of features and MAD_i the median absolute deviation of a feature calculated once over all samples. The MAD is used as a robust estimator of feature variability that normalizes the features' scale.

The task-specific term L_{task} depends on the application. Since we are interested in generating counterfactuals for a classification task, we use categorical cross-entropy.

To generate a counterfactual, an initial value $\hat{x}^{(0)}$ is chosen. The loss function is then optimized using gradient descent with respect to the counterfactual, repeatedly updating its values. This process starts with a low value λ_{init}. We use an exponential schedule for λ, multiplying it by a factor f after each step. The process ends once a desired change in the model output is reached. For our classification counterfactuals, we stop once the model output reaches the target class q.

3.4.2 ECINN

In [15], the authors propose *Efficient Counterfactuals based on Invertible Neural Network* (ECINN), a method for generating counterfactuals that requires neither repeated model evaluations nor optimization. Instead, it uses a generative classifier and its invertibility to calculate a suitable counterfactual.

The counterfactual is generated through the following steps. First, the input x is normalized using the model's flow f, resulting in the latent vector $z = f(x)$. Then, z is translated by a vector $\alpha\Delta$ and transformed back to the target space using f^{-1}. For an original class p and a target class q, we write

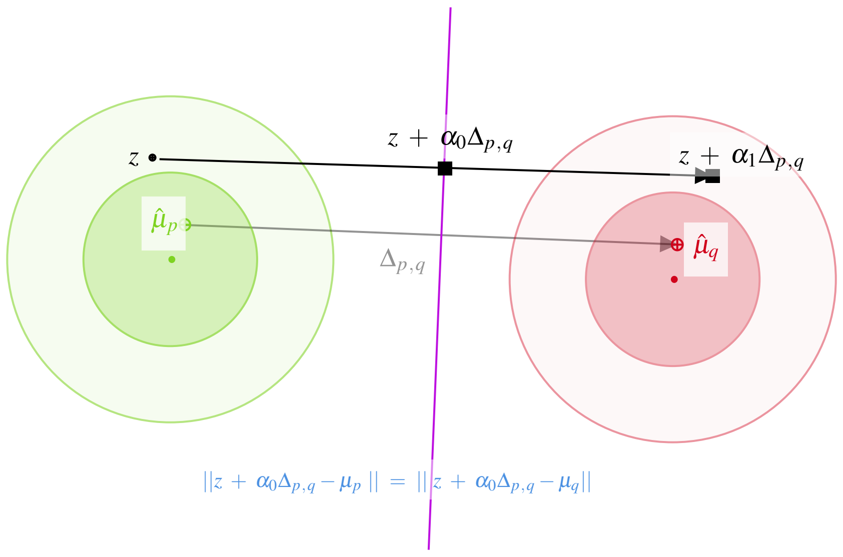

z
$z + \alpha_0 \Delta_{p,q}$
$z + \alpha_1 \Delta_{p,q}$
$\hat{\mu}_p$
$\hat{\mu}_q$
$\Delta_{p,q}$
$||z + \alpha_0 \Delta_{p,q} - \mu_p|| = ||z + \alpha_0 \Delta_{p,q} - \mu_q||$

Fig. 3.1 Latent space adjustments made by ECINN

$$\hat{x}^{(q)} = f^{-1}(f(x) + \alpha \Delta_{p,q}) \tag{3.24}$$

where the resulting $\hat{x}^{(q)}$ is a counterfactual classified as class q, but otherwise similar to the original sample x.

The translation vector consists of $\Delta_{p,q}$, the direction of translation and $\alpha \in \mathbb{R}$, the translation factor. Let $\hat{\mu}_c = \frac{1}{n}\sum_{i=1}^{n} f(x_i^{(c)})$ be the mean latent vector for all samples of class c. Then the direction moving a sample from class p to q is the difference $\Delta_{p,q} = \mu_q - \mu_p$.

Hvilshøj et al. [15] calculate two values for α. The first is α_0, which gives *tipping-point* counterfactuals that lie exactly on the classifier's decision boundary. Hence a tipping-point counterfactual constitutes a minimal set of feature changes needed to alter the prediction to the desired target class q. α_0 is derived by solving for it in

$$||z + \alpha_0 \Delta_{p,q} - \mu_p|| = ||z + \alpha_0 \Delta_{p,q} - \mu_q|| \tag{3.25}$$

where μ_c denotes the mean of a Gaussian mixture component, not the mean latent vector $\hat{\mu}_c$. The authors also define $\alpha_1 = w\alpha_0 + b$ to be the factor to generate *convincing* counterfactuals that are strongly classified as the target class q.

Chapter 4

Methods

In this chapter, we describe the methods we employ for our experiments. This includes model architectures and reference datasets. We also describe the construction of our synthetic node classification dataset. Then we explain in detail how we create node counterfactuals and finally cover what software and hardware we use.

4.1 Models

First, we describe the models we use in our experiments. Since we want to compare ECINN [15] counterfactuals to the model-agnostic Wachter et al. [36] method, we use two different models. The first is a standard discriminative classifier that we use as a baseline to evaluate Wachter et al. [36]'s method. The other is a generative classifier which is required to use ECINN. To disentangle the effects of counterfactual method and model architecture, we will evaluate the model-agnostic method by Wachter et al. [36] on both models. Within the constraints imposed by the generative classifier's normalizing flow, we try to structure both models similarly.

4.1.1 Discriminative classifier baseline

The baseline model is a regular discriminative classifier as described in section 3.1. It uses graph convolutions with ReLU activations as well as Dropout layers and outputs c class logits for every node. It is conditional on the edges E which are used in the graph convolution layers to aggregate neighborhood information. Figure 4.1 shows the network's structure and table 5.1 summarizes its layers' hyperparameters.

4.1.2 Generative classifier

The generative classifier also takes in node features X but outputs latent node features Z of the same dimensionality. As described in section 3.2.3, we construct it from a bijective normalizing flow with a Mixture of Gaussians as the base distribution.

The normalizing flow is composed of two affine coupling blocks which each have two affine coupling layers that transform half of the features as described in Section 3.2.2. The coupling layers' subnets s and t share the same structure: a graph convolution followed by ReLU activation and Dropout [33]. While the graph convolutions use the graph's edges, the edges are not modified by the network. As a result, one can consider the model to be generating node features conditional on the existing edges. Figure 4.1 visualizes the architecture and table 5.1 notes the layer hyperparameters used.

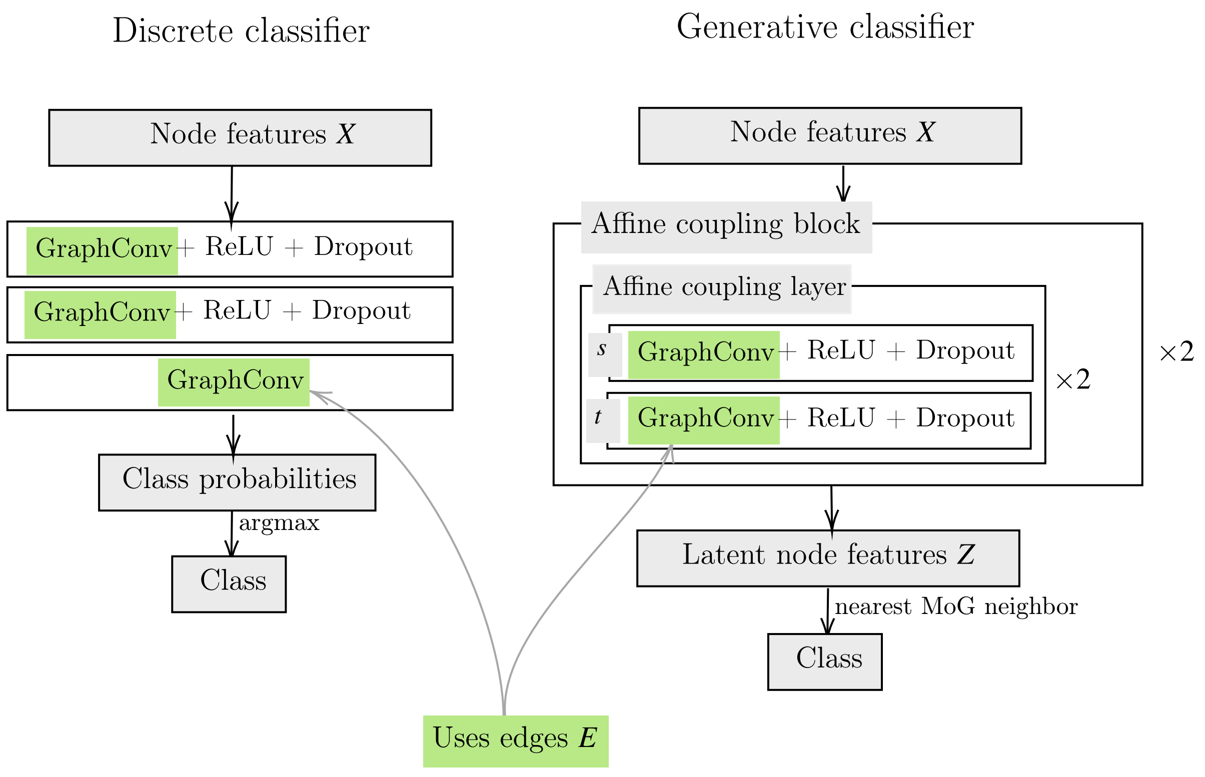


Fig. 4.1 Model architectures

4.2 Synthetic dataset

As discussed, one important property of a good counterfactual is that it modifies only semantically important features. In real-world datasets, however, this is usually impossible to evaluate quantitatively. As a result, many papers on counterfactuals resort to qualitative evaluation, showcasing and discussing a few examples. This comes with a few drawbacks: For one, comparing different methods' general performance is difficult based on only a few samples. Additionally, in domains with data that is not as readily humanly interpretable as computer vision and tabular data, it can be difficult to intuitively assess examples. Finally, qualitative comparisons are prone to cherrypicking and often highlight examples where a method performs particularly well compared to the reference.

To allow for a quantitative evaluation of counterfactual methods, we introduce a synthetic dataset for node classification which we describe in this section.

4.2.1 Construction

Our dataset is a graph with the same number of nodes for each of n_c classes. Every node has n features. Every class has an active feature set that is a random subset of all n features. Each feature is chosen with probability $p_{feature}$, resulting in feature sets $V_c \in \{0,1\}^n$ for each class c. As a result, the feature sets of every class pair p,q will tend to have some features in common. Features that are only in either feature set V_p or V_q can be used to distinguish whether a node comes from either class. We intentionally introduce these class-conditional relationships so we can evaluate how well counterfactuals take them into account.

Features $x_i; i \in \{1,...,n\}$ for a node of class c are then generated by drawing a sample from one of two Gaussian distributions, $\mathcal{N}_{inactive}$ or $\mathcal{N}_{active}$: if the feature is part of c's vocabulary, it is drawn from N_{active} with probability p_{active}. Otherwise, the feature value is drawn from $\mathcal{N}_{inactive}$. This introduces some noise so that each sample from a class differs. Finally, to model graph homophily, we add an edge between every node pair i,j with probability $p_{edge|c}$ if they are from the same class and a lower probability $p_{edge|\neg c}$ if not. The configuration values used in our experiments are summarized in Table 4.2.1.

Parameter	Value
n_c	5
n_{nodes}	100
$\mathcal{N}_{active}$	$\mathcal{N}(\mu = 1, \sigma^2 = 1/4)$
$\mathcal{N}_{inactive}$	$\mathcal{N}(\mu = -1, \sigma^2 = 1/4)$
p_{feature}	0.6
p_{active}	0.5
$p_{\text{edge}\|c}$	0.1
$p_{\text{edge}\|\neg c}$	0.01

Table 4.1 Synthetic dataset configuration used in our experiments

4.2.2 Ground truth of relevant features

Given a node with features x of class p, and a second class q we now define a ground truth mask of features that are relevant for differentiating this sample from those of class q. These are the features that we expect a counterfactual with target class q to change.

How do we define if a counterfactual changes a feature? We say that a feature $x^{(i)}$ was **meaningfully changed** if the counterfactual feature $\hat{x}^{(i)}$ changes which distribution (of $\mathcal{N}_{active}$ and $\mathcal{N}_{inactive}$) the value is most likely in. Concretely, if $x^{(i)}$ is closer to $\mathcal{N}_{inactive}$ and $\hat{x}^{(i)}$ closer to $\mathcal{N}_{active}$ we say that feature i was **activated**, and in the reverse case, we say feature i was **deactivated**. In practice, since we choose $\mathcal{N}_{active}$ and $\mathcal{N}_{inactive}$ to be Gaussians with means the same distance from 0, a meaningful change of a feature corresponds to a sign change of the feature value.

There are now two cases in which a feature i can be a relevant feature for a counterfactual. In both cases, i needs to be a feature that is in exactly one of p or q's feature sets. If the feature is only in the feature set of F_p, and the value x_i was sampled from the active distribution, a relevant change is to deactivate it. Alternatively, if the feature is in the feature set F_q only, activating it is a relevant change. Formally, we define the ground truth $GT \in \{true, false\}^n$ as

$$GT^{(i)} = ((F_p \wedge \neg F_q) \wedge x^{(i)}_{\text{active}}) \vee (\neg F_p \wedge F_q) \tag{4.1}$$

where x_{active} indicates features of this sample were drawn from the active distribution. Table 4.2.2 shows the values for an example sample, its ground truth and an example counterfactual.

Feature sets		Sample			Counterfactual		
F_p	F_q	x	x_{active}	GT	$\hat{x}$	$\hat{x}_{\text{changed}}$	
✓	-	0.9	✓	✓	-1	✓	True positive
✓	✓	1.1	✓	-	1	-	True negative
-	✓	-1.2	-	✓	-1	-	False negative
✓	-	-1	-	-	1.1	✓	False positive

Table 4.2 Example sample from synthetic dataset

A sample of class p from a dataset with $n = 4$ feeatures is generated using feature set F_p and a counterfactual $\hat{x}$ with target class q is given. GT indicates the features we expect the counterfactual to change and $\hat{x}_{\text{changed}}$ the features that did meaningfully change.

4.2.3 Metrics for evaluating counterfactuals

We can use this ground truth of relevant features to evaluate counterfactuals. We consider features that a counterfactual meaningfully changes a *positive* and other features a *negative*. With this, we define the following metrics:

Correctness

True-positive rate (TP) The true-positive rate quantifies how many of the relevant features are meaningfully changed (see 4.2.2). Depending on the counterfactual's application, a very high true-positive rate is not always better. If one wants to identify all relevant features that differentiate original class p from target class q, changing all features is helpful. At the same time, a sparser counterfactual that only changes a few of the relevant features will generally be more actionable, since fewer variables need to be modified. In any case, a very low true-positive rate indicates that the model output can be changed easily without affecting the relevant features. This could mean that the counterfactuals are unlikely in the data distribution and that the model is not robust to this.

False-positive rate (FP) We also measure how many irrelevant features were incorrectly changed in a meaningful way. This false-positive rate should be as low as possible.

Similarity

We also evaluate how *similar* a counterfactual $\hat{x}$ is to the original sample x.

Magnitude (total) The magnitude is the mean absolute feature change, i.e. $1/n \sum_{i=1}^{n} |\hat{x}_q^{(i)} - x^{(i)}|$. Since counterfactuals should be minimal, a lower magnitude is better.

Magnitude (irrelevant) Additionally, we calculate the magnitude on just the irrelevant features, meaning those where $\neg GT^{(i)}$. Unlike the total magnitude, which cannot be 0, the magnitude of irrelevant feature changes has a theoretical optimum of 0.

Table 4.3 shows how an example sample from table 4.2.2 is evaluated using these metrics.

Metric	Value
Ratio of true pos.	$1/2$
Ratio of false pos.	$1/2$
Magnitude (total)	$4.3/4$
Magnitude (irrelevant)	$2.2/2$

Table 4.3 Evaluation of example counterfactual

An example sample from the synthetic dataset with $n = 4$ and target class q. The feature set F_p defines how x is sampled. An example counterfactual $\hat{x}$ is evaluated. Metrics are also calculated for the counterfactual.

4.3 Citation datasets

When comparing ECINN to the model-agnostic method by Wachter et al. [36], we also need to ensure that the constraints on the model architecture do not limit the task performance. To ensure that generative classifiers are usable for graph node classification, we compare its performance to the baseline discriminative classifier on the three real-word citation datasets Cora [27], PubMed [25], and CiteSeer [10]. Each consists of a single citation graph where every node represents a paper and and each of the undirected edges represents a citation between two papers. Every node also has a class label that indicates which subject the paper is about. Papers about the same subjects are more tightly connected, a property known as homophily. Table 4.3 summarizes additional statistics of the datasets.

Dataset	$\lvert V \rvert$	$\lvert E \rvert$	Classes
Cora	2708	10556	7
CiteSeer	3327	9104	6
PubMed	9104	88648	3

Table 4.4 Citation dataset statistics

4.4 Generating node counterfactuals

In this section, we explain how we generate counterfactuals for nodes of a graph. Usually, to generate a counterfactual for a sample x, one can evaluate the model on the single sample: $\text{model}(x)$. However, the models we use take into account neighborhood information. We define the task of generating a counterfactual for a node as changing the output of the model while modifying only the selected node's features. This means that (1) the features of neighboring nodes are not modified and (2) no edges are modified. As a result, the counterfactual has to change the model output despite not being able to modify features in the neighborhood.

Having to incorporate edge information makes the generation of counterfactuals more expensive. Instead of evaluating the model on a single sample ($\text{model}(x)$), we need to input and evaluate the model on the whole graph ($\text{model}(X|E)$) to generate a single counterfactual.

Given a node at index i, and target class q, we provide additional details for the two counterfactual generation methods we use.

ECINN

ECINN uses a normalizing flow f to transform a sample to a latent space Z, translates it in the latent space and then maps it back to the input space using the inverse of the flow, f^{-1}. To generate a node counterfactual with ECINN while incorporating edge information, we (1) calculate all latent features $Z = f(X|E)$, (2) modify Z by translating (see 3.4.2) only the node's latent features Z_i, (3) evaluate the flow's inverse on the batch of modified latent features $X' = f^{-1}(\hat{Z}|E)$, and (4) select the node features $\hat{x} = \hat{X}_i$. See Figure 1.1 for a visualization of this approach.

While Hvilshøj et al. [15] fix the offset coeffcients for convincing counterfactuals to $b = \frac{4}{5}$ and $w = \frac{1}{2}$, we find that setting $b = 0$ and $w = 2$ yields consistent results without the need to finetune two additional hyperparameters.

Wachter et al.

This method requires performing gradient descent on the input space. We generate a counterfactual iteratively as described in section 3.4.1, but for every optimization step we do the following: (1) run the forward and backward pass of model$(X|E)$ to calculate the gradient with respect to the input node features X, (2) zero out all gradients except those for node i (3) apply the optimizer step.

We also use Wachter et al. [36]'s method together with a generative classifier. To compute the categorical cross-entropy for the task-specific loss term, we need an output of c relative class scores. We compute these by calculating the negative Euclidean distance of the latent features Z to the mean of each class' mixture component μ_c. Counterfactual generation is then performed as described in 3.4.1. Wachter et al. [36] recommend using a random initialization for $\hat{x}$, but we use $\hat{x}^{(0)} = x$ as we find that it results in better counterfactuals (as measured in chapter 5.

4.5 Software and hardware

We use PyTorch [28] as our deep learning framework of choice and PyTorch Geometric [8] for graph convolutional layers. Additionally, we use the FrEIA software package[1] by the authors of [26] for its implementation of affine coupling layers.

All experiments were conducted on a single NVIDIA GTX 1080 Ti graphics card.

[1]https://github.com/VLL-HD/FrEIA

Chapter 5

Experiments

This chapter describes the experiments we did with generative classifiers and counterfactuals. We first compare the classification performance of discriminative and generative classifiers and then evaluate counterfactuals quantitatively and qualitatively with our proposed dataset. Chapter 6 discusses the results presented here.

5.1 Generative classifiers for graph node classification

Before we investigate the counterfactuals produced by ECINN, we test that generative classifiers are usable for classification. After all, even if they are more interpretable, it is important that they compare to a discriminative classifier in terms of classification performance.

To test this, we train both kinds of classifiers on the three citation datasets described in 4.3. We use full-batch gradient descent to train both models. Table 5.1 summarizes the hyperparameter configurations for both models. They were tuned individually for the best performance on the Cora dataset and then used across all datasets. Since we find that the training of the generative classifier can be unstable, we employ gradient clipping to remedy this.

The results are summarized in Figure 5.1. We measure classification performance with the macro F1-score rather than accuracy due to the class imbalances in the datasets. We can see that the performance of both models is very similar.

5.2 Evaluating counterfactuals

Next we test the counterfactuals generated with ECINN and Wachter et al's method on correctness and magnitude as detailed in 4.2.3. For our experiments, we split the proposed synthetic

	Discriminative	Generative
Loss function	Categorical crossentropy	Information Bottleneck ($\beta = 1$)
Dropout strength	0.5	0.5
Epochs	300	300
Hidden layer size	256	128 (subnets)
Optimizer	ADAM($\eta = 0.001$)	ADAM($\eta = 0.0001$)
Gradient clipping	-	> 5

Table 5.1 Model and training hyperparameters for discriminative and generative classifer

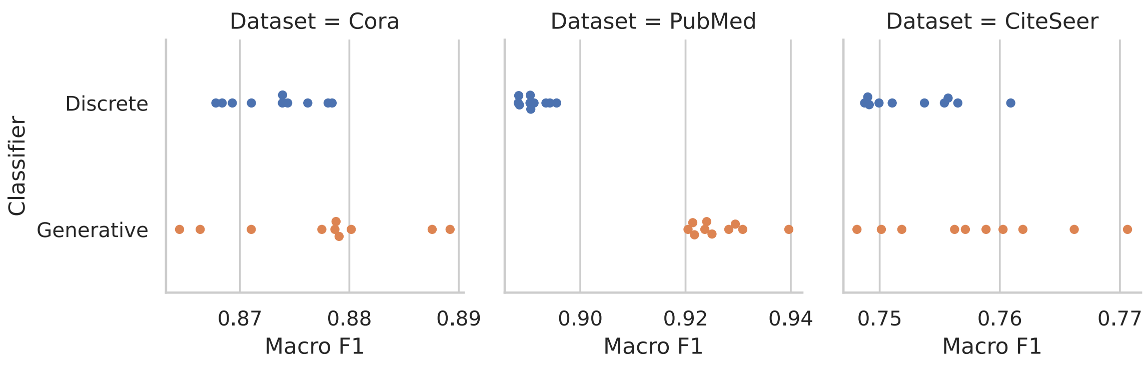


Fig. 5.1 Discriminative and generative classifier performance

Macro F1 score for the discriminative classifier and the generative classifier compared on the citation datasets Cora, PubMed and CiteSeer (4.3). We find no consistent difference in performance. We note that the generative classifier's performance varies slightly more across runs.

dataset (4.2) into an 80/20 training/validation split and train the discriminative and generative classifier (4.1) to classify the nodes.

5.2.1 Quantitative evaluation

We then generate counterfactuals with both models. We use ECINN to generate counterfactuals with the generative classifer and compare them to counterfactuals generated with the method proposed by Wachter et al. [36].

Both classifiers use the same model hyperparameters as in the previous experiments and are trained on our synthetic dataset. We train the discriminative classifier until the validation Macro F1-score converges to 100%, for 50 epochs. The generative classifier is trained until the generative loss converges; with learning rates $\{0.01, 0.0033, 0.001\}$ for 250 epochs each for a total of 750 epochs.

We use the following setups for generating counterfactuals:

- **ECINN (tipping)**: a generative classifier and ECINN with offset α_0 to generate tipping-point counterfactuals
- **ECINN (convincing)**: a generative classifier and ECINN with offset α_1 to generate convincing counterfactuals
- **Wachter (DC)**: a discriminative classifier and the method proposed by Wachter et al. [36] to generate counterfactuals.
- **Wachter (DC, no edges)**: same as the above, but with all edges removed from the dataset. See Appendix A.1 for an explanation.
- **Wachter (GC)**: a generative classifier and the method proposed by Wachter et al. [36] to generate counterfactuals.
- **Wachter (GC, no edges)**: same as the above, but with all edges removed from the dataset. See Appendix A.1 for an explanation.

We use the following set of hyperparameters we found performed best for Wachter et al's method: $\eta = 0.1$, $\lambda_{\text{init}} = 0.01$, $f = 1.01$.

After training, we take each node from the validation set and create counterfactuals for every differing target class q. Counterfactuals are generated as detailed in section 4.4.

Table 5.2.1 summarizes the results of the tested methods. Each model is trained using 5 different random seeds ($\{0,...,4\}$) and we report the mean of the runs. Since some configurations of Wachter's method do not converge to the target class, we stop the optimization after 300 steps and report as accuracy the percentage of samples for which a counterfactual could be successfully generated.

Method	TP	FP	Magn.	Magn. (irrelevant)	Accuracy
ECINN (tipping-point)	16.5%	**0.0037%**	**0.22**	**0.07**	**100%**
ECINN (convincing)	43.7%	0.025%	0.44	0.14	**100%**
Wachter (DC)	99.99%	25.9%	24.2	23.35	22%
Wachter (DC, no edges)	30%	12.2%	0.75	0.7	**100%**
Wachter (GC)	10.9%	7.9%	0.63	0.625	67.5%
Wachter (GC, no edges)	10.9%	4%	0.58	0.55	**100%**

Table 5.2 Performance of counterfactual generation methods

We report metrics for counterfactuals generated by different methods. The best result (where applicable) is **highlighted**. Results are discussed in chapter 6.

Training generative classifiers for ECINN

We also find that to get high-quality counterfactual generations with ECINN, the underlying generative classifier must have high generative performance. Figure 5.2.1 shows that the generative ability keeps improving long after classification performance converges.

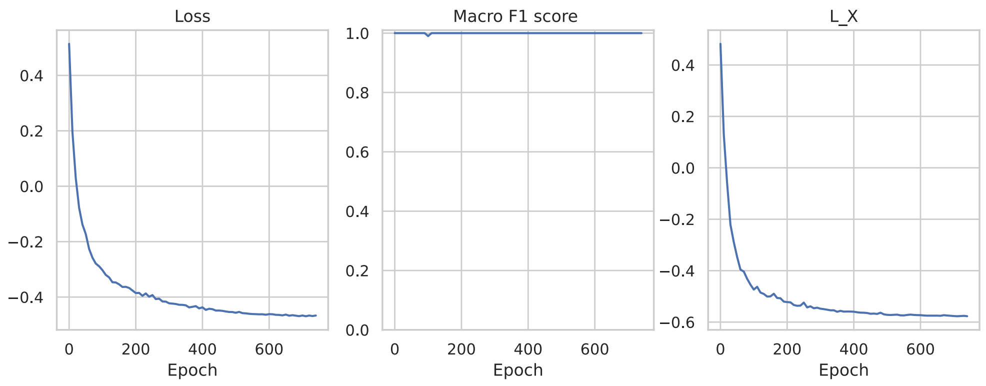


Fig. 5.2 Training progress of the generative classifier

While the classification performance quickly converges, the generative performance keeps improving for long after.

Controlling ECINN generation

Aside from the offsets α_0 and α_1 that generate tipping-point and convincing ECINNs, we also vary w to create in-between values. Figure 5.2.1 shows how magnitude and correctness can be traded off.

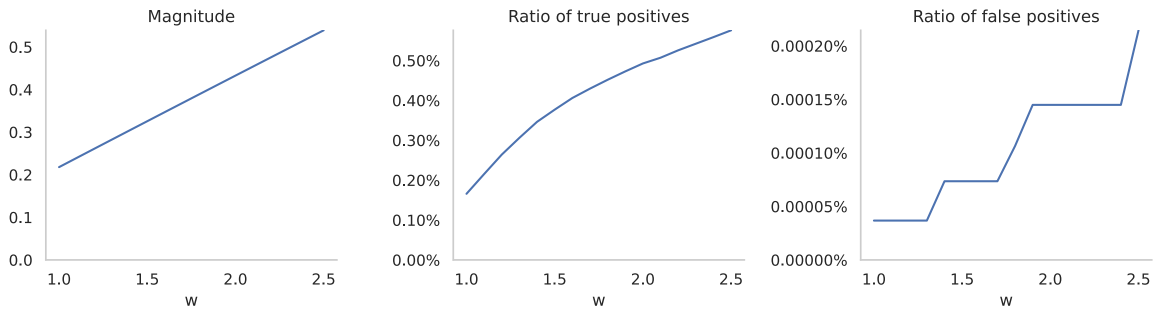


Fig. 5.3 Counterfactual metrics graphed to ECINN intensity

Varying the intensity of ECINN [15] counterfactuals between tipping-point and convincing provides predictable control. w is a factor that the tipping-point offset vector is multiplied with to generate convincing counterfactuals. $w = 1$ are tipping-point counterfactuals and $w = 2$ convincing counterfactuals. We see a smooth response in metrics for values between 1 and 2.

Computational efficiency

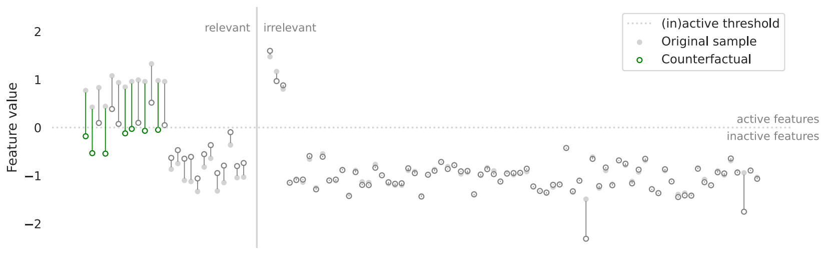
Next to the quality of counterfactuals, efficiency of generation is also a factor to consider. ECINN [15] can create counterfactuals in constant time: it always requires one evaluation of the flow f and its inverse f^{-1}. In contrast,Wachter et al. [36]'s method is iterative, with a variable number of iterations required. Under our set of hyperparameters used for *Wachter (GC)*, on average 80 iterations are required, with every iteration requiring a forward and backward pass through the model.

5.2.2 Qualitative evaluation

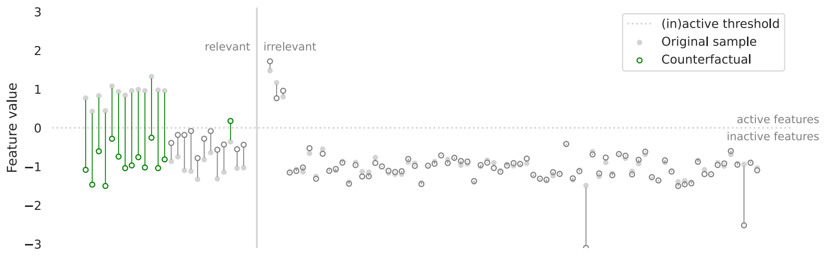
For a qualitative comparsion, we visually compare counterfactuals generated by the examined methods in figure 5.2.2.

We see for both variants of ECINN [15] that almost all large feature changes are to relevant features, while most irrelevant features are left unchanged. As should be expected, the convincing variant (b) makes larger changes than the tipping-point variant (a).

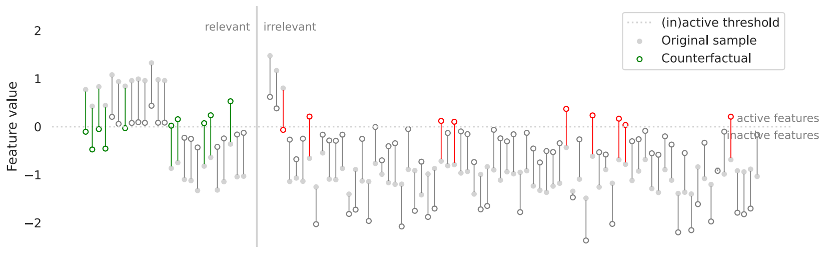
The two counterfactuals generated with Wachter et al. [36]'s method have similar magnitudes whether a feature is relevant or not. The counterfactual created for the generative classifier (d) makes only small changes to the features, while that created for the discrete classifier (c) makes large changes upwards and downwards.



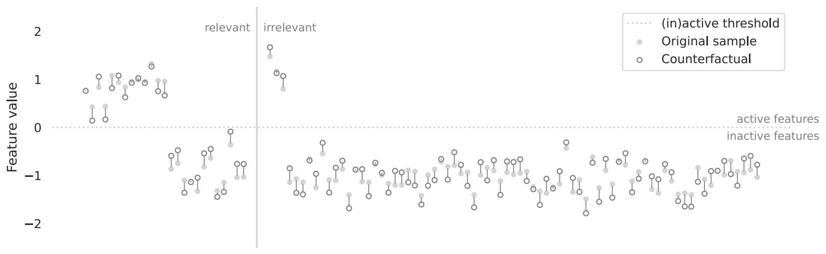
(a) ECINN (tipping-point)



(b) ECINN (convincing)



(c) Wachter (DC, no edges)



(d) Wachter (GC, no edges)

Fig. 5.4 Generated counterfactuals

Counterfactuals generated by the different methods. The y-axis shows each feature value of the original and modified sample. Relevant features (see 4.2.2) are on the left and irrelevant ones are shown to the right. True positives are highlighted in green and false positives in red. Best viewed in color. Wachter counterfactuals in the presence of edges are discussed and visualized in Appendix A.1.

Chapter 6

Discussion

In this chapter, we analyze the results of our experiments. We discuss our findings regarding the efficacy of generative classifiers for classification, and how counterfactuals generated by the methods of ECINN [15] and Wachter et al. [36] compare.

6.1 Generative classifiers for classification

We find that generative classifiers can be used to obtain comparable performance to discriminative classifiers, as shown in Figure 5.2.1. There are some caveats to replacing a discriminative classifier with a generative one, though. For one, we need to train for more epochs to match the discriminative classifier's performance. Additionally, the generative classifier also needs a higher parameter count: in total, the generative classifier has about twice the graph-convolutional layers. This slows training down further.

That more training is needed is to be expected since two tasks are being optimized for simultaneously. The need for a larger model could also be due to our choice of invertible layers (affine coupling layers), and may be remedied by the use of invertible layers with better representational power.

6.2 Counterfactual methods

Next, we discuss how the two counterfactual methods by Hvilshøj et al. [15] and Wachter et al. [36] compare when applied to the same generative classifier.

Only ECINN reliably identifies relevant features, while Wachter's method fails to distinguish between relevant and irrelevant features, which can be seen in the close true and false positive rates. ECINN, on the other hand, finds relevant features, indicated by the very low false positive rate and low magnitude of irrelevant feature changes.

ECINNs are not sparse and tend to modify all relevant features, with larger changes to "deactivate" features than to "activate" them (see 5.2.2b)). The lack of sparsity can also be seen in the true positive rate. That said, the magnitude of irrelevant feature changes is very low compared to the over all magnitude.

Wachter et al. [36]'s method is sensitive to hyperparameters. Even small changes to one of the three parameters λ_{init}, f or η used for optimization can further degrade the quality of generated counterfactuals. While we are able to use the ground truth to finetune these for our task, this would not be possible on a real-world dataset. We think it could prove challenging to find a suitable set of hyperparameters and in turn create meaningful counterfactuals with the method by Wachter et al. [36].

ECINN [15], on the other hand, exposes a single hyperparameter w we can use to control the feature changes of an ECINN. We show in Figure 5.2.1 that varying w provides a predictable control to trade off magnitude and correctness.

ECINN is computationally efficient. We detail in Chapter 3 that ECINN generation has constant complexity, as it always requires a single evaluation of both f and f^{-1}. We find that in our case, generating high-quality Wachter counterfactuals requires, on average, 80 iterations, and each iteration consists of a forward and a backward pass.

One must also consider, though, that ECINN requires a generative classifier to be used, which Wachter et al. [36]'s method does not, and a generative classifier takes longer to train. These considerations in mind, we can say that generative classifers with ECINN are a way to build interpretable models, while black-box methods like Wachter et al. [36]'s let us explain existing models.

Chapter 7

Conclusion

We apply generative classifiers and counterfactual methods to the problem of graph node classification. We show that the normalizing flow-based generative classifiers can be used as an alternative to regular, discriminative classifiers. We study the task of generating node counterfactuals that change the model's predicted class by only modifying a node's features. To do so, we propose a synthetic node classification dataset with ground truth labels of relevant node features that allows us to quantitatively evaluate counterfactuals. We compare two methods for generating counterfactuals: ECINN [15] (Efficient Counterfactuals from Invertible Neural Networks), which uses the generative classifier's structure to efficiently create counterfactuals, and the model-agnostic method proposed by Wachter et al. [36]. We find that counterfactuals with Wachter et al. [36]'s method are not of usable quality, especially in the context of a node with many neighbors. On generative classifiers, ECINN [15] counterfactuals reliably identify relevant features and have a well-structured latent space. ECINN counterfactuals are also efficient to compute, while Wachter counterfactuals require an expensive optimization process with hard-to-tune hyperparameters. To conclude, we find that generative classifiers and ECINNs allow for building interpretable models for graph node classification.

7.1 Future work

The next step to improve the explanation power of ECINN counterfactuals may be to incorporate edges and edge features. Modeling a graph's adjacency as part of the flow is already done in [38], though only on small-scale molecular graphs. Scaling this approach to larger graphs is not straightforward since an adjacency matrix grows with $|V|^2$, though memory bottlenecks could be remedied to some degree by using the memory-efficient backpropagation algorithm [11] that the flow's invertibility allows for. With a different approach, Liu et al. [24] learn a low-dimensional representation of the adjacency matrix with an autoencoder. In any case, successfully tackling this problem would allow us to apply generative classifiers to link prediction and in turn ECINN [15] to create much richer counterfactuals.

The structured latent space afforded by generative classifiers could also be used for more fine-grained control of ECINN [15] counterfactuals. For example, one could search within the known class boundary of the target class for more sparse counterfactuals. If sparse solutions can be found, this could also enable us to find more diverse counterfactuals.

References

[1] Abrate, C. and Bonchi, F. (2021). Counterfactual Graphs for Explainable Classification of Brain Networks. *arXiv:2106.08640 [cs].*

[2] Ardizzone, L., Mackowiak, R., Rother, C., and Köthe, U. (2021). Training Normalizing Flows with the Information Bottleneck for Competitive Generative Classification. *arXiv:2001.06448 [cs, stat].*

[3] Bojarski, M., Del Testa, D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L. D., Monfort, M., Muller, U., Zhang, J., Zhang, X., Zhao, J., and Zieba, K. (2016). End to End Learning for Self-Driving Cars. *arXiv:1604.07316 [cs].*

[4] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. (2020). Language Models are Few-Shot Learners. *arXiv:2005.14165 [cs].*

[5] Dandl, S., Molnar, C., Binder, M., and Bischl, B. (2020). Multi-Objective Counterfactual Explanations. *arXiv:2004.11165 [cs, stat]*, 12269:448–469.

[6] Dinh, L., Krueger, D., and Bengio, Y. (2015). NICE: Non-linear Independent Components Estimation. *arXiv:1410.8516 [cs].*

[7] Dinh, L., Sohl-Dickstein, J., and Bengio, S. (2017). Density estimation using Real NVP. *arXiv:1605.08803 [cs, stat].*

[8] Fey, M. and Lenssen, J. E. (2019). Fast Graph Representation Learning with PyTorch Geometric.

[9] Ghorbani, A., Abid, A., and Zou, J. (2018). Interpretation of Neural Networks is Fragile. *arXiv:1710.10547 [cs, stat].*

[10] Giles, C. L., Bollacker, K. D., and Lawrence, S. (1998). CiteSeer: An automatic citation indexing system. In *Proceedings of the Third ACM Conference on Digital Libraries - DL '98*, pages 89–98, Pittsburgh, Pennsylvania, United States. ACM Press.

[11] Gomez, A. N., Ren, M., Urtasun, R., and Grosse, R. B. (2017). The Reversible Residual Network: Backpropagation Without Storing Activations.

[12] Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative Adversarial Networks. *arXiv:1406.2661 [cs, stat].*

[13] Henter, G. E., Alexanderson, S., and Beskow, J. (2020). MoGlow: Probabilistic and controllable motion synthesis using normalising flows. *ACM Transactions on Graphics*, 39(6):1–14.

[14] Ho, J., Jain, A., and Abbeel, P. (2020). Denoising Diffusion Probabilistic Models. *arXiv:2006.11239 [cs, stat].*

[15] Hvilshøj, F., Iosifidis, A., and Assent, I. (2021). ECINN: Efficient Counterfactuals from Invertible Neural Networks. *arXiv:2103.13701 [cs].*

[16] Jin, W., Barzilay, R., and Jaakkola, T. (2019). Junction Tree Variational Autoencoder for Molecular Graph Generation. *arXiv:1802.04364 [cs, stat].*

[17] Kim, B., Khanna, R., and Koyejo, O. O. (2016). Examples are not enough, learn to criticize! Criticism for Interpretability. In *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc.

[18] Kingma, D. P. and Ba, J. (2017). Adam: A Method for Stochastic Optimization. *arXiv:1412.6980 [cs].*

[19] Kingma, D. P. and Dhariwal, P. (2018). Glow: Generative Flow with Invertible 1x1 Convolutions. *arXiv:1807.03039 [cs, stat].*

[20] Kingma, D. P. and Welling, M. (2014). Auto-Encoding Variational Bayes. *arXiv:1312.6114 [cs, stat].*

[21] Kipf, T. N. and Welling, M. (2016). Variational Graph Auto-Encoders. *arXiv:1611.07308 [cs, stat].*

[22] Kipf, T. N. and Welling, M. (2017). Semi-Supervised Classification with Graph Convolutional Networks. *arXiv:1609.02907 [cs, stat].*

[23] Krizhevsky, A. (2014). One weird trick for parallelizing convolutional neural networks. *arXiv:1404.5997 [cs].*

[24] Liu, J., Kumar, A., Ba, J., Kiros, J., and Swersky, K. (2019). Graph Normalizing Flows. *arXiv:1905.13177 [cs, stat].*

[25] Lu, Q. and Getoor, L. (2003). Link-based Text Classification. page 8.

[26] Mackowiak, R., Ardizzone, L., Köthe, U., and Rother, C. (2020). Generative Classifiers as a Basis for Trustworthy Image Classification. *arXiv:2007.15036 [cs].*

[27] McCallum, A. K., Nigam, K., Rennie, J., and Seymore, K. (2000). Automating the Construction of Internet Portals with Machine Learning. *Information Retrieval*, 3(2):127–163.

[28] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

[29] Selvaraju, R. R., Das, A., Vedantam, R., Cogswell, M., Parikh, D., and Batra, D. (2017). Grad-CAM: Why did you say that? *arXiv:1611.07450 [cs, stat].*

[30] Simonyan, K., Vedaldi, A., and Zisserman, A. (2014). Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps. *arXiv:1312.6034 [cs].*

[31] Sohl-Dickstein, J., Weiss, E. A., Maheswaranathan, N., and Ganguli, S. (2015). Deep Unsupervised Learning using Nonequilibrium Thermodynamics. *arXiv:1503.03585 [cond-mat, q-bio, stat].*

[32] Song, J., Meng, C., and Ermon, S. (2021). Denoising Diffusion Implicit Models. *arXiv:2010.02502 [cs].*

[33] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958.

[34] Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., and Fergus, R. (2014). Intriguing properties of neural networks. *arXiv:1312.6199 [cs].*

[35] Tishby, N., Pereira, F. C., and Bialek, W. (2000). The information bottleneck method. *ArXiv*.

[36] Wachter, S., Mittelstadt, B., and Russell, C. (2017). Counterfactual Explanations Without Opening the Black Box: Automated Decisions and the GDPR. SSRN Scholarly Paper ID 3063289, Social Science Research Network, Rochester, NY.

[37] Yang, Z., Cohen, W. W., and Salakhutdinov, R. (2016). Revisiting semi-supervised learning with graph embeddings. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, pages 40–48, New York, NY, USA. JMLR.org.

[38] Zang, C. and Wang, F. (2020). MoFlow: An Invertible Flow Model for Generating Molecular Graphs. *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 617–626.

[39] Zhang, Z., Bu, J., Ester, M., Zhang, J., Yao, C., Yu, Z., and Wang, C. (2019). Hierarchical Graph Pooling with Structure Learning. *arXiv:1911.05954 [cs, stat].*

[40] Zhao, T., Liu, G., Wang, D., Yu, W., and Jiang, M. (2021). Counterfactual Graph Learning for Link Prediction. *arXiv:2106.02172 [cs].*

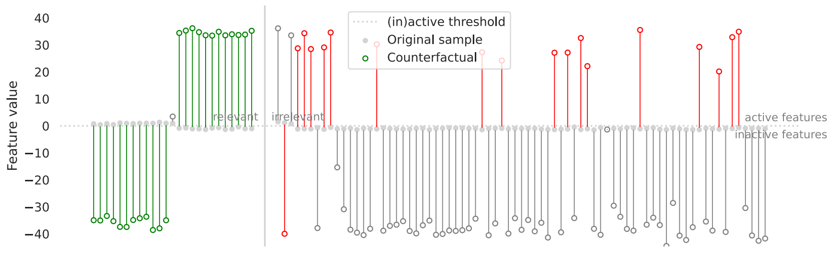
Appendix A

A.1 Wachter node counterfactuals in the presence of edges

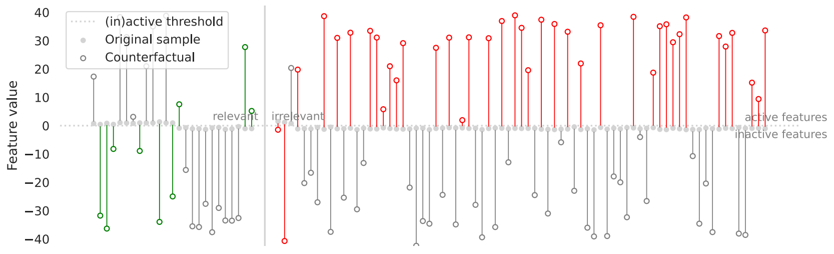
While generating counterfactuals for graph-convolutional classifier, using Wachter's method, we find that the counterfactuals have extremely high magnitudes. Figure A.1 shows an example of such a counterfactual.

We assume this to be the case because the counterfactual is not allowed to change the features of neighboring nodes. Since these nodes will by construction mostly have features indicative of the original class, this makes it very hard to change to model output to a target class.

We control for this by removing all edges from the dataset, retraining the classifier and applying the Wachter method. The resulting counterfactuals are of much lower magnitude and we report the results in chapter 5.



(a) Wachter (DC, with edges)



(b) Wachter (GC, with edges)

Fig. A.1 High-magnitude counterfactuals generated when applying Wachter's method on the synthetic dataset with edges