Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Mathematical Sciences

Arne Rustad

tabGAN: A Framework for Utilizing Tabular GAN for Data Synthesizing and Generation of Counterfactual Explanations

Master's thesis in Applied Physics and Mathematics
Supervisor: Kjersti Aas
June 2022

NTNU
Norwegian University of
Science and Technology

Arne Rustad

tabGAN: A Framework for Utilizing Tabular GAN for Data Synthesizing and Generation of Counterfactual Explanations

Master's thesis in Applied Physics and Mathematics
Supervisor: Kjersti Aas
June 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Mathematical Sciences

NTNU
Norwegian University of
Science and Technology

Abstract

Counterfactual explanations is an emerging method for explaining predictions from black-box models by utilizing "what-if" scenarios. In this thesis, we create a Wasserstein Generative Adversarial Network (WGAN) based tabular data synthesizing framework, tabGAN, and later we modify this WGAN framework to create a model-based counterfactual synthesizer framework, which we call tabGANcf. The counterfactual framework is more a proof-of-concept, while the data synthesizing framework is more complete, with a lot of customization available and default values based on extensive hyperparameter tuning. During the creation of the data synthesizing framework tabGAN, we also create a new type of transformation, which we include as a preprocessing option for the numerical variables in a dataset. The novel transformation is a stochastic version of quantile transformation, which we in this thesis name the Randomized Quantile Transformation. In addition to a regular WGAN implementation, the data synthesizing framework tabGAN also implements a WGAN with a conditional generator inspired by the CTGAN data synthesizer [1]. The conditional architecture and training process aim to provide more representation for rare categories in imbalanced columns.

We compare six data synthesizing methods from the tabGAN framework against the state-of-the-art data synthesizer methods CTGAN [1], TVAE [1], CopulaGAN [2], GaussianCopula [2], and TabFairGAN [3]. The comparison includes an evaluation of the recreated marginal and joint distributions of a real dataset, as well as a comparison of machine learning efficacy on four real datasets. The methods from the tabGAN framework consistently outperform the other data synthesizing methods. Additionally, the methods from the tabGAN framework run substantially faster than the other GAN based data synthesizers in the evaluation, around 3-4 times faster than CTGAN and CopulaGAN for the real dataset used in the training time comparison. The comparison also indicates that the novel transformation method, the Randomized Quantile Transformation, is very beneficial for dataset variables with many repeated values. We visually verify that methods from the tabGANcf framework are able to generate counterfactual explanations that change the predictions of a black-box classifier whilst not making unnecessary changes to the discrete variables. Sparsity of proposed changes in the numerical variables is, however, still an issue. We propose a potential solution for this that can be investigated in future research. We also provide a list of other extensions that can be implemented in future GAN based counterfactual synthesizers.

Sammendrag

Kontrafaktiske forklaringer er en metode for å forklare prediksjoner fra black-box modeller gjennom henvising til alternative lignende virkeligheter der et annet utfall skjer. Denne eksempel-baserte forklaringsmetoden er i ferd med å bli et populært hjelpemiddel innen forklaring av avanserte AI-modeller. I denne avhandlingen presenterer vi et rammeverk for å syntetisere tabelldata ved hjelp av Wasserstein Generative Adversarial Networks (WGAN) som vi navngir tabGAN. I tillegg modifiserer vi dette rammeverket til også å kunne lage modell-baserte kontrafaktiske forklaringer. Det nye rammeverket for generering av kontrafaktiske forklaringer kaller vi for tabGANcf, selv om det i praksis er mer et pilotprosjekt for å synliggjøre at man kan lage kontrafaktiske forklaringer på denne måten. Rammeverket for syntetisering av tabelldata, tabGAN, er imidlertid langt mer komplett. Det tilrettelegger for stor individuell brukstilpasning, og de anbefalte standardverdiene i rammeverket er basert på et omfattende hyperparametersøk. Gjennom arbeidet med rammeverket tabGAN fikk vi ideen til en ny type transformasjon som kan brukes som preprosesseringsmetode for numeriske variabler. Vi kaller den for randomisert kvantiltransformasjon (the Randomized Quantile Transformation), ettersom den er en stokastisk versjon av kvantiltransformasjon. I tillegg til en mer standard WGAN versjon implementerer vi et WGAN med betinget generator, inspirert av datasyntetiseringsmetoden CTGAN [1]. Tanken bak den betingede arkitekturen og spesialtilpassede treningsprosessen er å være bedre i stand til å representere de sjeldne kategoriene for ubalanserte diskrete kolonner, slik at det blir lettere for generatoren å lære seg å gjenskape de også riktig.

Vi sammenligner seks datasyntetiseringsmetoder fra rammeverket tabGAN mot de beste datasyntetiseringsmetodene innen dette feltet, som CTGAN [1], TVAE [1], CopulaGAN [2], GaussianCopula [2] og TabFairGAN [3]. I sammenligningen evaluerer vi hvor godt datasyntetiserne er i stand til å gjenskape både marginal- og simultanfordelinger fra et reelt datasett. I tillegg evaluerer vi hvor godt maskinlæringsmodeller trent på syntetisk data fra hver datasyntetiserer presterer på ett reelt testdatasett, sammenlignet med hvor godt samme modell trent på det originale treningsdatasettet gjør det. Dette gjentar vi for fire forskjellige reelle datasett. Metodene fra tabGAN-rammeverket gjør det konsekvent bedre i disse evalueringene sammenlignet med de andre datasyntetiserermetodene. I tillegg til dette er metodene fra tabGAN-rammeverket raskere å trene enn de andre datasyntetiserne som også er basert på GAN. Resultatene indikerer videre at randomisert kvantiltransformasjon er svært nyttig for numeriske datasettvariable med mange repeterte verdier. For det kontrafaktiske rammeverket tabGANcf utfører vi en visuell verifisering av at metodene fra rammeverket er i stand til å generere kontrafaktiske forklaringer som faktisk endrer prediksjonene til en black-box klassifiseringsmodell. Vi observer at de ulike metodene fra rammeverket fint klarer å endre prediksjonene uten å gjøre unødvendige endringer på diskrete variable, men at dette ikke er tilfellet for de numeriske variablene. I denne avhandlingen foreslår vi en mulig løsning både på dette problemet og andre potensielle utfordringer vi identifiserer. Grunnet tidsbegrensninger overlater vi testing av disse forslagene til fremtidig forskning.

Preface

With this I complete a wonderful five-year period of my life and my Master of Science (M.Sc.) in Applied Physics and Mathematics at the Norwegian University of Science and Technology (NTNU) in Trondheim, where I have first specialized in Industrial Mathematics and later further specialized in statistics. The work leading up to this thesis was conducted in the spring semester of 2022 as part of the course TMA4900; a continuation of the initial work carried out in the the fall of 2021 during my project thesis in the course TMA4500. The initial goal of this combined one year of work was to perform a deep-dive into an exciting new interpretable machine learning tool known as counterfactual explanations and hopefully make a contribution to this up-and-coming field by creating a new model-based counterfactual synthesizer. Whilst working toward this goal I created a Generative Adversarial Network (GAN) based data synthesizing method, which I excitingly enough found was more than competitive with current state-of-the-art data synthesizers. I therefore slightly shifted the focus of my work into additionally creating a full framework for data synthesizing. The process of creating this framework inspired a new numerical preprocessing transformation, which I have named the Randomized Quantile Transformation. Working on and writing this master thesis has been a very rewarding experience, although of course demanding at times, and I am proud to present this as the final work of my master's degree here at NTNU.

I would like to extend a huge thanks to my supervisor Kjersti Aas for excellent guidance during this last year. Additionally, I would like to thank my family, friends and girlfriend for their constant moral support. I would especially like to thank those who took time out of their busy days to help me proofread this work.

Contents

Chapter 1

Introduction

Machine learning models have exploded in popularity over the last decade [4] and are being widely adopted across industrial sectors [5]. This includes low-stake applications, such as personalized recommendations for what you should watch next on Netflix or which country is going to win the next football world championship, but also high-stake domains, such as medical diagnosis and policy making [5]. As more and more decision tasks are automated by machine learning models, often even without human input [6], it becomes increasingly crucial to have tools to scrutinize models and ensure that they are working as we intend them to. This is especially important for black box models such as neural networks, gradient boosting trees, random forest, etc. Even an expert in the field will not understand much about how a neural network reasons for a specific task by looking at the possibly millions of parameters that are fitted. This is where interpretable machine learning (explainable AI) methods come in: a toolbox for understanding machine learning models or specific predictions.

The interpretable machine learning field includes methods for feature attribution of a single prediction, feature attribution of whole machine learning models, fitting surrogate models that are easier to interpret, and visualization tools for how predictions change as input features change, to name a few [7, 8]. An emerging interpretable machine learning method that has gained attention in recent years is counterfactual explanations [5]. Karimi *et al.* [9] describe a counterfactual as "*a statement of how the world would have to be different for a desirable outcome to occur*". Multiple counterfactuals for the same situation are of course possible since there may be multiple desirable outcomes or multiple ways to achieve each of these outcomes.

A common example used to describe how counterfactual methods can be utilized is the case of a person applying for a loan at a bank where the bank uses a black box model to automatically decide who gets a loan based on information about the person applying and the loan requested. Assume the loan application was rejected. Currently, the EU General Data Protection Regulation (GDPR) includes in the guideline the right to request "*meaningful information about the logic involved, as well as the significance and the envisaged consequences*" [10]. Although this is not currently a requirement [10], one way that the bank in question could satisfy this guideline would be to give counterfactual explanations to the person getting a loan rejection. To exemplify, we here list some possible counterfactual explanations suitable for this case. If the person had fifty thousand less in previous debt, then the loan request would have been approved. If the person had two fewer records of non-payment, then the loan request would have been approved. If the person had applied for a smaller loan of size two hundred thousand instead of three hundred thousand, then the loan request would have been approved.

As seen above, counterfactuals can be used as an explanation for why a specific desired outcome was not reached and can also be used as a guide on how to reach the desired outcome in the

future. They are, however, also directly useful for machine learning practitioners to evaluate the robustness or fairness of their model [11]. A closely related method to counterfactual explanations is adversarial perturbations. Adversarial perturbations are small changes to the input of a machine learning model that change the predicted outcome [6]. However, adversarial pertubations are more about fooling the predictor whilst counterfactuals focus more on interpreting the model or explaining an individual prediction. Initial counterfactual methods resembled adversarial pertubations [5], but later methods also optimize for other objectives that benefit counterfactual explanations, such as being sparse for easy interpretation and being plausible. Counterfactual explanations are a lot less intuitive and useful for reasoning if they are outside the data distribution [5]. Another closely related method to counterfactual explanations is actionable recourse [12]. Actionable recourse is very similar to counterfactuals, but for actionable recourse the focus is generating actionable examples to receive a more favourable outcome, instead of making predictions interpretable.

Counterfactual methods can be either model-specific or model-agnostic [7]. The former usually utilizes the internal structure of a specific machine learning model (for instance the trained weights) to generate explanations, while model-agnostic methods work for arbritary machine learning methods. Model agnostic counterfactual methods often treat the model as a black box and only utilizes the model input and output. Another way to separate counterfactual explanation methods into categories is whether they are algorithmic-based or model-based methods [5]. Fundamentally, these two categories have different views on how to solve the task of generating counterfactuals. Algorithmic-based counterfactual methods treat the task as an optimization problem, while model-based methods focus on first using a model to learn the data distribution and then use the model to generate counterfactuals. Although algorithmic counterfactual methods can also optimize for plausible explanations, model-based methods generally achieve plausibility automatically through the model.

The aim of this thesis is to create a model-based counterfactual method for tabular data utilizing generative adversarial networks (see Section 2.2). We believe an intuitive way of approaching this task is to first either create or select a Generative Adversarial Network (GAN) based data synthesizer to use as a base and then modify it to suit the task of generating counterfactual explanations. In this thesis, we choose to develop a GAN-based data synthesizer, but take inspiration from a range of papers on data synthesizing [1, 3, 13]. As the performance of a data synthesizer is perhaps more easily measured with a single metric compared to a counterfactual method, we spend much effort on creating a well-performing data synthesizer, hoping that the quality will at least partially carry over to the counterfactual generator. To this end, we investigate the benefit of different preprocessing methods, model architectures, and combinations of hyperparameters. We create a framework that allows for a very flexible definition of GAN methods that we employ to perform fairly extensive hyperparameter tuning. The process of creating the tabular GAN inspired a new transformation for numerical data, which we in this thesis name the Randomized Quantile Transformation (QTR).

The creation of the tabular data synthesizer has significant value in itself. A Gartner[1] 'Predict' published July 2021 in the Wall Street Journal states that "By 2024, 60% of the data used for the development of AI and analytics projects will be synthetically generated"[2]. Furthermore, according to a recent survey by the Kaggle data science platform, tabular data is the most common type of data encountered in business and the second most common type encountered in academia [13]. Privacy concerns often prevent enterprises from obtaining outside help and even within companies data sharing is not always permitted or unproblematic [2]. Privacy is also an issue in other instances. For example, sharing of medical data is strictly regulated, and applications for access can be a very

[1] An American consultant firm

[2] Link to Wall Street Journal article behind paywall: `https://www.wsj.com/articles/fake-it-to-make-it-companies-beef-up-ai-models-with-synthetic-data-11627032601`

slow process. Data synthesizers can be a useful tool for alleviating these issues, as the sharing of synthetic data avoids many privacy concerns. If these synthetic datasets follow the original data distribution, then they can replace the usage of the original datasets altogether or be used as a quick start option while waiting for access to the original dataset. In a crowd-sourcing experiment, where experienced data scientists were hired to write complex features for a dataset that were to be applied to a machine learning model, no significant difference was found between the groups that got synthetic datasets compared to those that got the original dataset [2], thus indicating that data scientists can potentially be as productive with synthetic datasets as with original data. Synthetic data can also be used as a low effort course of action for a researcher or consultant to verify that a method has potential before applying for data access, or for a business to evaluate an outside partner before deciding to share the original data. Sharing of synthetic data has the added benefit of decreasing the chance of the original data being compromised in case of an accidental data breach for outside partners. Additionally, synthetic data can be utilized for initial training of a model in the time before the real data is accessible. When the real dataset becomes available, the model does not have to start from scratch, but instead has a starting point. Another application of synthetic data is to resolve critical bottlenecks for machine learning methods [13], such as missing data or class imbalances. Here imputation methods can be used to fill in the missing values or a data synthesizer can try to learn the underlying data distribution and then generate datasets without missing values. For the class imbalance problem, data synthesis can be used to generate more samples of the rare classes.

The thesis is organized as follows. In Chapter 2 we present some background theory helpful for understanding the rest of the thesis. Next, in Chapter 3 we introduce data synthesis in a bit more detail and give a simple overview of different methods for synthesizing a single tabular dataset. We proceed to give a detailed description of the framework introduced in this thesis for tabular data generation, as well as a short description of specific state-of-the-art methods we will use as a basis for comparison when later evaluating the performance of the new data synthesis framework. In Chapter 4 we give a quite thorough explanation of quantile transformation along with some examples, before we introduce the new modification of quantile transformation called Randomized Quantile Transformation (QTR). We present the motivation for creating the QTR and mathematically describe a specific implementation of it. In Chapter 5 we present the counterfactual method based on the data synthesizing framework previously introduced. As a foundation, we first give a quite thorough explanation of counterfactual explanations with a focus on model-based counterfactual methods. In the end of the chapter we explain some possible extensions to the counterfactual framework currently implemented. In Chapter 6 we present the datasets later used in the experiments in this thesis. In Chapter 7 we evaluate the performance of some of our data synthesizers against state-of-the-art data synthesizers. Next, we investigate and perform some simple experiments with our new counterfactual method in Chapter 8. Finally, in Chapter 9 we discuss the overall results and conclude how the methods introduced in this paper perform compared to the state-of-the-art methods. This is also where we give an outlook on what future related investigations can entail. This master thesis is a partial continuation of a project thesis [14] we wrote the fall of 2021. Although much of the text from the project thesis is discarded or rewritten, there will be some borrowed sentenced or paragraphs here and there, especially for Chapter 1 and Sections 2.2, 5.1, 5.2 and 7.4. In the project thesis, we created a model based counterfactual synthesizer by combining a data synthesizer with Monte Carlo sampling and a post-processing step. We compared it to the algorithmic based counterfactual method MOC [11], which utilize an evolutionary algorithm to simultaneously optimize multiple objectives. For the master thesis we chose to concentrate on model based counterfactual methods and data synthesizers. We spent effort on creating a more advanced data synthesizer and later modified it to directly synthesize counterfactual explanations,

thus eliminating the need for Monte Carlo sampling and a post-processing step.

Chapter 2

Background

In this chapter, we present some theory that we believe is relevant or necessary for understanding the rest of this thesis. It can easily be skipped if the reader is already familiar with these topics. As we in this thesis both create a data synthesizer and a counterfactual method based on Generative Adversarial Networks (GAN), we believe it is relevant to include theory on neural networks and GAN. In Section 2.1, we present a thorough introduction of neural networks. We begin with a single neural network node and expand the explanation to multi-layer dense neural networks with activation functions. Together with the explanation we also include some comments about the biological inspiration and some history notes. In Section 2.1.1, we present an informal list of possible activation functions. Then, in Section 2.1.2, we briefly explain how neural networks are trained and some possible choices of loss functions. Neural networks are trained using optimization algorithms. As we in this thesis use gradient-based optimization algorithms to train the neural networks, we give in Section 2.1.2 a list of relevant optimization algorithms. The list and the explanation is structured in such a way that they partially explain the reason why the Adam optimization algorithm is the default choice in the tabGAN framework which we present in this thesis. One of the main reasons we are able to train neural networks efficiently is that the backpropagation algorithm allows for efficient calculation of the gradient for all the parameters in the model. In Section 2.1.2, we present the backpropagation algorithm. Next, in Section 2.1.3, we present batch normalization and layer normalization, both of which are optional techniques used in the tabGAN framework. Then, in Section 2.2, we explain Generative Adversarial Networks. We also present in Section 2.2.1 a modification of GAN known as Wasserstein GAN, which is the type of GAN implemented in this thesis.

Later in this thesis, in Chapter 7, we evaluate the data synthesizers created in this thesis against state-of-art data synthesizers. One of the methods we use to compare the performance of a data synthesizer is machine learning efficacy. That is, we observe how a machine learning model trained on a synthesized dataset perform on a real test dataset never seen by the data synthesizer. In this thesis we use the gradient boosting trees framework, XGBoost, as the machine learning model. Therefore, we give in Section 2.3 a brief introduction to XGBoost. To evaluate the performance of the machine learning model we need to specify some metrics. In Section 2.4 we present some common metrics used for classification models. We only list classification metrics since classification tasks are the only machine learning problems considered in this thesis.

2.1 Neural Networks

Neural networks are machine learning algorithms loosely inspired by how the human brain functions. Similar to how the brain consists of interconnected neurons that can transmit signals between

them, a neural network consists of nodes that are interconnected. For each incoming connection, the node will assign a separate weight. Then, it adds the weighted incoming signals together, thus yielding a single value that is passed to other nodes through its outgoing connections. For a single node $\hat{y}_j$ receiving input from a multiple of nodes $\mathbf{x} = [x_1, \dots, x_n]^T$ and weight values $\mathbf{w}_j = [w_{1,j}, \dots, w_{n,j}]^T$, this can mathematically be written written as

$$\hat{y}_j = \sum_{i=1}^{n} w_{i,j} x_i = \mathbf{w}_j^T \mathbf{x}. \tag{2.1}$$

We note that this is merely a linear transformation of $\mathbf{x}$. In the human brain, the connection equivalent known as synapses only "fire" and release a signal if the signal from the neuron with outgoing signal is strong enough, that is, if the signal is stronger than some threshold. Inspired by this, neural networks often employ activation functions that have a similar purpose. Activation functions have the added benefit of introducing non-linearity to the neural network. Perhaps the activation function most resembling to how we have described the functionality of the synapses, would be the binary step size function, i.e. an indicator function of whether x is larger than some threshold b.

$$a_{bs}(x) = \begin{cases} 1 & \text{if } x \geq b \\ 0 & \text{if } x < b. \end{cases}$$

However, the gradient of this function is 0 everywhere, except at $x = b$ where it is not defined. This makes gradient-based learning impossible, which a large share of neural networks use. Furthermore, we note that the inclusion of the threshold b could instead be achieved by adding a bias node b_j to the weighted sum in Equation (2.1), thus making the threshold learnable. An activation function mimicking the threshold "firing" procedure whilst still being differentiable everywhere, is the sigmoid function

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

The sigmoid function is close to zero for large negative x values, equal to 0.5 for $x = 0$ and close to one for large positive x values. Adding a sigmoid activation function and a bias node to Equation (2.1), we get

$$\hat{y}_j = \sigma(b_j + \sum_{i=1}^{n} w_{i,j} x_i) = \sigma(b_j + \mathbf{w}_j^T \mathbf{x}). \tag{2.2}$$

There are many activation functions to choose from, and moving forward we will use $a(\cdot)$ to denote any activation function. A selected sample of activation functions are described in Section 2.1.1. Until now, we have merely described how a single node receives and "processes" input from nodes connected to it. In fact, the first trainable neural network, the perceptron [15], only consisted of a input layer, a hidden node and an output node. A hidden node refers to the fact that the node is neither an input node nor an output node. Although the original perceptron used a binary step size activation function for the hidden node, it can easily be replaced by any other activation function. Many problems can be solved by simply using this structure. For instance, it can become a linear model by using identity activation both for the hidden node and the output node, or it can become a logistic regression model by using identity activation for the hidden node and sigmoid activation function for the output node. However, for more complex problems, more nodes are needed. A natural extension to the perceptron architecture is to increase the number of

hidden nodes and/or the number of hidden layers. This architecture is called feedforward neural networks or multilayer perceptrons (MLP). More mathematically, this structure can be defined as

$$\begin{aligned}\hat{\mathbf{y}} &= a_{(o)}\left(\mathbf{z}^{(o)}\right)\\ &= a_{(o)}\left(\mathbf{b}^{(o)} + \mathbf{W}^{(o)}\mathbf{a}^{(H)}\right)\\ &= a_{(o)}\left(\mathbf{b}^{(o)} + \mathbf{W}^{(o)}a_{(H)}\left(\mathbf{z}^{(H)}\right)\right)\\ &= a_{(o)}\left(\mathbf{b}^{(o)} + \mathbf{W}^{(o)}a_{(H)}\left(\mathbf{b}^{(H)} + \mathbf{W}^{(H)}\mathbf{a}^{(H-1)}\right)\right)\\ &\vdots\\ &= a_{(o)}\left(\mathbf{b}^{(o)} + \mathbf{W}^{(o)}a_{(H)}\left(\mathbf{b}^{(H)} + \mathbf{W}^{(H)}a_{(H-1)}\left(\ldots\left(a_{(1)}\left(\mathbf{b}^{(1)} + \mathbf{W}^{(1)}\mathbf{x}\right)\right)\right)\right)\right),\end{aligned} \tag{2.3}$$

where $\hat{\mathbf{y}}$ is the output layer after applying activation function $a_{(o)}$, $\mathbf{z}^{(o)}$ is the output layer before applying the activation function, $\mathbf{x}$ is the input layer, $\mathbf{a}^{(h)}$ is the hidden layer number h after applying activation function $a_{(h)}$ for $h = 1,\ldots,H$, $\mathbf{z}^{(h)}$ is the hidden layer number h before applying activation for $h = 1,\ldots,H$, $\mathbf{b}^{(o)}$ is the bias node vector for the output layer, $\mathbf{b}^{(h)}$ is the bias node for hidden layer h for $h = 1,\ldots,H$, $\mathbf{W}^{(o)}$ is a matrix containing the weights connecting the last hidden layer to the output layer, $\mathbf{W}^{(h)}$ is a matrix containing the weights connecting hidden layer $h-1$ to layer h for $h = 2,\ldots,H$ and $\mathbf{W}^{(1)}$ is a matrix containing the weights connecting the input layer to the first hidden layer. A weight matrix $\mathbf{W} = [\mathbf{w}_1,\ldots,\mathbf{w}_i,\ldots\mathbf{w}_n]^T = \left[\left\{w_{i,j}\right\}_{\substack{i=1,\ldots,n\\ j=1,\ldots,m}}\right]$ is defined such that row i contains the weights connecting node i in the current layer to all the nodes in the previous layer. Here n is the length of the current layer, while m is the length of the previous layer. A bias vector $\mathbf{b}$ is of the same length as the layer it is connected to, and each bias node is connected to a single, unique node in the other layer.

There are a number of more advanced types of neural networks, such as convolution neural networks (CNN), recurrent neural networks (RNN) and transformer neural networks to name a few. As the tabGAN framework mainly uses combinations of feedforward neural network structures, we do not elaborate on any of these other model architectures. We end this part of the section with a motivation for why even standard feedforward neural network structures are really interesting. The universal approximation theorem states that under very mild conditions, any continuous function can be uniformly approximated by a feedforward neural network with only a single hidden layer [16]. This is, of course, mostly a theoretical result, as the hidden layer might have to be of infinite size, but it still speaks to the universality of neural networks.

2.1.1 Activation Functions

We have already introduced the binary step size activation function and the sigmoid activation function. However, there are a myriad of different possible activation functions. In this subsection, we will present another really famous activation function, ReLU, along with some other activation functions used in the work leading up to this thesis. We stress that this is far from a complete list, but instead an informal list of activation functions considered or used by the authors in the making of the tabGAN framework.

The rectified linear unit activation function, ReLU, [17] is defined as

$$ReLU(x) = \max\{0, x\} = \begin{cases} x & \text{if } x > 0\\ 0 & \text{if } x \leq 0.\end{cases}$$

The ReLU activation function is easily calculated and its substitution of the sigmoid activation function in the hidden layers greatly improved the performance of feedforward neural networks. It

largely solved the problem with vanishing gradients in deep (multi-layered) neural networks [18]. However, ReLU still has some limitations. Perhaps most severe is the case where updates to the node weights can result in the input to the activation function always being negative, regardless of what is given as input to the neural network. This is called node death. "Dead" nodes might never activate again since the gradient through them is always zero and thus gradient based optimization algorithms will not adjust the weights of the "dead" unit. Similarly to the problem of vanishing gradients, we might expect learning to be slow when training ReLU networks with constant gradients of zero [19]. Several extensions have been proposed to counteract this problem.

The LeakyReLU activation function [19] allows for a small gradient when the node is inactive.

$$LeakyReLU_\alpha(x) = \max\{x, \alpha x\} = \begin{cases} x & \text{if } x \geq 0 \\ \alpha x & \text{if } x < 0. \end{cases}$$

Here $\alpha < 1$ is some parameter that can be chosen. The exponential linear unit, ELU, [20] keeps the linear part of the ReLU function for $x > 0$, but uses a parameterized exponential function that quickly converges to $-\alpha$ for negative x values. The parameter α controls the smoothness of the transition.

$$ELU_\alpha(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha(e^x - 1) & \text{if } x < 0. \end{cases}$$

A modification of ELU that attempts to incorporate batch normalization (see Section 2.1.3 for a definition) into the activation function is the scaled exponential linear unit activation function, SELU [21]. It is defined as

$$SELU_\alpha(x) = \begin{cases} \lambda x & \text{if } x \geq 0 \\ \lambda\alpha(e^x - 1) & \text{if } x < 0, \end{cases}$$

where $\lambda \approx 1.0507$ and $\alpha \approx 1.6733$. Another proposed modification of RELU is the Swish [22] activation function proposed by the Google Brain Team. It is also known as the Sigmoid Linear Unit activation function, SiLU. The method is kind of a compromise between ReLU and sigmoid, and it is defined as

$$Swish(x) = x\sigma(x) = \frac{x}{1 + e^{-x}}.$$

Another activation function is Mish [23], which the authors describe as a self-regularized non-monotonic activation function. The original paper propose an intuitive explanation for why the first derivative of the activation function may be acting as a regularizing element in the optimization of a deep neural network. It is mathematically defined as

$$Mish(x) = x\tanh(softplus(x)) = x\tanh\left(\ln\left(1 + e^x\right)\right).$$

Another modification is the Gaussian error linear unit activation function, GELU [24]. We will present this activation function in a bit more detail because it is the default choice in the tabGAN framework and its derivation partially explains the intention behind the activation function. The GELU activation function combines ReLU with properties of dropout [25] and zoneout [26]. Dropout is a regularization method used during training of a neural network where some nodes are randomly ignored, whilst zoneout is a regularization method for recurrent neural networks where a node's activation function is randomly set to be the activation from the previous time step. The authors of Hendrycks and Gimpel [24] seek to merge this functionality by stochastically multiplying the input by zero or one (note that input here does not mean the input of the neural network,

but rather, more generally, the layer on which the GELU activation function is used). They let the probability of this zero-one mask depend upon each input. More specifically, for each neuron input x they multiply it by a $Bernoulli(\Phi(x))$ distributed value m, where Φ is the cumulative distribution function of the standard normal distribution. To arrive at a deterministic version of this, the authors of Hendrycks and Gimpel [24] take the expectation, i.e

$$\mathrm{E}[mx] = \Phi(x) \cdot 1x + (1 - \Phi(x)) \cdot 0x = x\Phi(x).$$

Thus, the GELU activation function is defined as

$$\begin{aligned} GELU(x) &= xP(X \leq x) = x\Phi(x), && \text{if } X \sim \mathcal{N}(0,1) \\ &= x \cdot \frac{1}{2}\left[1 + erf(x/\sqrt{2})\right] \\ &= x \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}. \end{aligned}$$

This can, if needed, be approximated by

$$GELU(x) \approx 0.5x\left(1 + \tanh\left[\sqrt{2/\pi}\left(x + 0.44715x^3\right)\right]\right).$$

Hitherto in this subsection, we have listed activation functions generally applied on the hidden layer nodes. However, activation functions are sometimes also applied to the output node or layer. For example, in the case of a binomial classification problem, the output is often a probability, and therefore should lie in the interval between 0 and 1. The sigmoid activation function is commonly applied to such situations, as it maps the space $[-\infty, \infty]$ to the space $[0, 1]$. For the multinomial classification problem where classes are mutually exclusive, the output is generally a layer of probabilities. The sigmoid activation function is not very appropriate for this case, as it does not ensure that the estimated probabilities in the output layer sum up to 1. Instead, the Softmax activation function is commonly used

$$Softmax(\mathbf{x}) = \left[\left\{\frac{e^{x_i}}{\sum_{k=1}^{n} e^{x_k}}\right\}_{i=1,\ldots,n}\right]^T. \tag{2.4}$$

The softmax activation function can also be used as an approximation (with defined gradients) to the arg max function.

2.1.2 Training Procedure

If we are to improve a neural network, it is essential that we have a way to measure the performance of a neural network, otherwise, it is difficult to know how to change it for the better. This is where a loss function comes in (or alternatively, a measurement of how well the neural network performs). A loss function is a measurement of how poorly a neural network performs. To train a neural network, we merely need to change the weights of the neural network so that the loss function is minimized. There are many loss functions and no strict rules for which to use when. However, a common loss function for the case of prediction of a continuous response variable is the mean squared error (MSE)

$$MSE = \frac{1}{n}\sum_{i}^{n}(y_i - \hat{y}_i)^2,$$

where y_i is the response variable for observation i, $\hat{y}_i$ is the prediction of the neural network for observation i, and there are n observations. For the case of a binomial classification problem a common loss function is cross entropy loss

$$CE = \sum_{i=1}^{n} y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i).$$

Cross entropy is also a common loss function for multinomial classification problem

$$CE = \sum_{i=1}^{n} \sum_{j=1}^{K} y_{i,k} \log(\hat{y}_{i,k}),$$

where $[y_{i,1}, \ldots, y_{i,k}]$ are the true class probabilities for observation i, whilst $[\hat{y}_{i,1}, \ldots, y_{i,k}]$ are the corresponding estimated class probabilities for observation i.

Optimization Algorithms

There are many optimization algorithms available for minimzing a loss function. Some are gradient-based; others optimize the loss function without using the gradient. These are not specific optimization algorithms for neural networks, but general algorithms that can be used to optimize many functions. We will now present a selection of first-order (gradient-based) methods. We will not present any second-order methods such as Newton's method, as they are generally computationally infeasible for neural networks with potentially millions or billions of parameters. The next couple of paragraphs will generally be based on Ruder [27] and are structured in such a way that they partially explain the reason why the Adam optimization algorithm is the default choice in the tabGAN framework.

Perhaps the simplest gradient-based optimization algorithm is gradient descent. If we let $\boldsymbol{\theta}$ be the learnable parameters of a neural network and J be the loss as a function of $\boldsymbol{\theta}$ measured on the whole dataset, then the gradient descent algorithm can be written as

$$\boldsymbol{\theta}^{(t)} = \boldsymbol{\theta}^{(t-1)} - \eta \cdot \nabla_{\boldsymbol{\theta}^{(t-1)}} J\left(\boldsymbol{\theta}^{(t-1)} | \mathbf{X}, \mathbf{y}\right),$$

where $\boldsymbol{\theta}^{(t)}$ are the parameters at iteration t of the optimization algorithm and η is a hyperparameter called the learning rate that controls how fast the algorithm changes the parameter estimates. Some disadvantages of this algorithm are that it may become trapped at a local minimum and that if the dataset is huge, it will be quite memory intensive and possibly slow to calculate the gradient on the whole dataset.

A variant called stochastic gradient descent tries to remedy some of these disadvantages. Instead of calculating the loss for the whole dataset $\{\mathbf{X}, \mathbf{Y}\}$ for each step before taking the gradient, the stochastic gradient descent version calculates for each step only the loss based on a single observation $\{\mathbf{x}_i, y_i\}$. This gives much more frequent updates and consequently it can converge in less time. It also requires much less memory and has less chance of getting stuck at a local minima. However, this version also has its disadvantages. It has a high variance in the estimation of the model parameters $\boldsymbol{\theta}$ and even after reaching the global minima it may continue to overshoot. A compromise between these two methods is the mini-batch gradient descent algorithm. With this algorithm, the dataset is divided into batches, and the parameters $\boldsymbol{\theta}$ are updated after calculating the loss function gradient for each of these batches.

Mini-batch gradient descent also has a few disadvantages, many of which it shares with the other gradient descent algorithms mentioned above. Choosing a suitable learning rate can be difficult, as a too small learning rate leads to very slow convergence, while a too large learning rate can both

hinder convergence and cause fluctuations around the minimum. In some cases, a too high learning rate might also cause divergence. Additionally, the same learning rate applies to all parameter updates, which might not be optimal. Another challenge is that the gradient descent algorithms described above have been shown to struggle to escape from saddle points, which usually have a plateau of the same loss function value and therefore the gradient is close to zero in all dimensions.

To overcome the problem of not escaping local minima or saddle points, a few algorithms propose to add a momentum term that helps the algorithm. The Momentum algorithm [27] helps dampen oscillation as well as accelerate mini-batch stochastic gradient descent in the correct direction by adding a fraction γ of the previous update step to the current update step. It is given by

$$\begin{aligned}\mathbf{v}^{(t)} &= \gamma \cdot \mathbf{v}^{(t-1)} + \eta \cdot \nabla_{\boldsymbol{\theta}^{(t-1)}} J(\boldsymbol{\theta}^{(t-1)}|\mathbf{X}, \mathbf{y}) \\ \boldsymbol{\theta}^{(t)} &= \boldsymbol{\theta}^{(t-1)} - \mathbf{v}^{(t)}.\end{aligned}$$

The idea behind the momentum term can be captured by a ball rolling down a hill. If the ball encounters a small flat period or a small upward slope (a saddle point or a local minima), the momentum it has collected will help it to not get stuck. However, if it were to reach the actual lowest point, then this momentum term would propel it further and perhaps upwards the hill again. Ideally, we want a "smarter" ball that has a notion of the terrain it is traveling in and, therefore, slows a bit down when the slope straightens out. To not make this list of optimizers too long, we will now skip ahead to the Adam optimizer algorithm. The Adam algorithm [28] takes into account how the slope changes and also individually adapts the learning rate to each parameter. The Adam algorithm is defined as

$$\begin{aligned}g_j^{(t)} &= \frac{\partial}{\partial \theta_j^{(t-1)}} J(\boldsymbol{\theta}^{(t-1)}|\mathbf{X}, \mathbf{y}) \\ m_j^{(t)} &= \beta_1 m_j^{(t-1)} + (1-\beta_1) g_j^{(t)} \\ v_j^{(t)} &= \beta_2 v_j^{(t-1)} + (1-\beta_2)(g_j^{(t)})^2 \\ \hat{m}_j^{(t)} &= \frac{m_j^{(t)}}{1-\beta_1^t} \\ \hat{v}_j^{(t)} &= \frac{v_j^{(t)}}{1-\beta_2^t} \\ \theta_j^{(t)} &= \theta_j^{(t-1)} - \frac{\alpha}{\sqrt{\hat{v}_j^{(t-1)}} + \epsilon} \hat{m}_j^{(t)},\end{aligned}$$

where $\theta_j^{(t)}$ is the jth parameter at time step t and $g_j^{(t)}$ is the partial derivative of the loss function with respect to $\theta_j^{(t-1)}$ evaluated at $\boldsymbol{\theta}^{(t-1)}$. The parameters β_1 and β_2 are hyperparameters that determine the weight decay in the estimators for respectively the first order moment (the mean), $m_j^{(t)}$, and the second order moment (the uncentered variance), $v_j^{(t)}$, of the gradient respectively. Since $m_j^{(t)}$ and $v_j^{(t)}$ are initialized as zero, they are biased towards zero. The bias-corrected estimates are $\hat{m}_j^{(t)}$ and $\hat{v}_j^{(t)}$. The parameter α represents the learning rate, while ϵ is a small number to avoid division by zero. The authors of Kingma and Ba [28] propose default values $\beta_1 = 0.9$ and $\beta_2 = 0.999$. They also show empirically that the Adam algorithm performs favourably compared to other first order gradient-based optimization algorithms.

Backpropagation

Until now we have not discussed how the gradients are calculated, merely how they are used in an optimization algorithm to train the neural network. Calculating the gradient of a neural network efficiently is, however, no small task, as there can be millions or billions of learnable parameters to calculate the derivative for. Thankfully, a very efficient method called backpropagation [29] has been developed for calculating the gradient of the parameters of a neural network. For a feedforward neural network, the gradient is calculated for each layer separately and the chain rule is used to avoid excess computations. We will use the notation defined for Equation (2.3).

First, a forward pass through the neural network is performed. For simplicity, we assume a single input observation $\mathbf{x}$. A forward pass consists simply of using the current values for the connection weights and bias nodes (the learnable parameters $\boldsymbol{\theta}$) to calculate the values for each hidden layer $\mathbf{z}^{(h)}$, the output layer $\hat{\mathbf{y}}$ and the loss function $J(\boldsymbol{\theta}|\mathbf{x}, y)$. Next comes the backpropagation step. Note that when calculating the derivative of $J(\boldsymbol{\theta}|\mathbf{x}, y)$ with respect to $\mathbf{W}^{(l)}$ or $\mathbf{b}^{(l)}$ it can be split into a product using the chain rule.

$$
\begin{aligned}
\frac{\partial J(\boldsymbol{\theta}|\mathbf{x}, y)}{\partial \mathbf{W}^{(l)}} &= \frac{\partial J(\boldsymbol{\theta}|\mathbf{x}, y)}{\partial \mathbf{z}^{(l)}} \frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{W}^{(l)}} \\
&= \begin{cases} \frac{\partial J(\boldsymbol{\theta}|\mathbf{x}, y)}{\partial \mathbf{z}^{(l)}} \mathbf{x}^T & \text{if } l = 1 \\ \frac{\partial J(\boldsymbol{\theta}|\mathbf{x}, y)}{\partial \mathbf{z}^{(l)}} \left[\mathbf{a}^{(l-1)}\right]^T & \text{if } l = 2, \ldots, H \\ \frac{\partial J(\boldsymbol{\theta}|\mathbf{x}, y)}{\partial \mathbf{z}^{(l)}} \left[\mathbf{a}^{(l)}\right]^T & \text{if } l = o \end{cases} \\
\frac{\partial J(\boldsymbol{\theta}|\mathbf{x}, y)}{\partial \mathbf{b}^{(l)}} &= \frac{\partial J(\boldsymbol{\theta}|\mathbf{x}, y)}{\partial \mathbf{z}^{(l)}} \frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{b}^{(l)}} \\
&= \frac{\partial J(\boldsymbol{\theta}|\mathbf{x}, y)}{\partial \mathbf{z}^{(l)}}.
\end{aligned}
$$

Hence, all the partial derivatives of interest can be computed easily given $\frac{\partial J(\boldsymbol{\theta}|\mathbf{x}, y)}{\partial \mathbf{z}^{(l)}}, l = 1, \ldots, H, o$. As we shall see below, the backpropagation step gives an efficient way to calculate these derivatives. The partial derivative of the loss function with respect to the output layer (before applying the output activation function), $\mathbf{z}^{(o)}$, can be calculated as

$$
\frac{\partial J(\boldsymbol{\theta}|\mathbf{x}, y)}{\partial \mathbf{z}^{(o)}} = \frac{\partial J(\boldsymbol{\theta}|\mathbf{x}, y)}{\partial \hat{\mathbf{y}}} \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{z}^{(o)}} = \frac{\partial J(\boldsymbol{\theta}|\mathbf{x}, y)}{\partial \hat{\mathbf{y}}} * a'_{(o)}\left(\mathbf{z}^{(o)}\right),
$$

where $*$ denotes elementwise multiplication. Having found an expression for the partial derivative of the loss function with respect to the output layer $\mathbf{z}^{(o)}$, the rest of the partial derivatives $\frac{\partial J(\boldsymbol{\theta}|\mathbf{x}, y)}{\partial \mathbf{z}^{(l)}}$, $h = 1, \ldots, H$ can be found recursively.

$$
\begin{aligned}
\frac{\partial J(\boldsymbol{\theta}|\mathbf{x}, y)}{\partial \mathbf{z}^{(h)}} &= \frac{\partial J(\boldsymbol{\theta}|\mathbf{x}, y)}{\partial \mathbf{z}^{(h+1)}} \frac{\partial \mathbf{z}^{(h+1)}}{\partial \mathbf{a}^{(h)}} \frac{\partial \mathbf{a}^{(h)}}{\partial \mathbf{z}^{(h)}} = \frac{\partial J(\boldsymbol{\theta}|\mathbf{x}, y)}{\partial \mathbf{z}^{(h+1)}} * \mathbf{W}^{(h+1)} \mathbf{1}_{n_h} * a'_{(h)}\left(\mathbf{z}^{(h)}\right) \\
&= \left[\mathbf{W}^{(h+1)}\right]^T \frac{\partial J(\boldsymbol{\theta}|\mathbf{x}, y)}{\partial \mathbf{z}^{(h+1)}} * a'_{(h)}\left(\mathbf{z}^{(h)}\right),
\end{aligned}
$$

where $\mathbf{1}_n$ is a vector of ones and is of length n, n_h is the length of the hidden layer number h and for simplicity of notation we have used $o = H + 1$. This constitutes the backpropagation algorithm. The reason for lack of citations in this subsection is that these equations have been derived from scratch by following the basic idea of backpropagation and utilizing chain and matrix derivation

rules. Additionally, while we in this thesis credit the Rumelhart *et al.* [29] paper from 1986 with a formal introduction of backpropagation and a significant contribution to the popularization of backpropagation, the idea had already been around for decades [30].

2.1.3 Batch and Layer Normalization

Batch normalization is a widely adopted technique that has been shown to enable more efficient and stable training of deep neural networks [31]. It is usually applied to hidden layers in a neural network either before or after applying activation function. During training the technique consists of standardizing each node in the input vector based on the first and second moments of the current mini-batch for the specific node [32]. Here input vector is used in the more general sense of input to the batch normalization layer. For a batch of n_{batch} input vectors $\left\{\mathbf{z}^{[i]}\right\}_{i=1,\ldots,n_{\text{batch}}}$ to a batch normalization layer, the input is transformed as follows

$$
\begin{aligned}
\boldsymbol{\mu} &= \frac{1}{n_{\text{batch}}} \sum_i \mathbf{z}^{[i]} \\
\boldsymbol{\sigma} &= \frac{1}{n_{\text{batch}}} \sum_i \left(\mathbf{z}^{[i]} - \boldsymbol{\mu}\right) * \left(\mathbf{z}^{[i]} - \boldsymbol{\mu}\right) \\
\mathbf{z}_{\text{BN}}^{[i]} &= \boldsymbol{\gamma} * \left(\mathbf{z}^{[i]} - \boldsymbol{\mu}\right) * \frac{1}{\sqrt{\boldsymbol{\sigma}^2 + \epsilon}} + \boldsymbol{\beta}, \qquad \text{for } i = 1, \ldots, n_{\text{batch}},
\end{aligned}
$$

where $\left\{\mathbf{z}_{\text{BN}}^{[i]}\right\}_{i=1,\ldots,n_{\text{batch}}}$ is the batch of outputs after applying batch normalization, $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ are two vectors of trainable parameters, $*$ denotes elementwise vector multiplication, and ϵ is a small constant to avoid numerical division by zero. The vectors $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$ are used to perform a linear transformation after standardizing the batch, where $\boldsymbol{\gamma}$ elementwise adjusts the standard deviation, and $\boldsymbol{\beta}$ elementwise adjusts the bias. During the training phase $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}^2$ are calculated for each batch, while $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$ are updated using the optimization function for the neural network. For the inference phase, however, the normalization is computed in a slightly different way. For an evaluation call to the neural network there may not be a full batch available, consequently, relying on calculating $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}^2$ for each evaluation call is not a very stable solution for small inference batch sizes and is even infeasible if the batch size is equal to 1. Instead during inference, $\boldsymbol{\mu}$ and $\boldsymbol{\gamma}$ are calculated as a moving average of the mean and standard deviations seen during training.

Despite being a very effective technique, the reasons for the effectiveness of batch normalization is not very well understood. The original paper Ioffe and Szegedy [32] believed the effectiveness was a result of batch normalization changing the input distributions to reduce what is referred to as "internal covariate shift". Informally, internal covariate shift alludes to the change in the distribution of a layer input caused by updates (during training) to the preceding layer [31]. The paper Santurkar *et al.* [31] demonstrated, however, that such distribution stability of the layer inputs appear to have little to do with the success of the batch normalization technique. Instead they found that fundamentally batch normalization makes the optimization landscape a lot smoother. They reason that this smoothness might induce a more stable and predictive behaviour of the gradients, thus leading to faster training.

Batch normalization is not always feasible (as we shall later see in Section 2.2.1) or desirable. An alternative to batch normalization is layer normalization [33]. Instead of computing the normalization separately for each node and based on a batch of training cases, layer normalization computes a single mean μ_l and a single variance σ_l for the entire layer based on a single training case. Consequently, layer normalization can be computed for all batch sizes and it performs

the same computation for the training and inference phase. Similar to batch normalization, layer normalization includes learnable parameters that perform a linear transformation for each node after the initial standardization. Mathematically, when adding layer normalization to a single input vector $\mathbf{z} = [z_1, \ldots, z_{n_{\text{layer}}}]$, the input is transformed as

$$\begin{aligned}
\mu_l &= \frac{1}{n_{\text{layer}}} \sum_{j=1}^{n_{\text{layer}}} z_j \\
\sigma_l^2 &= \frac{1}{n_{\text{layer}}} \sum_{j=1}^{n_{\text{layer}}} \left(z_j - \mu_l\right)^2 \\
\mathbf{z}_{\text{LN}} &= \mathbf{g} * \frac{\mathbf{z} - \mu_l}{\sqrt{\sigma_l^2 + \epsilon}} + \mathbf{b},
\end{aligned}$$

where $\mathbf{z}_{\text{LN}}$ is the output after applying layer normalization to the layer of a single training case, μ_l is the input layer mean, σ_l^2 is the input layer variance, ϵ is a small numerical constant to avoid division by zero and $\mathbf{g}$ and $\mathbf{b}$ are trainable vectors that perform a linear transformation for each node after standardization.

Xu *et al.* [34] argue that the effectiveness of layer normalization is mostly due to the rescaling and recentering of the gradients (computed backwards in the neural network). They also claim that layer normalization is prone to overfitting and suggest based on a few empirical experiments that a simpler version without scaling and bias terms, $\mathbf{g}$ and $\mathbf{b}$, may be just as or more effective. In the same paper they propose a modification of layer normalization, which they denote AdaNorm [34] for adapted normalization. It is defined as

$$\begin{aligned}
\mu_l &= \frac{1}{n_{\text{layer}}} \sum_{j=1}^{n_{\text{layer}}} z_j \\
\sigma_l^2 &= \frac{1}{n_{\text{layer}}} \sum_{j=1}^{n_{\text{layer}}} \left(z_j - \mu_l\right)^2 \\
\mathbf{z}_{\text{LN,simple}} &= \frac{\mathbf{z} - \mu_l}{\sqrt{\sigma_l^2 + \epsilon}} \\
\mathbf{z}_{\text{AN}} &= \phi(\mathbf{z}_{\text{LN,simple}}) * \mathbf{z}_{\text{LN,simple}} = C\left(1 - k\mathbf{z}_{\text{LN,simple}}\right) * \mathbf{z}_{\text{LN,simple}},
\end{aligned}$$

where $\mathbf{z}_{\text{AN}}$ is the output, C is a hyperparameter, and k is a parameter set to $\frac{1}{10}$ by the authors. In implementation the term $\phi(\mathbf{z}_{\text{LN,simple}})$ is regarded as an adjustable constant, thus the gradients are not calculated using the product rule for $\phi(\mathbf{z}_{\text{LN,simple}}) * \mathbf{z}_{\text{LN,simple}}$, but instead calculated for $\mathbf{z}_{\text{LN,simple}}$ and elementwise multiplied by the term $\phi(\mathbf{z}_{\text{LN,simple}})$, which is regarded as a vector of constants.

2.2 Generative Adversarial Networks (GANs)

Goodfellow *et al.* [35] introduced generative adversarial networks. The generative adversarial framework consists of two neural networks: a generative model G and a discriminative model D. The generative model G tries to capture the data distribution, while the discriminative model D attempts to distinguish which samples come from the real data distribution rather than being generated by G. The training process of G consists of optimizing G to fool the discriminator D, while the training

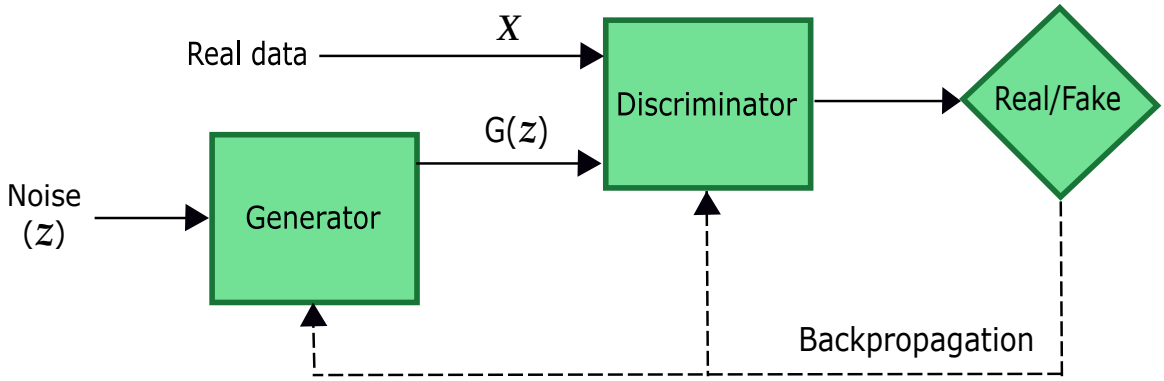


Figure 2.1: Illustration of a Generative Adversarial Model (GAN).

process of D consists of learning to separate between generated samples by G and real samples from the desired data distribution. Thus G and D are in competition with each other. The neural networks G and D are trained simultaneously. The aspiration is that the competition between the two networks will force them both to gradually improve. If G and D both are in the space of arbitrary functions, then there exists a unique solution where G recovers the training data distribution and D is equal to 0.5 everywhere since there is no difference between real samples and samples generated by G [35].

A simile (illustrative comparison) for a generative adversarial network is the competition between a forger trying to create counterfeit money and the police trying to determine which money is fake. As the forger improves, it will be harder for the police to separate counterfeit money from real money. If the police want to stay in the game, they will have to improve their detection skills. Similarly, when the police get better at detecting counterfeit money, the forgers will have to improve their skill at making imitations of real money if they want to continue to make money and avoid getting caught. If there was no police or anyone else detecting false money, forgers could get away with creating bad imitations. In the same way, the discriminator forces the generator to actually create samples that follow the approximate data distribution wanted.

The min-max game between G and D can be written mathematically. The exact equation depends on the chosen loss functions for G and D. Goodfellow *et al.* [35] suggest training D to maximize the probability of assigning correct label to both training data examples and samples from G and simultaneously train G to minimize $\log(1 - D(G(\mathbf{z})))$. Here $\mathbf{z}$ is a latent variable (noise) with probability distribution $P_{\mathbf{Z}}$. The generator G generates samples in an attempt to approximate the real data distribution from samples $\mathbf{z}$ drawn from $P_{\mathbf{Z}}$. With this loss function, the min-max game can be written

$$\min_G \max_D V(D,G) = E_{\mathbf{x}\sim\hat{P}_{\mathbf{X}}}[\log(D(\mathbf{x})] + E_{\mathbf{z}\sim P_{\mathbf{Z}}}[\log(1 - D(G(\mathbf{z})))].$$

Here $E_{\mathbf{x}\sim\hat{P}_{\mathbf{X}}}$ means taking the expectation with respect to $\mathbf{x}$ drawn from $\hat{P}_{\mathbf{X}}$, where $\hat{P}_{\mathbf{X}}$ is the observed data distribution (an approximation of the true data distribution $P_{\mathbf{X}}$). Likewise $E_{\mathbf{z}\sim P_{\mathbf{Z}}}$ means taking the expectation with respect to $\mathbf{z}$ drawn from $P_{\mathbf{Z}}$. We give an illustration of a GAN model in Figure 2.1.

Returning to the simile of the competition between a forger and the police, let us examine the situation where the police suddenly becomes (or always was) much better at detecting false money than the forger is at creating imitations. Then, it would be really difficult for the forger to understand how to improve, since no matter what the forger did or tried, the police easily classified his

imitations as fake. This is also a problem for generative adversarial networks. If the discriminator becomes much better than the generator, then the gradients provided by D will not be sufficient to train G to improve. As a small fix to this, Goodfellow *et al.* [35] suggest training G to maximize $\log(D(G(\mathbf{z}))$ early on, since when G performs badly, this gives stronger gradients. The paper Arjovsky *et al.* [36] suggest a subclass of GANs called Wasserstein GAN (WGAN) where this problem is fixed.

Another common problem with regular GAN is mode collaps. This happens when the generator only learns a portion of real data distribution, for instance, only generates the numbers 2 and 7, when we want it to generate all numbers between 0 and 9. Wasserstein GANs improve upon this issue but do not solve it completely.

2.2.1 Wasserstein GAN (WGAN)

Wasserstein GANs were introduced in the paper Arjovsky *et al.* [36]. Most of the theory presented in this subsection will be based on this paper. As the mathematical derivation of Wasserstein GANs can be a little complicated to follow for a reader the first time, we will first give a more intuitive and simpler explanation before we proceed to give a more mathematical explanation. Unlike regular GANs as first proposed in Goodfellow *et al.* [35], WGANs use a different loss function called the Earth-Mover distance. The authors give both theoretical and empirical motivations for why they think this loss function is more beneficial than the Jensen-Shannon divergence. Optimizing the Earth-Mover distance directly can be difficult, but thankfully there exists an equivalent version of the distance. This distance, however, requires a supremum over 1-Lipschitz functions. To solve this, the WGAN method uses a neural network forced to be 1-Lipschitz and trained to maximize the expression (this acts as an approximation to the suprenum over 1-Lipschitz functions). This neural network is referred to as the critic and replaces the discriminator.

A benefit of using the Earth-Mover distance as a loss function for G is that the derivatives are useful even when G is underperforming. Thus, the problem of the discriminator becoming too good too fast is avoided. Instead the better the discriminator is, the better it approximates the Earth-Mover distance. As the discriminator is no longer trained to predict probabilities, the discriminator in a Wasserstein GAN is called a critic. As the critic now approximates the Earth-Mover distance, we have an actual measurement of how well the generator performs. In a standard GAN there is no way to quantitatively measure how good the generator is, as the generator loss function is dependent on the discriminator. A good generator can have a worse loss function value than a bad generator if the discriminator corresponding to the bad generator is equally bad or worse. Additionally, it has been shown empirically that WGAN decreases the probability of mode collapse. This is intuitive since the loss function is a distance function and, with the elimination of unhelpful gradients for bad G, the critic can be trained to optimality. At optimality, the critic will penalize mode collapse, and the generator will receive encouragement to generate from all modes.

Now, we return to the more mathematical explanation of Wasserstein GANs. The Earth-Mover distance or Wasserstein-1 distance measures how close the generated distribution of G is to the desired data distribution [36]. The Earth-Mover distance between two probability distributions $\mathbb{P}_r$ and $\mathbb{P}_g$ can be written

$$W(\mathbb{P}_r, \mathbb{P}_g) = \inf_{\gamma \in \Pi(\mathbb{P}_r, \mathbb{P}_g)} E_{(\mathbf{x},\mathbf{y})\sim\gamma}[\|\mathbf{x}-\mathbf{y}\|]. \tag{2.5}$$

Here $\Pi(\mathbb{P}_r, \mathbb{P}_g)$ is the set of all joint distributions $\gamma(\mathbf{x}, \mathbf{y})$ whose marginal distributions are, respectively, $\mathbb{P}_r$ and $\mathbb{P}_g$. An interpretation of $\gamma(\mathbf{x}, \mathbf{y})$ is the amount of "mass" that must be transported from $\mathbf{x}$ to $\mathbf{y}$ in order to transform the marginal distribution $\mathbb{P}_r$ into the marginal distribution P_g. Fol-

lowing this intuition, the Earth-Mover distance can be regarded as the "cost" of the optimal transport plan [36].

The infinitum in Equation (2.5) is quite unmanageable, but using Kantorovich-Rubenstein duality $W(\mathbb{P}_r, \mathbb{P}_g)$ this equation be rewritten in a more useful form [36].

$$W(\mathbb{P}_r, \mathbb{P}_g) = \sup_{\|f\|_L \leq 1} E_{\mathbf{x}\sim\mathbb{P}_r}[f(\mathbf{x})] - E_{\mathbf{y}\sim\mathbb{P}_g}[f(\mathbf{y})]. \tag{2.6}$$

Here the supremum is taken over all 1-Lipschitz functions $f : \mathcal{X} \xrightarrow{\mathbb{R}}$. If f instead was K-Lipschitz for some constant $K > 0$, then the distance would be $K \cdot W(\mathbb{P}_r, \mathbb{P}_g)$ [36]. The definition of a K-Lipschitz function f is that it satisfies

$$\|f(\mathbf{x}) - f(\mathbf{y})\|_L \leq K \cdot \|\mathbf{x} - \mathbf{y}\|_L.$$

If we consider a parametric family of functions $\{f_w\}_{w\in\mathcal{W}}$ where all functions are K -Lipschitz and that the supremum in Equation (2.6) is achieved for some $w \in \mathcal{W}$, then we could instead solve for the optimization problem

$$\max_{w\in\mathcal{W}} E_{\mathbf{x}\sim\mathbb{P}_r}[f_w(\mathbf{x})] - E_{\mathbf{y}\sim\mathbb{P}_g}[f_w(\mathbf{y})].$$

Note that the assumption of supremum in Equation (2.6) being achieved by f_w for some w is a quite strong assumption [36]. It will, however, still give us an approximation to the Earth Mover distance times a constant K. Here, it might be motivational to remember the Universal Approximation theorem and that neural networks can in theory uniformly approximate any function to any degree of accuracy. The idea behind Wasserstein GANs is to force the critic D to be K-Lipschitz such that

$$\sup_D E_{\mathbf{x}\sim\mathbb{P}_\mathbf{X}}[D(\mathbf{x})] - E_{\mathbf{z}\sim\mathbb{P}_\mathbf{z}}[D(G(\mathbf{z}))], \tag{2.7}$$

is an approximation of K times the Earth-Mover distance between the generated distribution by G and the real data distribution $P_\mathbf{X}$.

The paper Arjovsky *et al.* [36] ensures that the critic is K-Lipschitz by forcing the weights of the neural network $\mathbf{w}$ to lie in a compact space $\mathcal{W}$. They do this by clipping the weights to a fixed box. This is not an optimal method. As the authors themselves state "*Weight clipping is a clearly terrible way to enforce a Lipschitz constraint. If the clipping parameter is large, then it can take a long time for any weights to reach their limit, thereby making it harder to train the critic til optimality. If the clipping is small, this can easily lead to vanishing gradients when the number of layers is big, or batch normalization is not used.*" [36].

Gulrajani *et al.* [37] show examples of WGAN with weight clipping having either exploding or vanishing gradients, as well as failing to capture higher moments of the data distribution. As such, they instead propose an alternative way to enforce the Lipschitz constraint by including a gradient penalty. A differentiable function is 1-Lipshitz if and only if it has gradients with norm at most 1 everywhere [37]. Including a term penalizing the gradient for being higher than 1 enforces a soft version of the Lipschitz constraint. For simplicity, Gulrajani *et al.* [37] use a two-sided penalty, thereby encouraging the gradient norm to go towards 1, not only be below 1. Empirically they found that this two-sided penalty did not constrain the critic too much compared to the one-sided penalty [37].

The new objective function is

$$L = E_{\mathbf{z}\sim P_\mathbf{z}}[D(G(\mathbf{z}))] - E_{\mathbf{x}\sim P_\mathbf{X}}[D(\mathbf{x})] + \lambda \cdot E_{\tilde{\mathbf{x}}\sim P_{\tilde{\mathbf{x}}}}\left[(\|\nabla_{\tilde{\mathbf{x}}} D(\tilde{\mathbf{x}})\|_2 - 1)^2\right],$$

where they define the probability distribution $P_{\tilde{\mathbf{X}}}$ by sampling uniformly along straight lines between the pairs of points sampled from the real data distribution, $P_{\mathbf{X}}$, and the generated data distribution $P_{\hat{\mathbf{X}}}$. The probability distribution $P_{\hat{\mathbf{X}}}$ is the distribution of $G(\mathbf{z})$ when $\mathbf{z}$ is sampled from $P_{\mathbf{Z}}$. In Algorithm 1 we give the training procedure for a WGAN with gradient penalty using Adam [28] as the optimizer.

When using Wasserstein GANs with gradient penalty (WGAN-GP) it is important to not use batch normalization. Batch normalization changes the critic's task of mapping a single input to a single output to the task of mapping an entire batch of inputs to a batch of outputs, and then the penalizing method is no longer valid [37]. Therefore, when we in Section 3.2 create our data synthesizer framework and in Section 5.3 define our model based counterfactual framework using WGAN-GP, we will not use batch normalization for the critic architecture.

Algorithm 1: WGAN with gradient penalty. Gulrajani *et al.* [37] uses the default values $\lambda = 10$, $n_{critic} = 5$, $\alpha = 0.0001$, $\beta_1 = 0$, $\beta_2 = 0.9$. The algorithm is taken from Gulrajani *et al.* [37], but altered to match the notation of this thesis.

Input : λ `/* Gradient penalty coefficient */`
Input : m `/* Batch size */`
Input : α, β_1, β_2 `/* Adam hyperparameters */`
Input : $\mathbf{w}_0$ `/* Initial critic parameters */`
Input : $\boldsymbol{\theta}_0$ `/* Initial generator parameters */`
while $\boldsymbol{\theta}$ has not converged **do**
  for $t = 1, \ldots, n_{critic}$ **do**
    for $i = 1, \ldots, m$ **do**
      Sample real data $\mathbf{x} \sim P_{\mathbf{X}}$
      Sample latent variable $\mathbf{z} \sim P_{\mathbf{Z}}$
      Sample a random number $\epsilon \sim U[0, 1]$
      $\hat{\mathbf{x}} \leftarrow G_{\boldsymbol{\theta}}(\mathbf{z})$
      $\tilde{\mathbf{x}} \leftarrow \epsilon \mathbf{x} + (1 - \epsilon)\hat{\mathbf{x}}$
      $L^{(i)} \leftarrow D_{\mathbf{w}}(\hat{\mathbf{x}}) - D_{\mathbf{w}}(\mathbf{x}) + \lambda(||\nabla_{\tilde{\mathbf{x}}} D_{\mathbf{w}}(\tilde{\mathbf{x}})||_2 - 1)^2$
    end
    $\mathbf{w} \leftarrow Adam(\nabla_{\mathbf{w}} \frac{1}{m} \sum_{i=1}^{m} L^{(i)}, \mathbf{w}, \alpha, \beta_1, \beta_2)$
  end
  Sample a batch of latent variables $\{\mathbf{z}^{(i)}\}_{i=1}^{m} \sim P_{\mathbf{Z}}$
  $\boldsymbol{\theta} \leftarrow Adam(\nabla_{\boldsymbol{\theta}} \frac{1}{m} \sum_{i=1}^{m} -D_{\mathbf{w}}(G_{\boldsymbol{\theta}}(\mathbf{z})), \boldsymbol{\theta}, \alpha, \beta_1, \beta_2)$
end

2.3 XGBoost - An Implementation of Gradient Boosted Decision Trees

XGBoost [38] is an implementation of gradient-boosted decision trees. The XGBoost framework is designed to be highly eefficient, flexible and portable. In order to not to make this section too long, as the XGBoost model is not very essential to the contributions of this thesis, we assume the reader is already familiar with what a decision tree [39] is. However, a very brief explanation is a flowchart tree structure of nodes, where each node is basically an if-else statement. The terminal (leaf) nodes are thus the result of a series of if-else statements and hold the predicted values for samples satisfying these series of conditions. Decision trees can be efficient and interpretable, but such models often have high variance and do not give the best performance [39]. One solution to

		Truth	
		Positive	Negative
Predicted	Positive	True Positive (TP)	False Positive (FP)
	Negative	False Negative (FN)	True Negative (TN)

Table 2.1: Confusion matrix.

this is to use a combination of many trees. Bagging [40] fits many trees to bootstrapped versions of the training data and uses an average of predictions for each tree for regression problems as well as usually majority vote for classification problems. Random forest is a fancier version of bagging that reduces correlation between the trees by using a random subset of covariates as candidates for each split [41]. Boosting trees are a method where each new tree is trained to improve upon the errors of the previous trees [42, 43]. Boosting algorithms generally train new models to improve on the errors of previous models by reweighting the training observations. Gradient boosting methods, on the other hand, utilize gradient descent to minimize the loss function when adding a new model to the ensemble of models [44, 45]. Thus, each new model is fit to predict the residuals of the ensemble of previous models. A version of this algorithm is implemented by XGBoost. The XGBoost version of the algorithm also include stochastic subsampling of features (similar to random forest), regularized objective function and shrinkage of weight parameter for each new tree [38]. The addition of stochastic subsampling means that XGBoost is an implementation of stochastic gradient boosting trees.

2.4 Metrics for Efficacy of a Classification Machine Learning Model

There are many different metrics for classification models. Let $\mathbf{y} = [y_1, \ldots, y_n]$ be a vector of observations with the true classes, and let $\hat{\mathbf{y}} = [\hat{y}_1, \ldots, \hat{y}_n]$ be the corresponding vector of predicted classes according to some classification model. Here y_i can either be binary or multiclass. The accuracy metric measures the percent of correctly classified observations

$$Accuracy(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{n} \sum_{i}^{n} I(y_i = \hat{y}_i),$$

where I is the indicator function. Accuracy is a useful metric for classification problems that are well balanced and do not contain very imbalanced classes. Let us imagine a classification problem of predicting whether a person has a heart attack in the next day. Even a simple model that always predicts no heart attack will likely have an accuracy score close to 100% even if it is practically useless.

For binary classification problems, precision and recall are other useful metrics [46]. To define these it is useful to define what a true positive, false positive, false negative and true negative is. Let one class be the positive class and the other be the negative class. A true positive (TP) is when the model correctly predicts the positive class, while a false positive (FP) is when the model predicts the positive class, but in reality it is the negative class. A false negative is when the model predicts the negative class (FN) when it actually is the positive class, while a true negative is when the model correctly predicts the negative class (TN). See Table 2.1 for an overview using a confusion matrix.

Precision is defined as the proportion of predicted positives that are actually positive

$$Precision(\mathbf{y}, \hat{\mathbf{y}}) = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}},$$

while recall is the proportion of actual positives that are correctly classified

$$Recall(\mathbf{y}, \hat{\mathbf{y}}) = \frac{\text{TP}}{\text{TP} + \text{FN}}.$$

Here, and for the rest of this section, we merely write TP, FP, FN and TN in the equations instead of specifying that they are functions of $\mathbf{y}$ and $\hat{\mathbf{y}}$. Returning to the classification problem of predicting whether a person has a heart attack in the next day with the naive model of always predicting no heart attack. If we set heart attack as the positive class, then both precision and recall are equal to zero since we never predict the positive class. Precision is a useful evaluation metric if we want to be very sure of our prediction (when we predict the positive class). For example if we want to build a system for decreasing the credit limit for some customers, then we usually would want to be very certain that we did not decrease peoples credit limit unnecessary, else we might get a lot of dissatisfied customers. A downside to being very precise is that we might miss out on many of the observations that actually are of the positive class. Recall is a useful evaluation metric when we desire to catch as many of the positive class as possible. For instance if we want to predict heart attack, it is much more unfavorable to misclassify the positive class (heart attack) compared to the negative class (not heart attack). Be aware, however, that a naive model that always predicts heart attack will have a recall of 1. Note that both recall and precision can be used for multiclass classification problems. This is done by setting a single class as the positive class and all the other classes as the negative class.

The F1 score [46] is a trade-off between precision and recall. The F1 score is a number between 0 and 1 and is defined as

$$F_1(\mathbf{y}, \hat{\mathbf{y}}) = 2 \cdot \frac{Precision(\mathbf{y}, \hat{\mathbf{y}}) \cdot Recall(\mathbf{y}, \hat{\mathbf{y}})}{Precision(\mathbf{y}, \hat{\mathbf{y}}) + Recall(\mathbf{y}, \hat{\mathbf{y}})}.$$

With the F1 score, both a model that always predicts the positive class as well as a model that always predicts the negative class, will get a value of 0. A downside to the F1 score is that it weights both prediction and recall equally. Depending on the application, it might be desirable to weight them differently. The weighted F1 metric fixes this, but we do not include it here as it is not used in this thesis. If desired, a F1 value can be calculated for each class. This is common in multinomial classification problems. The F1 values can be evaluated separately or they can be combined into a single metric. There are multiple ways to combine the F1 scores. In this thesis, we only use macro-averaged F1 score, also known as macro F1 score. The macro F1 score is merely the unweighted average of the F1 score for each class.

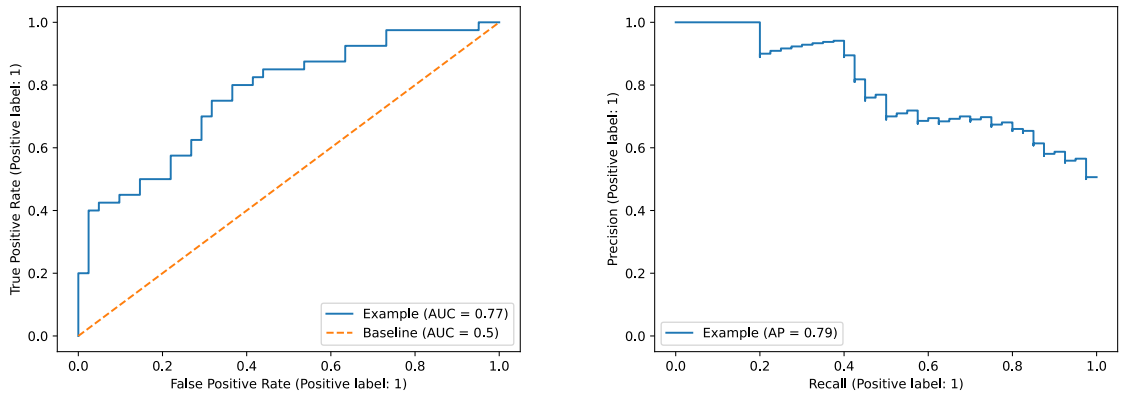
Another metric is the AUC [46], also known as AUROC. It is the area under the ROC curve. To define ROC it is useful to first define sensitivity and specificity. Sensitivity is the proportion of the positive class that we actually predict as positive, i.e the same as recall, while specificity is the proportion of the negative predictions actually belonging to the negative class

$$Sensitivity(\mathbf{y}, \hat{\mathbf{y}}) = \frac{\text{TN}}{\text{TN} + \text{FP}}.$$

The ROC curve is defined as sensitivity plotted on the y-axis against $(1 - \text{specificity})$ on the x-axis for multiple probability thresholds when classifying each class. $(1 - \text{specificity})$ is the same as the proportion of the true negative observations falsely predicted to be of the positive class

$$1 - Sensitivity(\mathbf{y}, \hat{\mathbf{y}}) = \frac{\text{TN}}{\text{TN} + \text{FP}}.$$

If a model has no predictive ability, the ROC curve will merely be a straight line from $(0, 0)$ to $(1, 1)$. The corresponding AUC value will then be zero. If a model correctly classifies all observations, the ROC curve will be a straight line from $(0, 0)$ to $(0, 1)$ in combination with a straight line from



(a) ROC curve

(b) Precision-recall curve

Figure 2.2: Example of ROC and precision-recall curve for a simple example. We let $\mathbf{p} = [p_1, \ldots, p_{81}] = [0.10, 0.11, 0.12, \ldots, 0.49, 0.5, 0.51, \ldots, 0.88, 0.89, 0.90]$ be the estimated probabilities from the model and let the supposed ground truth for this example be defined as $y = [y_1, \ldots, y_{81}]$ where $y_i = I(u_i < p)$ and $u_1, \ldots, u_{81}$ are independent and identically distributed samples from a Uniform$(0, 1)$ distribution. $I(\cdot)$ is the indicator function.

$(0, 1)$ to $(1, 1)$. The AUC value will, therefore, be 1. Note that if a classification model classifies all observations wrongly, it gets an AUC value of 0. An AUC value of 0.5 is considered a baseline as it is the score of a model without predictive capabilities.

The AUC curve might not be optimal for very skewed machine learning problems (very imbalanced classes) [47]. An alternative to the ROC curve is the Precision-Recall curve. The Precision-Recall curve [47] plots precision on the y-axis and recall on the x-axis. The metric AUPRC is defined as the area under the Precision-Recall curve. The baseline for the AUPRC model is the fraction of observations with the positive class. In Figure 2.2, we give an example of a ROC curve in Figure 2.2a and an example of a precision-recall curve in Figure 2.2b.

Chapter 3

Data Synthesizing

Synthetic data is characterized by being artificially created instead of being obtained directly from measurements of events in the real world [48]. The umbrella that is synthetic data contains both data obtained through simulation scenarios and data generated by sampling techniques trained on or applied to real-world data. The book *Synthetic data for deep learning* [49] distinguishes between synthetic data and data augmentation; where they define data augmentation as a set of techniques to modify real data rather than create new synthetic data. However, they state that there is a blurry line. The book also defines three types of main usage for synthetic data [49, Chapter 1].

1. Using synthetic datasets to train machine learning models directly; usually either with intention to use them on real data or in order to train (usually generative) models to refine data, making it more suitable for training.
2. Using synthetically generated data to extend or augment existing real datasets. For these situations, the synthetic data samples are usually used to provide more examples for parts of the data distribution that are not represented well enough in the real dataset.
3. Using synthetically generated data to avoid legal or privacy issues associated with using or sharing real data. This is especially applicable for healthcare where there are strict regulations with respect to sharing of images or health care records, but also in many other sectors and applications.

In this thesis, we focus on creating synthetic datasets that follow the data distribution of a real-world dataset. We give a brief overview of such methods in Section 3.1. Next, in Section 3.2 we explain in detail the GAN-based data synthesizer framework we develop in this thesis. Lastly, we give a brief introduction for each of the state-of-the-art data synthesizers that will be used as baselines for experiments in Chapter 7.

3.1 A Brief Overview of Generating Synthetic Data from Real Data

The book *Accelerating AI with Synthetic Data* [48, Chapter 1] divides methods for generating synthetic data from real data into two classes. Both classes have a generative component and a discriminative component. The generative component, often referred to as generator, creates an approximation of the real data distribution and uses this to generate synthetic data. The discriminative component, often referred to as the discriminator, compares the synthetic data against the real data, and in case it is found that the synthetic data differs from the real data, the parameters of the generative component are adjusted. This process is iterated until the generator learns to synthesize data of acceptable quality. The book uses the input data to the generator as the distinguishing element between the classes. The two classes are:

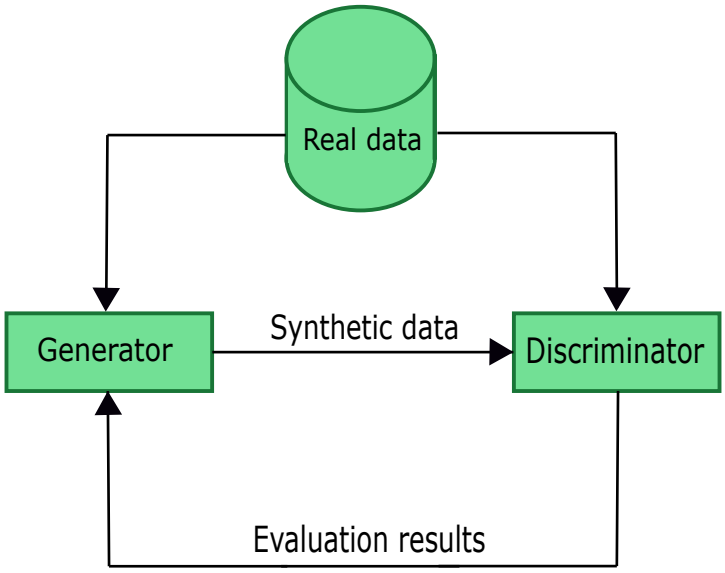


Figure 3.1: A general schematic for synthesizing data when the primary input to the generator during training is real data. The figure is a redrawn version of Figure 1-2 in the book *Accelerating AI with Synthetic Data* [48, Chapter 1].

1. Methods for generating synthetic data that use real data as input to the generator
2. Methods for generating synthetic data that use random data as input to the generator

We will explain the difference between the two classes in a bit more detail and give some examples of methods that belong to each class. The first class is an umbrella of many methods, but they all have in common that the generator has direct access to the real data during training. For an illustration of this, see Figure 3.1. One of the simplest sets of approaches within this class is, perhaps, to fit independent distributions to all variables in the real data and then compute the correlation between the variables. The first step can consist of choosing and fitting a parametric distribution for each variable or a more simple method that, for example, only uses the mean and variance statistics of each variable. Having collected this information, a potential way to sample synthetic data is to use Monte Carlo simulation techniques while enforcing similar correlation patterns to those observed in the real data [48, Chapter 1]. Alternatively, or in combination with Monte Carlo sampling, a copula can be used. A copula is a method that allows for the decomposition of a joint probability distribution into marginal distributions and a function that models the dependence between the variables. If the copula allows for direct sampling, then synthetic data can be sampled directly from the marginals and correlation structure. An example of a Gaussian copula implementation that can directly generate synthetic data is given in Section 3.3.3.

Numerous more advanced approaches exist. Algorithms based on linear models, decision trees, stochastic vector machines (SVM), random forest, or neural networks may for example be used to create synthesized data. One usage of these could be to decompose the joint probability distribution of the data, $P_{\mathbf{X}}(\mathbf{x})$, into products of conditional probability distributions $P_{X_j|X_1,\ldots,X_{j-1}}(x_j|x_1,\ldots,x_{j-1})$ and then use the machine learning methods above to learn each of the conditional distributions [50, 51]. We will not go into any more detail about these methods, but Drechsler and Reiter [50] and Dandekar *et al.* [51] find that decision trees appear to be especially suitable for such algorithmic probability decomposition methods.

Deep learning synthesizing methods, such as variational autoencoders (VAE), are other good alternatives. Autoencoders consist of an encoder and a decoder and are a way of learning a repres-

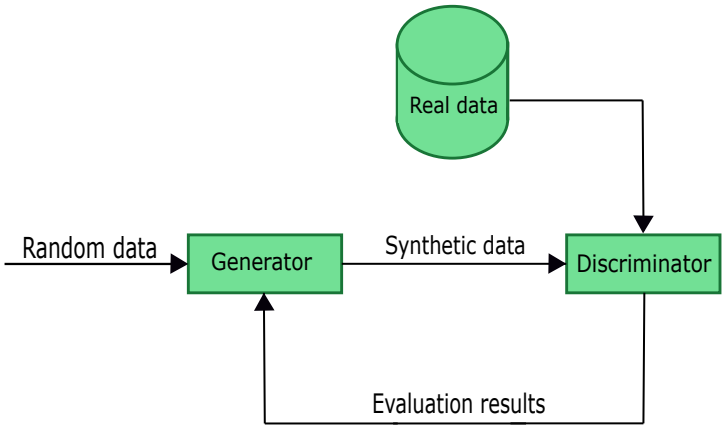


Figure 3.2: A general schematic for synthesizing data when the primary input to the generator during training is random noise. The figure is a redrawn version of Figure 1-3 in the book *Accelerating AI with Synthetic Data* [48, Chapter 1].

entation of data (typically a lower-dimensional encoding). Variational autoencoders assume that the input information is defined by an underlying latent distribution (encoding) and that it is possible to reconstruct the input data from the representation in the latent space (decoding) [4, 12]. VAEs also try to estimate the parameters of the latent distribution. When the underlying latent distribution is modelled and the decoder structure is sufficiently learned, then synthesized data can easily be generated by sampling from the latent distribution and then applying the decoder.

The main set of methods belonging to the second class is Generative Adversarial Networks, which we described in Section 2.2. In the standard GAN structure, the generator only receives a noise vector, whilst the discriminator is the only one with access to the real data. The generator learns to replicate the original data distribution indirectly through feedback from the discriminator. In Figure 3.2, we plot a general scheme for the second class of generative methods. Some examples of GAN-based data synthesizers are given in Section 3.2 and Section 3.3.

3.2 A Wasserstein GAN Based Data Synthesizer Framework – tabGAN

In this section we present our contribution to the wide range of already existing data synthesizing methods. The framework, which we name tabGAN, is implemented with a pipeline for automatic preprocessing and post-processing of data. The general steps performed by the framework can be defined as follows. First, it executes a preprocessing step that transforms the data to a format that can be given as input to a neural network. It then implements a Wasserstein GAN with gradient penalty to model the underlying data distribution of the transformed data. We implement several different model architecture and training process options. Finally, tabGAN generates samples from the learned data distribution and invert the transformations done in the preprocessing step to arrive at a dataset with the original format. The "tab" part of the name refers to the fact that the method is intended for tabular data. The preprocessing step and the model architecture of the GAN is inspired by state-of-the-art tabular GANs [1, 3, 13]. We implement two training procedures: one based on regular GAN and one based on conditonal GAN (CGAN) [52]. The intention behind the conditional GAN is to be able to get more representation for the rare categories in the dataset and therefore help the generator with learning of data distributions with imbalanced classes. The conditional GAN method is inspired from the CTGAN synthesizer [1] and the authors describe this as the main intent. An added benefit, however, is that it allows for conditional sampling of synthetic data based

on a single column category. We explain each of the steps in more detail in the sections below.

3.2.1 Preprocessing Step

Since the data to be modeled is tabular, it can consist of a combination of continuous, integer, binary and categorical data. A neural network requires both input and output to be numerical. Thus, for the input to the critic to be on an acceptable format and for the generator to have an effective data representation that can be learned as the output format, we must first perform some data transformation. An additional reason for performing preprocessing is that standardizing or normalizing the numerical data can make it easier for the neural networks to learn.

For binary and categorical data we perform one-hot encoding. One-hot encoding means that one binary column is added for each category in the categorical column (an alternative form of one-hot encoding could be to add one binary column for all categories except the last category and let the last category be represented by 0 in all other binary columns). Currently, there is no extra functionality for ordinal categorical data, they are merely regarded as nominal by the framework. In the tabGAN framework, there is an option to add some uniform noise to the one-hot encoding. This is controlled by the parameter $\delta_{\text{oh-noise}}$. For a one-hot encoded variable $\mathbf{d}_j = [d_{j,1}, \ldots, d_{j,l_j}]$ where l_j is the number of categories for discrete column j, then the one-hot encoded variable after applying noise, $\mathbf{d}_j^{[\text{noisy}]}$, is defined

$$
\begin{aligned}
\mathbf{d}_j^{[\text{noisy}]} &= [d_{j,1}^{[\text{noisy}]}, \ldots, d_{j,l_j}^{[\text{noisy}]}] \\
d_{j,k}^{[\text{noisy}]} &= d_{j,k} - u_{j,k} \cdot I(d_{j,k} = 1) + u_{j,k} \cdot I(d_{j,k} = 0), \qquad \text{for } k = 1, \ldots, l_j \\
u_{j,k} &\sim \text{Uniform}(0, \delta_{\text{oh-noise}}), \qquad \text{for } k = 1, \ldots, N_j.
\end{aligned}
$$

With respect to integer and continuous data we implement three options. The first option is to simply standardize integer and continuous columns. Standardization is performed by subtracting the mean of the column and dividing by the standard deviation. The transformed column will then have mean zero and standard deviation equal to 1. The mean and the standard deviation of each column are stored, meaning that they can be used in the inverted transformation.

The second option that we implement for integer and continuous columns is a quantile transformation. Quantile transformation is described in Section 4.1. We implement the quantile uniform transformation and the quantile normal transformation as possible choices. The quantile transformation with Gaussion (normal) output distribution is, however, the default choice and the one we will use when we later in this thesis evaluate the second preprocessing. The third option that we implement is the Randomized Quantile Transformation (QTR). The Randomized Quantile Transformation is described in Section 4.3. It is a novel transformation that we were inspired to create during the work leading up to this thesis, which we believe has beneficial qualities over regular quantile transformation when a column has many repeated values (which is common for integer columns or continuous columns with a large share of values equal to zero). If there are no repeated values in a column, then the Randomized Quantile Transformation is no different than regular quantile transformation. Similar to the regular quantile transformation option, we implement QTR with both uniform and Gaussian output distribution, but Gaussian output distribution is the default choice.

3.2.2 Model Architectures

Let there be a dataset with N_D categorical columns and N_C numerical columns and let a row of the transformed data using one of the methods from Section 3.2.1 be represented as

$$\mathbf{r}' = \mathbf{c}' \oplus \mathbf{d}' = c'_1 \oplus \cdots \oplus c'_{N_C} \oplus \mathbf{d}'_1 \oplus \cdots \oplus \mathbf{d}'_{N_D}$$
$$l_j = dim(\mathbf{d}'_j), \quad \text{for } j = 1, \ldots, N_D$$
$$l_D = dim(\mathbf{d}') = \sum_{j}^{N_D} l_j$$
$$l_w = dim(\mathbf{r}'),$$

where $\mathbf{c}'$ represents the transformed numerical columns, $\mathbf{d}'$ represents the discrete columns, c'_j represent jth transformed numerical column, $\mathbf{d}'_j$ denotes the one-hot encoded vector of the jth categorical column, and $\oplus$ is the operator for concatenating vectors. The dimension of the whole row is l_w, while the jth discrete column's one-hot encoding is denoted by l_j. The dimension of the concatenated one-hot encoded vectors of the discrete columns, $\mathbf{d}'$, is l_D. We use the $'$ notation to signify that it is the transformed version of the data.

Since we are providing a framework, it is difficult to define a single model architecture that will represent the framework. On the other hand, it can potentially be a bit overwhelming for the reader if all possible configurations and options are listed and explained at once. We aim at a compromise between these two approaches. We postulate that the single most important user choice in the framework is whether to use the regular WGAN implementation or the conditional WGAN implementation. For each of these two options, we will present the default model architecture in the tabGAN framework. We will also, where it is convenient, give brief explanations about some of the available customization choices. To separate the two main choices, from now on we will refer to the regular WGAN implementation of the tabGAN framework as tabGAN and the conditional WGAN implementation of the tabGAN framework as ctabGAN. To avoid confusion with the name of the framework, we will always include the word "framework" when referring to the tabGAN framework.

Regular WGAN Model Architecture – tabGAN

For the regular WGAN implementation, the only input to the generator is a realization of a standard normally distributed latent variable, $\mathbf{z}$, of length l_z. The input layer is fully connected to a neural layer of size $l_{h,g}$ with GELU activation function (defined in Section 2.1.1) followed by a batch normalization layer (defined in Section 2.1.3). Here, GELU activation is the default, but the framework allows for replacing GELU with any of the other activation functions referenced in Section 2.1.1 that are suitable as activation functions for a hidden layer. It is also an option to use identity activation function, which is the same as no activation function. This option is, however, not recommended as activation functions are what allows the neural network to learn non-linear relationships.

The first hidden layer in the generator is fully connected to a second hidden layer of size $l_{h,g}$ with GELU activation followed by a batch normalization layer. Again, the activation function is interchangeable. The second hidden layer is then connected to several layers. Firstly, it is fully connected to a dense layer of length N_C. This is the numerical output layer. Secondly, it is connected to multiple dense layers with respective lengths $l_1, \ldots, l_{N_D}$. Each of these dense layers corresponds to a discrete column. Letting these dense layers be the output for each discrete column would be an acceptable solution. The neural network would, after learning, try to output layers with the one-hot encoding format, but would have a hard time doing this since the one-hot encoding format is difficult for a neural network to replicate without help, at least in the author's experience. Taking arg max of each dense output categorical layer would solve the one-hot encoding perfectly but is a

bad choice since it is a non-differentiable function (thus making learning through back-propagation impossible). A better alternative would be to use a softmax activation function. Then the output from each categorical output layer would be transformed such that all nodes lie between 0 and 1 and that all nodes in each layer sum up to 1. For a dense layer $\mathbf{y}'_j = [y'_{j,1}, \ldots, y'_{j,l_j}]$, the softmax activation function is defined as

$$d'_{j,k} = \frac{e^{y_{j,k}/\tau}}{\sum_{i=1}^{l_j} e^{y_{j,i}/\tau}} \qquad \text{for } k = 1, \ldots, l_j,$$

where τ is the softmax temperature (sometimes set equal to 1 in certain softmax definitions such as the one in Equation (2.4)). When $\tau \to 0$ the softmax function converges to the arg max function. The softmax function in the Tensorflow library [53] always has $\tau = 1$, but we implement a wrapper around it that allows for specifying the softmax temperature τ in the tabGAN framework.

Let us now analyze one categorical output layer and for simplicity we assume that the output in the numerical layer and all other categorical output layers are given (in reality, for our neural network architecture, the generator synthesize the outputs in parallel). When all other output columns are given, what we want is for the generator to generate output in the last discrete layer that is realistic to observe in combination with the other numerical and categorical variables. When we use softmax as an activation function for the output categorical layer, what we implicitly ask the generator to do in the last dense categorical layer, is to generate samples. Unless there is only one suitable category, we want the last categorical layer to output different categories for different realizations of the latent variable $\mathbf{z}$, and we want the ratio of categories sampled to be equal to the ratio in the real data distribution conditioned on the same numerical and categorical variable values. This is doable for a neural network, but a challenge can be that it easily neglects (and therefore undersamples) the categories with low but still larger than zero probabilities. What potentially might be easier for the neural network, could be to have the dense categorical output layer estimate the conditional probabilities (or at least ratios) for each category and then have the activation function take care of the random sampling. This is the recommended and default approach in the tabGAN framework. Inspired by Xu *et al.* [1] and Rajabi and Garibay [3], we by default use a Gumbel softmax [54] activation function to produce one-hot encoded vector samples instead of softmax activation function. Despite Gumbel softmax being the default, regular softmax can, in the tabGAN framework, easily be chosen instead of Gumbel softmax. During our hyperparameter tuning we found no clear winner between Gumbel softmax and regular softmax.

The Gumbel softmax activation function for a dense layer with probabilities $\boldsymbol{\pi}'_j = [\pi'_{j,1}, \ldots, \pi'_{j,l_j}]$ is defined

$$d'_{j,k} = \frac{e^{(\log(\pi_{j,k})+g_k)/\tau}}{\sum_{i=1}^{l_j} e^{(\log(\pi_{j,i})+g_i)/\tau}} \qquad \text{for } k = 1, \ldots, l_j,$$

where $g_1, \ldots, g_{l_j}$ are $Gumbel(0,1)$ distributed. The $Gumbel(0,1)$ distribution can be sampled using inverse transform sampling by generating $u_i \sim Uniform(0,1)$ and computing $g_i = -\log(-\log(u_i))$. For our model, we let each dense categorical output layer $\mathbf{y}'_j = [y'_{j,1}, \ldots, y'_{j,l_j}]$ be the logit values $y'_{j,k} = \log(\pi_{j,k}/(1-\pi_{j,k}))$. Thus, using the Gumbel activation function we get

$$d'_{j,k} = \frac{e^{(-\log(1-\exp\{y_{j,k}\})+g_k)/\tau}}{\sum_{i=1}^{l_j} e^{(-\log(1-\exp\{y_{j,k}\})+g_i)/\tau}} \qquad \text{for } k = 1, \ldots, l_j.$$

As a final step in the generator architecture, the categorical output layers mapped through the Gumbel activation function (or alternatively softmax if desired by the user) are concatenated. Thus,

the default model architecture can formally be described as

$$\begin{cases} \textit{Input: } \mathbf{z} \qquad \text{(latent vector)} \\ \mathbf{h}_1 = BN\left(GELU\left(FC_{l_z \to l_{h,g}}(\mathbf{z})\right)\right) \\ \mathbf{h}_2 = BN\left(GELU\left(FC_{l_{h,g} \to l_{h,g}}(\mathbf{h}_1)\right)\right) \\ \hat{\mathbf{c}}' = FC_{l_{h,g} \to l_{N_C}}(\mathbf{h}_2) \\ \hat{\mathbf{d}}' = Gumbel_\tau\left(FC_{l_{h,g} \to l_1}(\mathbf{h}_2)\right) \oplus \cdots \oplus Gumbel_\tau\left(FC_{l_{h,g} \to l_{N_D}}(\mathbf{h}_2)\right) \\ \textit{Output: } \hat{\mathbf{c}}', \hat{\mathbf{d}}', \end{cases}$$

where $FC_{a \to b}$ represents a fully connected layer with input size a and output size b, $GELU(\mathbf{y})$ denotes applying a GELU activation on $\mathbf{y}$, $BN(\cdot)$ denotes applying a batch normalization layer and $Gumbel_\tau(y)$ denotes applying Gumbel softmax activation with temperature parameter τ on a vector $\mathbf{y}$. Notationwise, we let $\mathbf{h}_1$ and $\mathbf{h}_2$ be the two hidden layers and $\hat{\mathbf{c}}'$ and $\hat{\mathbf{d}}'$ be the output numeric layer and the output discrete layer, respectively. The input to the generator is the latent vector $\mathbf{z}$. The architecture of the generator for the Adult dataset treated in Section 6.2 is plotted in Figure 3.3.

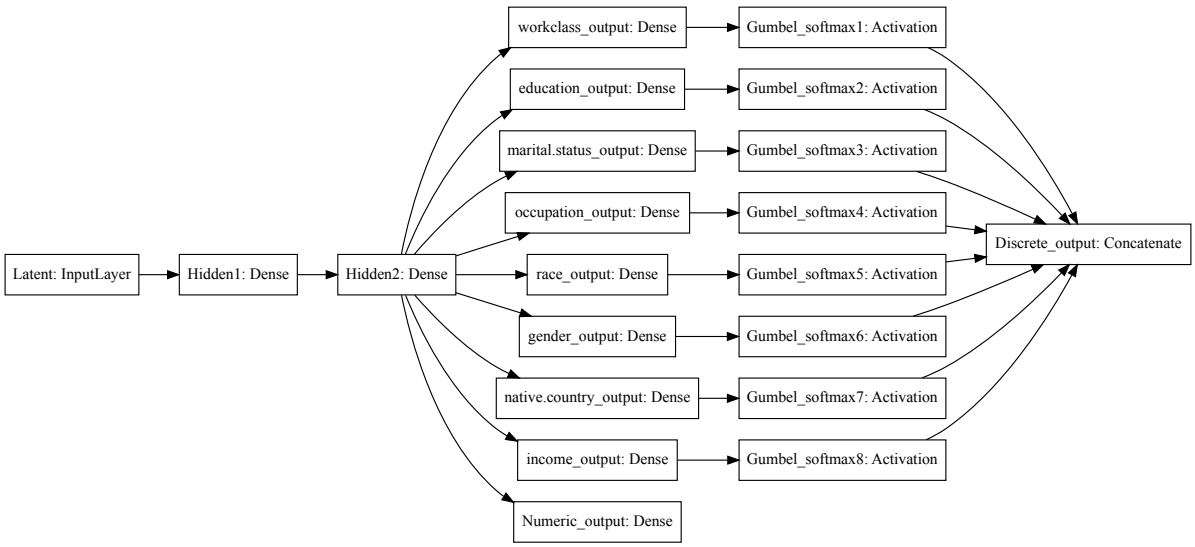


Figure 3.3: Plot of default generator architecture for the regular GAN implementation in the tabGAN framework for the Adult dataset treated in Section 6.2. The GELU activation function and the batch normalization layers added to the first and second hidden layer (Hidden1 and Hidden2) are not included in the plot. The dense layers corresponding to each categorical variable is named [variable name]_output.

For the regular WGAN implementation, the critic receives two inputs: the numerical columns, $\mathbf{c}'$, and the categorical columns, $\mathbf{d}'$. Recall that since we implement a Wasserstein GAN we use a critic instead of a discriminator. By default these two inputs are merely concatenated, but the separate input structure easily allows for more complicated designs. Next, the concatenated layer is fully connected to a neural layer of size $l_{h,d}$ with GELU activation function as default. This first hidden layer is, again, fully connected to a second hidden layer also of size $l_{h,d}$ with by default GELU activation function. The second hidden layer is then fully connected to a single output node. The value of the output node is the predicted logit value of the given input being an observation from the real data distribution. Formally, the critics default architecture can be described as

$$\begin{cases} \textit{Input: } \mathbf{c}', \mathbf{d}' & \text{(numeric and discrete output of generator or transformed real data)} \\ \mathbf{h}_{IN} = \mathbf{c}' \oplus \mathbf{d}' \\ \mathbf{h}_1 = GELU\left(FC_{l_w \to l_{h,d}}(\mathbf{h}_{IN})\right) \\ \mathbf{h}_2 = GELU\left(FC_{l_{h,d} \to l_{h,d}}\, \mathbf{h}_1)\right) \\ h_{OUT} = FC_{l_{h,d} \to 1}(\mathbf{h}_2) \\ \textit{Output: } h_{OUT}, \end{cases}$$

where $\mathbf{h}_{IN}$ is the input layer, $\mathbf{h}_1$ and $\mathbf{h}_2$ are the two hidden layers, and h_{OUT} is the output node. The architecture of the discriminator for the Adult dataset described in Section 6.2 is plotted in Figure 3.4.

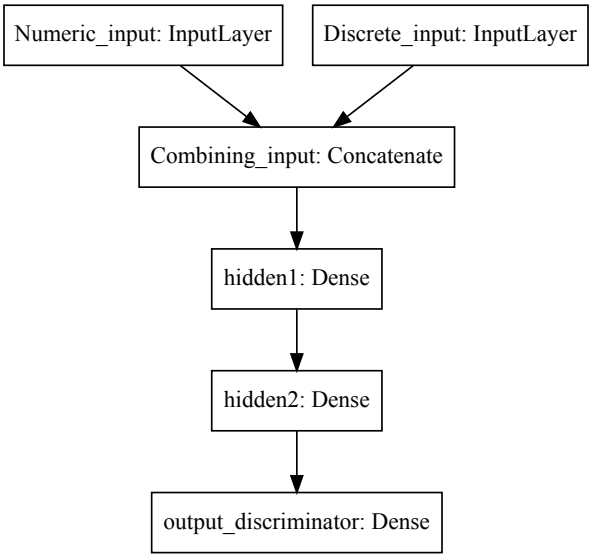


Figure 3.4: Plot of the default critic architecture for the regular GAN implemenation in the tabGAN framework for the Adult dataset treated in Section 6.2. The GELU activation function added to the first and second hidden layer (hidden1 and hidden2) is not included in the plot.

Conditional WGAN Model Architecture – ctabGAN

For the conditional WGAN implementation, the generator receives two inputs: a realization of a standard normally distributed latent variable, $\mathbf{z}$, of length l_z and a conditional vector $\mathbf{m}' = \mathbf{m}'_1 \oplus \ldots \oplus \mathbf{m}'_{N_D}$ of length l_D. Each of the conditional vectors $\mathbf{m}'_j$ are of length l_j and with binary elements. The conditional vector $\mathbf{m}'$ can be regarded as a concatenation of smaller mask vectors associated with each of the one-hot encoded discrete columns. It is defined such that exactly a single element of $\mathbf{m}'$ is equal to 1 instead of zero, that is $\sum_{j=1}^{N_D} \sum_{k=1}^{l_j} m'_{j,k} = 1$ and $d_{j,k} \in \{0,1\}$ for $k = 1, \ldots, l_j$ for $j = 1, \ldots, N_D$. The single value where $d_{j,k} = 1$ informs the generator that we want to condition on the kth category of the jth column. In other words, we want the sample generated by the generator to be of the kth category for the jth column. The reason we only implement conditioning on discrete columns, is that for a category value of a single column in the dataset, there are usually multiple observations available in the original dataset. On the other hand, for a single numerical value there is usually only a single matching observation for continuous columns or a

few matching observations for integer columns. We therefore believe it is reasonable to assume that this makes it simpler to estimate the conditional distribution of a discrete column compared to the conditional distribution of a numerical column. It would, however, be interesting to investigate if it can be beneficial to include conditioning on numerical columns. Perhaps the generator will be able to learn the conditional distribution for each of the numerical columns directly, or maybe an eligible option can be to condition on intervals of the numerical columns. Both input layers $\mathbf{z}$ and $\mathbf{m}'$ are concatenated into a single layer. The rest of the default ctabGAN model architecture is the same as the default architecture for tabGAN. The ctabGAN default model architecture is formally described as

$$\begin{cases} \textit{Input: } \mathbf{z}, \mathbf{m}' \qquad \text{(latent vector and conditional vector)} \\ \mathbf{h}_{IN} = \mathbf{z} \oplus \mathbf{m}' \\ \mathbf{h}_1 = BN\left(GELU\left(FC_{l_z \to l_{h,g}}\left(\mathbf{h}_{IN}\right)\right)\right) \\ \mathbf{h}_2 = BN\left(GELU\left(FC_{l_{h,g} \to l_{h,g}}\left(\mathbf{h}_1\right)\right)\right) \\ \hat{\mathbf{c}}' = FC_{l_{h,g} \to l_{N_C}}\left(\mathbf{h}_2\right) \\ \hat{\mathbf{d}}' = Gumbel_\tau\left(FC_{l_{h,g} \to l_1}\left(\mathbf{h}_2\right)\right) \oplus \cdots \oplus Gumbel_\tau\left(FC_{l_{h,g} \to l_{N_D}}\left(\mathbf{h}_2\right)\right) \\ \textit{Output: } \hat{\mathbf{c}}', \hat{\mathbf{d}}'. \end{cases}$$

In the default architecture for the conditional WGAN implementation the critic receives three inputs: the numerical columns $\mathbf{c}'$, the categorical columns $\mathbf{d}'$, and the conditional vector $\mathbf{m}'$. The conditional vector informs the critic of which conditional distribution the pair $\{\mathbf{c}', \mathbf{d}'\}$ should be drawn from. By default, these three inputs are merely concatenated and the rest of the default critic architecture for ctabGAN is the same as for tabGAN. Consequently, the default critic architecture for ctabGAN can formally be written as

$$\begin{cases} \textit{Input: } \mathbf{c}', \mathbf{d}', \mathbf{m}' \qquad \text{(transformed numeric and discrete columns and conditional vector)} \\ \mathbf{h}_{IN} = \mathbf{c}' \oplus \mathbf{d}' \oplus \mathbf{m}' \\ \mathbf{h}_1 = GELU\left(FC_{l_w \to l_{h,d}}\left(\mathbf{h}_{IN}\right)\right) \\ \mathbf{h}_2 = GELU\left(FC_{l_{h,d} \to l_{h,d}}\left(\mathbf{h}_1\right)\right) \\ h_{OUT} = FC_{l_{h,g} \to 1}\left(\mathbf{h}_2\right) \\ \textit{Output: } h_{OUT}. \end{cases}$$

Extra Customization Available for Model Architecture in tabGAN Framework

More options are implemented in the tabGAN framework than what we have described earlier in this section. Now we try to efficiently describe some of them. For the default tabGAN architecture and for the default ctabGAN architecture, both the generator and the critic have two hidden layers. Additionally, the default value for generator hidden layer size is $l_{h,g} = 256$ and the critic hidden layer size is $l_{h,d} = 256$. The number of generator and critic layers can be set to any positive integer (and not necessarily the same value), and additionally, each hidden layer can easily have its own layer size. By default, the dimension of the latent vector, $\mathbf{z}$, is $l_z = 128$. This can also be set by user to any positive integer. Additionally, dropout layers (see Section 2.1.1) can be freely added after the concatenated input layer or after each of the hidden layers. The dropout rate of the layers can be set to different values for the generator and critic, if desired. For the generator, we use batch normalization after applying activation function to each hidden layer by default. Since there

is debate whether batch normalization should be applied before or after the activation function, we implement the option to include batch normalization before the activation instead. Of course, we also implement the option to not use batch normalization. As stated in Section 2.2.1, batch normalization can not be used in the critic architecture for Wasserstein GANs with gradient penalty, as it changes the critic's task of mapping a single input to a single output to the task of mapping an entire batch of inputs to a batch of outputs, and then the penalizing method is no longer valid. Layer normalization can, however, be used as a drop-in replacement for batch normalization, since with layer normalization the critic still maps a single input to a single output. In the tabGAN framework, we incorporate layer normalization as an option for the critic, either before or after the activation function is applied to each hidden layer. However, by default we do not include layer normalization, as it was unclear, from the limited testing we performed with layer normalization, if it on average is beneficial or not.

The tabGAN framework also accommodates more complex layer interactions for the generator. If desired a dense connection between the numeric output layer and each of the discrete column output layers can be added. There is also the option of adding a hidden layer of arbitrary size and an activation function between the numeric output layer and each of the discrete column output layers. Then the model architecture for ctabGAN can be written as

$$
\begin{cases}
\textit{Input: } \mathbf{z}, \mathbf{m}' \qquad \text{(latent vector and conditional vector)} \\
\mathbf{h}_{IN} = \mathbf{z} \oplus \mathbf{m}' \\
\mathbf{h}_1 = BN\left(GELU\left(FC_{l_z \to l_{h,g}}\left(\mathbf{h}_{IN}\right)\right)\right) \\
\mathbf{h}_2 = BN\left(GELU\left(FC_{l_{h,g} \to l_{h,g}}\left(\mathbf{h}_1\right)\right)\right) \\
\hat{\mathbf{c}}' = FC_{l_{h,g} \to l_{N_C}}\left(\mathbf{h}_2\right) \\
\mathbf{h}_{c \to d} = a\left(FC_{l_{N_C} \to l_{h,c \to d}}\left(\hat{\mathbf{c}}'\right)\right) \\
\hat{\mathbf{d}}' = Gumbel_\tau\left(FC_{(l_{h,g}+l_{c \to d}) \to l_1}\left(\mathbf{h}_2 \oplus \mathbf{h}_{c \to d}\right)\right) \oplus \cdots \oplus Gumbel_\tau\left(FC_{(l_{h,g}+l_{c \to d}) \to l_{N_D}}\left(\mathbf{h}_2 \oplus \mathbf{h}_{c \to d}\right)\right) \\
\textit{Output: } \hat{\mathbf{c}}', \hat{\mathbf{d}}',
\end{cases}
$$

where a is some arbitrary activation function, $\mathbf{h}_{c \to d}$ is the hidden layer of size $l_{h,c \to d}$ between the numeric output layer, $\hat{\mathbf{c}}'$, and each of the discrete column output layers. Alternatively, the same connection can be added from the discrete output layer to the numeric output layer, with an optional hidden layer and activation function in between. Then the model architecture for ctabGAN can be written

$$
\begin{cases}
\textit{Input: } \mathbf{z}, \mathbf{m}' \qquad \text{(latent vector and conditional vector)} \\
\mathbf{h}_{IN} = \mathbf{z} \oplus \mathbf{m}' \\
\mathbf{h}_1 = BN\left(GELU\left(FC_{l_z \to l_{h,g}}\left(\mathbf{h}_{IN}\right)\right)\right) \\
\mathbf{h}_2 = BN\left(GELU\left(FC_{l_{h,g} \to l_{h,g}}\left(\mathbf{h}_1\right)\right)\right) \\
\hat{\mathbf{d}}' = Gumbel_\tau\left(FC_{l_{h,g} \to l_1}\left(\mathbf{h}_2\right)\right) \oplus \cdots \oplus Gumbel_\tau\left(FC_{l_{h,g} \to l_{N_D}}\left(\mathbf{h}_2\right)\right) \\
\mathbf{h}_{d \to c} = a\left(FC_{l_D \to l_{h,d \to c}}\left(\hat{\mathbf{d}}'\right)\right) \\
\hat{\mathbf{c}}' = FC_{(l_{h,g}+l_{h,d \to c}) \to l_{N_C}}\left(\mathbf{h}_2 \oplus \mathbf{h}_{d \to c}\right) \\
\textit{Output: } \hat{\mathbf{c}}', \hat{\mathbf{d}}',
\end{cases}
$$

where $\mathbf{h}_{d \to c}$ is the hidden layer of size $l_{h,d \to c}$ between the discrete output layer, $\hat{\mathbf{d}}'$, and the numeric output layer, $\hat{\mathbf{c}}'$. The last two options can not be chosen simultaneously. Otherwise, there would be

a loop from numeric output layer to discrete output layer and back to numeric output layer again. A third option that can be used independently of the two previous is to add a dense connection from the conditional vector, $\mathbf{m}'$, directly to each of the discrete column output layers. During hyperparameter tuning, we found that these extra connections were sometimes beneficial, but the trends were not clear enough that we trusted it to generalize well to other datasets. They are, however, options that can be tried to squeeze out some extra performance. In particular, the connection from the discrete output layer to the numeric output layer tended to perform well. More exploration is needed, as we do not investigate it thoroughly in this thesis.

3.2.3 Training Phase

The training phase in the tabGAN framework depends a bit upon chosen user configurations. However, for the regular WGAN implementation, tabGAN, the training phase is pretty straightforward. It basically follows Algorithm 1 to the letter. As default choice, we use $n_{critic} = 10$ updates of the critic for each update of the generator, $\lambda = 10$ as the penalty coefficient, a batch size of $n_{\text{batch}} = 500$ and the default train length is 300 epochs. The default optimizer is Adam with $\beta_1 = 0.7$ and $\beta_2 = 0.999$. However, multiple other optimizers are available. Defining the length of an epoch is actually not straight forward when using a WGAN as n_{critic} times as many rows of observations are used to train the critic as the generator (since the critic is updated n_{critic} times for each time the generator is updated). We choose, however, to use the same definition as implemented by Xu *et al.* [1] and Rajabi and Garibay [3], which we also believe to be the most reasonable one. One epoch is then defined as $\max\{n_{\text{train}} // n_{\text{batch}}, 1\}$ where // signifies integer division and n_{train} is the number of observations in the training dataset given to the synthesizer.

With the conditional GAN implementation, ctabGAN, the training phase is a bit more complicated. For each row in a batch, a conditional vector $\mathbf{m}'$ must be chosen. The conditional vector should specify a category of single column to condition on. What distribution the column and category choice should follow is up for debate, but we have chosen to implement the same two choices as in the Xu *et al.* [1] paper. They divide the process of choosing the conditional vector $\mathbf{m}'$ for each row into two steps. First a column is chosen uniformly from the list of discrete columns. Having chosen a discrete column j, the category is drawn from a distribution where category k has probability $p_k^{[j]}$ of being chosen. We implement two different choices for the probabilities. Let $n_k^{[j]}$ be the number of observations with category k in discrete column j.

1. The first choice is to let the category probabilities be equal to the proportion of each category in the column, that is $p_k^{[j]} = n_k^{[j]} / \sum_{i=1}^{l_j} n_i^{[j]}$. This gives the generator more chances to learn the common categories, but as a downside it might not get enough representation of the rare categories.
2. The second option is to let the probabilities be defined as the logarithm of the frequency of the category. The probabilities are of course normalized such that they sum to 1. Then the probability for category k is defined as $p_k^{[j]} = \log(n_k^{[j]}) / \sum_{i=1}^{l_j} \log(n_i^{[j]})$. This gives the generator a bit more chances to learn the rare categories whilst still sampling the common categories more often. This is the default choice for ctabGAN in the tabGAN framework.

Having chosen category k for column j, the conditional vector for that row is created as a vector of length l_D with zeros everywhere except at the kth element of $\mathbf{m}_j$, where the value is 1. When choosing a real observation to pair with a chosen $\mathbf{m}'$, an observation is uniformly drawn from all observations that are of the kth category for column j. Except for this, the training procedure is very similar to Algorithm 1. Instead of merely giving a batch of random noise vectors, we give a batch of random noise vectors along with a batch of conditional vectors. Instead of merely giving

a batch of real or fake data to the critic, we give a batch of real or fake data along with a batch of corresponding conditional vectors. As the conditional vector z is given as input to the critic along with the data row r (generated sample or observation), we expand the gradient loss term to force the critic to be 1-Lipschitz for the entire input. We also add an additional term to the generator loss. This extra term is a cross-entropy loss between $\mathbf{m}^{[j^*]}$ and $\mathbf{d}_{j^*}$, where j^* is the chosen column. The loss term is averaged across the batch. The extra loss term is proposed to enforce the generator to produce the category of the column conditioned on. By default, ctabGAN includes this loss term as we have observed it to be beneficial, but the tabGAN framework also allows for the option not to include the extra generator loss term, as even without it the critic appears to implicitly impose the constraint on the generator.

Packing Training Samples for the Critic

The paper Lin *et al.* [55] propose an approach to mitigate mode collapse for GAN which they call packing. The main idea of the approach is to modify the discriminator (or in our case critic) to make decisions based on multiple samples at once. The packing method can easily be implemented on top of most existing GAN architectures, including Wasserstein GAN which we use in this thesis. To implement packing, all that needs to be done is to feed the critic n_{pac} samples at once. This should not be confused with giving the critic a batch of observations during training. Feeding the critic a batch of n_{batch} samples leads to n_{batch} independent inputs to the critic and consequently n_{batch} output values. Feeding the critic n_{pac} packed samples leads to a single pass through the critic and a single output value. With packing, the generator in the tabGAN architecture takes as input n_{pac} concatenated latent noise vectors and the critic takes as input n_{pac} concatenated observations. The ctabGAN architecture takes the same, but additionally n_{pac} concatenated conditional vectors.

By getting the critic to classify n_{pac} samples jointly, the critic gets to observe samples from product distributions. Lin *et al.* [55] argue that intuitively mode collapse is more clearly observed in such product distributions. They prove theoretically a fundamental connection between packing and mode collapse and show empirically that packing appears to reduce mode collapse. We implement packing as an option in the tabGAN framework, but as we during our hyperparameter tuning consistently got worse performance with packing, we do not include it by default. We believe that this might be because WGAN already reduces mode collapse and even though the paper Lin *et al.* [55] found packing to be beneficial for WGAN also, they only performed the experiment on a dataset with vast amounts of modes. We reason that packing might be useful for tabGAN if we have a dataset where the number of modes are very high.

3.2.4 Synthesizing Data on Original Data Format

Synthesizing data with the tabGAN framework after the training phase is over is a very simple process. To generate a dataset with n_{new} observations using tabGAN, the generator is first fed a batch with n latent noise vectors. The generator then outputs two datasets, each with n rows. The first dataset contains the numerical columns and the second dataset contains the discrete column. Then the numerical dataset is transformed back to the original data format using the inverted numerical transformation used for preprocessing, whether that was standardization, quantile transformation or the Randomized Quantile Transformation. Similarly, the one-hot encoded discrete columns are transformed back to the original data format. Then the numerical and discrete dataset, both on the original data format, are merged.

For ctabGAN the process is very similar, but there is one more step at the beginning. As ctabGAN generates samples for conditional distributions, we must supply a batch of conditional vectors along with the batch of latent noise vectors. No matter which category probabilities (either proportional to

log of frequency or just frequency) were chosen to use during training, the probabilities used during sampling are always the ones proportional to frequency of category, that is $p_k^{[j]} = n_k^{[j]} / \sum_{i=1}^{l_j} n_i^{[j]}$. The reason for this is that we want the generated data to follow the original data distribution, not the optional new data distribution where the rare categories are more common.

3.2.5 Implementation Details

The entire model is implemented as a class in Python [56]. The model is primarily implemented using the Tensorflow library [53] as well as the Tensorflow Probability library [57]. Other significant packages used are Pandas [58, 59] for data processing and Scikit-learn [60] for most of the preprocessing and post-processing of the data.

Much emphasis has been placed on creating a fast implementation. Many of the processes performed during training are written in such a way that Tensorflow can create a graph structure of the process, thus leading to significant speedups. Additionally, for time-consuming processes during training much care has been taken to ensure that Tensorflow is able to utilize just-in-time compilation with Accelerated Linear Algebra for even further speed-ups. Just-in-time compilation means that code is compiled during execution of the code, not beforehand. Additionally, the data fetching process for tabGAN (not ctabGAN) is implemented such that the CPU can perform the fetching, and if needed prefetch batches of the data, whilst the GPU is working. However, as tabular datasets are usually quite small size-wise (at least compared to batches of high-resolution images and such), the data fetching process is quite fast anyway and the performance gains are neglible. On the other hand, for very large tabular datasets or very large batch sizes, data prefetching might be beneficial.

3.3 A Selection of State-of-the-art Data Synthesizers

In this section, we present some data synthesizers that we will later use as baselines for comparison with our tabGAN methods in Chapter 7. We continue using the notation defined in Section 3.2. Many of the data synthesizing methods we describe below can be found in the Synthetic Data Vault (SDV) package [2]. SDV is an ecosystem that provides a couple of different synthetic data models along with some tools to benchmark data synthesizers and calculate metrics.

3.3.1 CTGAN

The CTGAN model was presented in Xu *et al.* [1]. It is quite similar to our default ctabGAN implementation explained in Section 3.2. The most significant difference is that CTGAN employs a different preprocessing transformation for the numerical columns. For each continuous column, the CTGAN method uses a variational Gaussian mixture model (VGM) to estimate the number of modes and fit a Gaussian mixture. A Gaussian mixture model assumes that the data points are generated from a finite and known number of Gaussian distributions with unknown parameters [61]. Another difference with CTGAN compared to ctabGAN is that CTGAN concatenates each new hidden layer in the generator with the previous layer. Additionally, for the numerical (not one-hot encoded) output, CTGAN uses the hyperbolic tangent function, tanh, as activation function. The generator architecture for CTGAN can mathematically be described as

$$
\begin{cases}
\textit{Input: } \mathbf{z}, \mathbf{m} \qquad \text{(latent vector and conditional vector similar to ctabGAN)} \\
\mathbf{h}_0 = \mathbf{z} \oplus \mathbf{m} \\
\mathbf{h}_1 = \mathbf{h}_0 \oplus ReLU\left(BN\left(FC_{(|\mathbf{m}|+|z|)\to 256}\left(\mathbf{h}_0\right)\right)\right) \\
\mathbf{h}_2 = \mathbf{h}_1 \oplus ReLU\left(BN\left(FC_{(|\mathbf{m}|+|z|+256)\to 256}\left(\mathbf{h}_1\right)\right)\right) \\
\hat{\alpha}_j = \tanh\left(FC_{(|\mathbf{m}|+|z|+512)\to 1}\left(\mathbf{h}_2\right)\right), \qquad \text{for } 1 \leq j \leq N_C \\
\hat{\boldsymbol{\beta}}_j = Gumbel_{\tau=0.2}\left(FC_{(|\mathbf{m}|+|z|+512)\to m_i}\left(\mathbf{h}_2\right)\right), \qquad \text{for } 1 \leq j \leq N_C \\
\hat{\mathbf{d}}_j = Gumbel_{\tau=0.2}\left(FC_{(|\mathbf{m}|+|z|+512)\to |\mathbf{d}_i|}\left(\mathbf{h}_2\right)\right) \\
\textit{Output: } \left\{\hat{\alpha}_j\right\}_{j=1,\ldots,N_C}, \left\{\hat{\boldsymbol{\beta}}_j\right\}_{j=1,\ldots,N_C}, \left\{\hat{\mathbf{d}}_j\right\}_{j=1,\ldots,N_D},
\end{cases}
$$

where $\mathbf{z}$ is the latent vector, $\mathbf{m}$ is the conditional vector defined for the ctabGAN architecture, $BN(\cdot)$ is batch normalization, $ReLU$ is the Rectified Linear activation function defined in Section 2.1 and $\hat{\mathbf{d}}_j$ is the generated one-hot encoding for the jth discrete column for $j = 1, \ldots, N_D$. The outputs $\left\{\hat{\alpha}_j\right\}_{j=1,\ldots,N_C}$ and $\left\{\hat{\boldsymbol{\beta}}_j\right\}_{j=1,\ldots,N_C}$ are representations of the numerical columns for CTGAN. For column j the one-hot encoded vector $\hat{\boldsymbol{\beta}}_j$ defines which mode in the Gaussian mixture model for column j the generated observation belongs to. Given $\hat{\boldsymbol{\beta}}_j$, the scalar $\hat{\alpha}_j$ denotes where in the mode the current value for column j of the generated observations is located. We will not elaborate further on the notation defined in the CTGAN paper or the preprocessing method, but refer to Xu *et al.* [1] for the interested reader.

The critic architecture for the CTGAN model is very similar to the critic of ctabGAN, except that CTGAN uses LeakyReLU instead of GELU as activation function after each hidden layer and CTGAN also uses dropout after each LeakyReLU activation. By default, CTGAN uses packing with pac size equal to 10 for the critic, but it can be changed. To not use packing, it is enough to set $n_{\text{pac}} = 1$. Without packing the CTGAN architecture can formally be described as

$$
\begin{cases}
\textit{Input: } \mathbf{r}', \mathbf{m} \qquad \text{(real or synthesized data row and conditional vector)} \\
\mathbf{h}_{IN} = \mathbf{r}' \oplus \mathbf{m}' \\
\mathbf{h}_1 = Dropout\left(LeakyReLU_{\alpha=0.2}\left(FC_{256\to 256}\left(\mathbf{h}_{IN}\right)\right)\right) \\
\mathbf{h}_2 = Dropout\left(LeakyReLU_{\alpha=0.2}\left(FC_{256\to 256}\left(\mathbf{h}_1\right)\right)\right) \\
h_{OUT} = FC_{256\to 1}\left(\mathbf{h}_2\right) \\
\textit{Output: } h_{OUT},
\end{cases}
$$

where $Dropout(\cdot)$ means a dropout layer and $LeakyReLU(\cdot)$ means applying the LeakyReLU activation function defined in Section 2.1.1. The CTGAN model is available both as a standalone Python package and as part of the Synthetic Data Vault package (SDV). The implementation in the SDV package is merely a wrapper around the original CTGAN implementation. When we later in this thesis evaluate the CTGAN synthesizer, we evaluate the method both with and without packing. We refer to the CTGAN method without packing as CTGAN-pac1 and the CTGAN method with ten packed samples to the critic as CTGAN-pac10.

3.3.2 TVAE

The TVAE model was presented in the same paper as CTGAN. It is a variational autoencoder structure specifically designed for tabular data. It uses the same preprocessing techniques as CTGAN, that is, variational Gaussian mixture models for the numeric columns and one-hot encoding of the

categorical columns. We will not give a detailed description of TVAE here; instead we will merely refer to the paper Xu *et al.* [1]. The default implementation of TVAE uses two hidden layers of size 128 in both the encoder and decoder neural architecture. As both the tabGAN framework and CTGAN by default use two hidden layers of size 256 for both the generator and critic, we will include a implementation of TVAE where the the hidden layers are of size 256. We call the slightly modified version of TVAE with hidden layer sizes equal to 256 for TVAE-mod, and the default version for TVAE-orig. The TVAE method is implemented in both the standalone Python package CTGAN and as part of the SDV package. The implementation in the SDV package is merely a wrapper around the original TVAE implementation for the CTGAN package.

3.3.3 GaussianCopula

A multivariate distribution can be decomposed into marginal distributions, correlation parameters and a dependency structure. This dependency structure is known as a copula and describes how the marginal distributions interact. More mathematically, a copula is a multivariate cumulative distribution function where each of the marginal probability distributions is uniform on the interval $[0, 1]$ [62]. A Gaussian copula is constructed from a multivariate normal distribution by utilizing the probability integral transform. A Gaussian copula data synthesizer, GaussianCopula, is implemented in the Stochastic Data Vault package. To accommodate categorical data, one-hot encoding is used for the discrete columns. Then marginal distributions are selected for each column (either original numerical columns or the one-hot encoded columns). The marginal distributions can be chosen manually by the user, but the GaussianCopula implementation also allows for automatically detecting the appropriate marginal distribution of each column. Having selected the marginal distribution types, the parameters of the marginal distributions are fitted to the data. The next step in the method is to fit a Gaussian copula. Having fitted the marginal distributions and the copula function, new data are easily synthesized by sampling from the multivariate Gaussian copula function and then transforming the samples using probability integral transformation.

3.3.4 CopulaGAN

CopulaGAN is another synthesizer implemented in the SDV package. It is basically a combination of CTGAN and the transformations from the GaussianCopula. The steps of CopulaGAN can be described as follows

1. Transform each non-categorical column using the same transformer as GaussianCopula. That is, if not specified, atuomatically select a suitable marginal distribution and fit it to the column data. Then transform each column to the standard normal space by first applying the cumulative distribution function of the fitted marginal distribution and then applying the inverse cumulative distribution function for the standard normal distribution.
2. Fit a CTGAN model to the transformed dataset.

To generate a synthetic dataset with CopulaGAN, the process is as follows

1. Sample a dataset of the desired size using CTGAN
2. Reverse the transformations from the added preprocessing step. That is, for each of the numerical columns, apply the cumulative distribution function of the standard normal distribution followed by the inverted cumulative distribution function that corresponds to each numerical column.

CopulaGAN or CTGAN might be the best performing methods of the well-known GAN-based data synthesizers. At least they outperform other GAN based data synthesizers such as MedGAN

[63], TableganSynthesizer [64], and VEEGANSynthesizer [65] in the leaderboard of the Python benchmarking framework SDGym, which is part of the Synthetic Data Vault project. In the same ranking they also outperform Bayesian network based synthesizers such as PrivBNSynthesizer [66] and CLBNSynthesizer [67].

3.3.5 TabFairGAN

The TabFairGan synthesizer [3] is a method for generating fair synthetic datasets (for instance by removing racial or genderwise bias). However, in this paper, we regard it as a standard tabular GAN and do not use the fairness functionality. The tabFairGAN synthesizer is not available in any Python packages, but it is available as a script on the Github belonging to this paper[1]. The tabFairGAN method uses quantile uniform transformation (see Section 4.1) as preprocessing for numerical columns and one-hot encoding for discrete columns. The generator architecture can formally be described as

$$\begin{cases} \textit{Input: } \mathbf{z} \qquad \text{(latent vector)} \\ \mathbf{h}_0 = \mathbf{z} \\ \mathbf{h}_1 = ReLU\left(FC_{l_w \to l_w}\left(\mathbf{h}_0\right)\right) \\ \hat{\mathbf{r}}' = ReLU\left(FC_{l_w \to l_w}\left(\mathbf{h}_1\right)\right) \oplus Gumbel_{\tau=0.2}\left(FC_{l_w \to N_C}\left(\mathbf{h}_1\right)\right) Gumbel_{\tau=0.2}\left(FC_{l_w \to l_1}\left(\mathbf{h}_1\right)\right) \oplus \ldots \oplus \\ Gumbel_{\tau=0.2}\left(FC_{l_w \to l_{N_D}}\left(\mathbf{h}_1\right)\right) \\ \textit{Output: } \hat{\mathbf{r}}', \end{cases}$$

where $\hat{\mathbf{r}}'$ is a synthesized row and l_w is, as we have previously defined, the dimension of a transformed data row $\mathbf{r}'$. The critic architecture can be described as

$$\begin{cases} \textit{Input: } \mathbf{r}' \qquad \text{(output of generator or transformed real data)} \\ \mathbf{h}_1 = LeakyReLU_{\alpha=0.01}\left(FC_{l_w \to l_w}\left(\mathbf{r}'\right)\right) \\ \mathbf{h}_2 = LeakyReLU_{\alpha=0.01}\left(FC_{l_w \to l_w}\left(\mathbf{h}_1\right)\right) \\ \textit{Output: } mean(\mathbf{h}_2). \end{cases}$$

The output from the critic is the mean of the layer $\mathbf{h}_2$. The paper Rajabi and Garibay [3] experimentally shows that TabFairGAN achieves similar performance to CTGAN (the default method of CTGAN which we refer to as CTGAN-pac10) for the Adult dataset.

[1] TabFairGAN Github link: `https://github.com/amirarsalan90/TabFairGAN`

Chapter 4

Randomized Quantile Transformation

One of the main contributions of this thesis is the introduction of the novel transformation method, which we have named the Randomized Quantile Transformation (QTR). Both regular quantile transformation and the Randomized Quantile Transformation are implemented as possible pre-processing methods for numerical data in our data synthesizer framwork, tabGAN, and our model-based counterfactual framework, tabGAN-cf. The QTR adds stochastic elements to what is often referred to as quantile transformation. The definitions of a quantile transformation in the literature is not entirely consistent or complete, but in Section 4.1, we try to give a definition based on the content of the papers Beasley *et al.* [68] and Bogner *et al.* [69] and the implementation in the Python [56] machine learning library scikit-learn [60]. Some of the examples given in the sections below are taken from these papers, others are natural extensions by the author. Since the quantile transformation and Randomized Quantile Transformation used in the tabGAN framework are built upon the quantile transformation implementation in the scikit-learn package, this specific implementation of quantile transformation is explained in some detail in Section 4.1. We proceed to explain the motivation for creating QTR in Section 4.2 and the mathematical defintion of QTR in Section 4.3.

4.1 Quantile Transformation

A quantile transformation is characterized by the usage of quantiles to map one probability distribution to another probability distribution. If the cumulative distribution function (CDF) is known, it can be used as the transformation. If x is a realization of the random variable X with the cumulative distribution function F_X, then $y = F_X(x)$ is a realization from a Uniform$(0, 1)$ distribution (or an approximation of it if F is left right continuous). If desired, it can be further transformed into any other distribution by utilizing the (generalized) inverse of that distribution's cumulative distribution function. For instance $z = \Phi^{-1}(y) = \Phi^{-1}(F_X(x))$ is a sample from a standard normal distribution. Converting to the Uniform$(0, 1)$ or standard normal distribution is the common choice.

For practical applications the true cumulative distribution function is rarely known. Thus, another transformation or an approximation must be used to map the original observations to the new distribution. We make the distinction between rank-based and non-rank-based transformation methods. Non-rank-based methods assume a particular cumulative distribution function for the observed data, estimate the parameters of the chosen distribution and then use the fitted CDF to transform the observed data to estimated quantiles.

Rank-based quantile transformations do not assume a parameterized distribution, but instead use sample ranks. A rank can be the position of the sample in an ordered version of the samples. A simple rank-based transformation is the empirical cumulative distribution function

$$\hat{F}_X^E(x|\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n} I(x_i \leq x),$$

where $\mathbf{x} = [x_1, \ldots, x_n]$ is the samples and I the indicator function. After applying the empirical cumulative distribution function, the samples can then, if desired, be mapped to the normal distribution. A tiny complication is that the largest sample is mapped to 1 by the empirical CDF. Thus, straightforward application of Φ will map the largest sample $x_{(n)}$ to positive infinity. An easy fix for this would be to apply the transformation

$$\hat{G}^{-1}(y|\mathbf{x}) = \Phi^{-1}\left(\min\left\{\hat{F}_X^E(x|\mathbf{x}), 1-\epsilon\right\}\right),$$

where ϵ is some small value. An issue with using the empirical CDF to transform to approximately uniform distribution, is that it is biased toward larger values (it includes a map to 1, but not 0). An alternative option could be to use the transformation

$$\hat{F}_X(x|\mathbf{x}) = \frac{1}{n+1}\sum_{i=1}^{n} I(x_i \leq x).$$

This also solves the problem with directly applying Φ^{-1}, since 0 and 1 are never being mapped to. It can be shown for a sample vector $\mathbf{x}$ of unique values, that Section 4.1 maps the sample of rank i to the mean of the ith order statistic for a uniform distribution when drawing n samples. See Appendix B for a derivation of the kth order statistic of the Uniform$(0,1)$ distribution being Beta$(k, n-k+1)$ with mean $\frac{k}{n+1}$. Inspired by the usage of expected uniform scores (expected value of the kth largest observation in a sample of size n) as a mapping, an option for transformation to the standard normal distribution is to use expected normal scores. This rank-based transformation skips the intermediate transformation step to the uniform distribution and instead maps directly to the normal distribution. Unfortunately, no analytical expression is available for expected normal scores. Using numerical integration, the paper Harter [70] from 1961 provid the most complete table of expected normal values [68]. In Blom [71] an approximation to the expected value of the ith normal order statistic for a sample of size n is proposed as

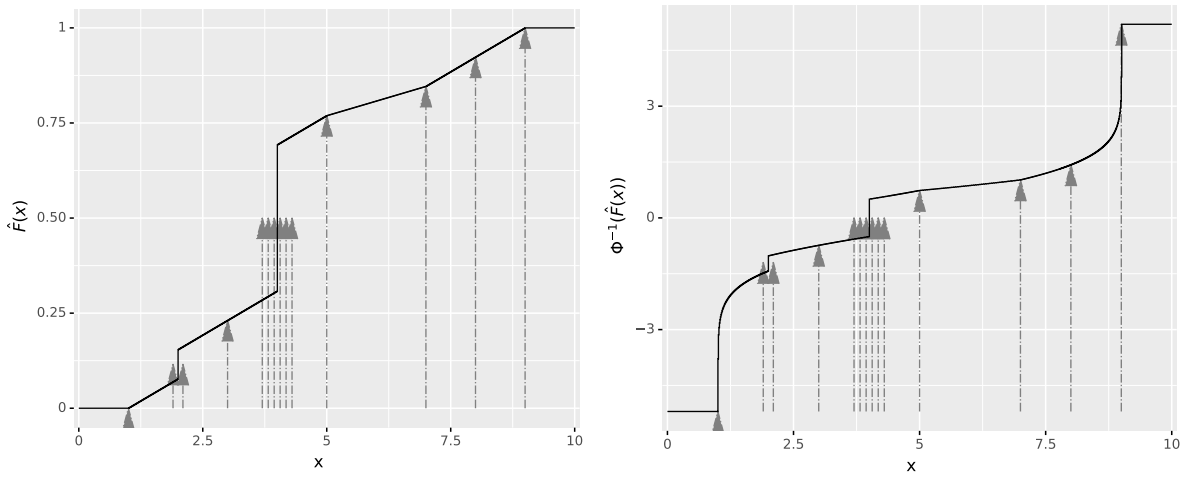
$$\mathrm{E}[z_{(i)}] = \Phi^{-1}\left(\frac{i-c}{n-2c+1}\right).$$

The paper Blom [71] proposes to use $c = \frac{3}{8} = 0.375$. The paper Beasley *et al.* [68] proposes to use $c = 0.363$ or, better yet, $c(n) = 0.314195 + 0.06333 \cdot \log_{10}(n) - 0.010895\left(\log_{10}(n)\right)^2$. More advanced approximations are also proposed.

Many other rank-based transformations can be thought of. For instance to use

$$\hat{F}_X^{\approx\text{scikit-learn}}(x|\mathbf{x}) = \frac{\left(\sum_{i=1}^{n} I(x_i \leq x\right) - 1}{n-1},$$

as an approximation to the CDF. This is basically what is implemented by the *quantile_transform* function and the *QuantileTransformer* class in the Python [56] package scikit-learn [60]. However, the scikit-learn package also implements linear interpolation and the option of choosing fewer (and possibly new) quantiles to define the transformation (which increases computational speed). By default, the scikit-learn package uses $n_{quantiles} = \min\{1000, n\}$ quantiles, where n is the number of observed values. We denote the actual approximation to the CDF implemented by scikit-learn as $\hat{F}_X^{\text{scikit-learn}}$. The scikit-learn package also has an option for quantile transformation to the standard normal distribution. This is defined as



(a) Quantile uniform transformation.

(b) Quantile normal transformation.

Figure 4.1: Example of quantile transformation with uniform and standard normal output distributions, respectively, given sample vector $\mathbf{x} = [1, 2, 2, 3, 4, 4, 4, 4, 4, 4, 5, 7, 8, 9]$. The solid black line represents the transformation function for a range of x-values, while the arrows represent the transformation for the initial samples in $\mathbf{x}$. For repeated values in $\mathbf{x}$ (for 2 and 4), the arrows are jittered a bit along the x-axis for easier visualization.

$$\left(\hat{G}_X^{\text{scikit-learn}}\right)^{-1} = \Phi^{-1}\left(\max\left\{\min\left\{\hat{F}_X^{\text{scikit-learn}}(x|\mathbf{x}), 1-10^{-7}\right\}, 10^{-7}\right\}\right).$$

Basically, this is just applying the inverse of the standard normal CDF to the values from the approximation to the cumulative distribution function of the samples whilst taking care of boundary extremities to avoid mapping to negative or positive infinity. The scikit-learn package also includes an option for inverting the quantile transformation. For the quantile normal transformation, this is done by using the transformation $(\hat{F}^{\text{scikit-learn}})^{-1}(\Phi(\cdot))$, while for the quantile uniform transformation this is done by using the transformation $(\hat{F}^{\text{scikit-learn}})^{-1}(\cdot)$. Inversion of the $\hat{F}^{\text{scikit-learn}}$ function is very quick and efficient when the chosen (possibly new) quantiles and their mappings are saved. For repeated quantiles (which can happen when there are repeated values in the sample vector $\mathbf{x}$) the $\hat{F}^{\text{scikit-learn}}$ transformation is not unique. In the scikit-learn package, this is solved by mapping to the middle of the range of possible values. See Figure 4.1 for an example of the quantile uniform transformation and an example of the quantile normal transformation when using the scikit-learn package.

4.2 Motivation for Creation of Randomized Quantile Transformation

The motivation for creating the randomized quantile transformation was that the author noticed that for some integer variables (columns) and some continuous variables with a large share of observations equal to zero, the quantile transformation had large gaps where no values were mapped to except exactly in the middle. Intuitively, we thought that this might be a problem due to two reasons. Firstly, the discriminator (or critic) could use these large gaps in the real dataset columns to distinguish between real and generated samples. Thus, the discriminator is given an unfair advantage. We reasoned that perhaps this unfair advantage might make the discriminator less effective in giving training directions to the generator, as it did not necessarily need to learn the difference in the generated and the desired distribution due to the presence of other indicators. This might

force the generator to "spend a lot of effort" learning to map exactly to the middle of these gaps, instead of only inside the gaps. This coincides with the second issue. If the generator does not see any observations inside these gaps, except exactly in the middle, how will it learn that it is allowed to utilize the entire gap (as the entire gap maps to the same value when inverting the quantile transformation)? The Randomized Quantile Transformation tries to solve this problem by stochastically mapping to an uniform interval centered around the center of each gap. A disadvantage of Randomized Quantile Transformation is that it introduces noise and is stochastic. Thus, it does not necessarily give the same result if applied twice to the same data. The introduction of noise will make it harder for the generator, but the benefits hopefully outweigh the disadvantages. The stochastic nature of the transformation should not be much of an issue when training a GAN network, but we are unsure how useful the transformation will be for other purposes. We fail to imagine how adding noise for no apparent reason for cases where there is no adversarial setup, and therefore no discriminator that otherwise might get an unfair advantage if noise was not introduced, will be helpful. We might, however, very well be proven wrong on this account. The quantile transformation is sometimes used in regression problems (for example quantile normal transformation has been used in many hydrological and meteorological applications [69] and gained popularity among genetic researchers [68]), but intuitively we do not believe randomized quantile transformation will be useful for this. It will make the result of the regression analysis stochastic, whilst no clear benefit is apparent. We note that Beasley *et al.* [68] are aware of a single quantile transformation with a stochastic element. While this approach has some theoretical advantage, Beasley *et al.* [68] state that the stochastic nature of the transformation seem to be discouraging to researchers and is rarely, if ever, used.

4.3 Mathematical Definition of Randomized Quantile Transformation

As seen in Section 4.1, quantile transformation can be implemented in a myriad of different ways. Consequently, as the Randomized Quantile Transformation (QTR) is only a slight modification on top of quantile transformation, a multitude of different implementations of the Randomized Quantile Transformation also exist. In this paper, we implement QTR on top of the quantile transformation from the scikit-learn package in Python.

First we define some notation. Let $\hat{F}$ be an approximation of the cumulative distribution function for the sample vector $\mathbf{x}$ and let $\{q_l\}_{l=1,\dots,n_{\text{quantiles}}}$ be the chosen quantiles (usually selected observations or interpolations from the sample vector $\mathbf{x}$). Let the references $\{r_l\}_{l=1,\dots,n_{\text{quantiles}}}$ be a set where each element is defined by the relation $q_l = \hat{F}^{-1}(r_l)$. Each reference r_l only maps to a single quantile q_l, but there can be repeated values in the set $\{q_l\}_{l=1,\dots,n_{\text{quantiles}}}$, thus multiple references can map to the same quantile value. Scikit-learn implements the set of references as $r_l = \frac{l-1}{n_{\text{quantiles}}}$ for $l = 1, \dots, n_{\text{quantiles}}$ and the set of quantiles as the corresponding percentiles of the sample vector $\mathbf{x}$. It solves the non-unique mapping problem for quantile values with multiple references as

$$
\begin{aligned}
r_{\text{low}(q_l)} &= \min\left\{r_m | \hat{F}_{-1}(r_m) = q_l\right\} \\
r_{\text{high}(q_l)} &= \max\left\{r_m | \hat{F}_{-1}(r_m) = q_l\right\} \\
\hat{F}_{\{q_l\}}(q_l) &= \frac{r_{\text{low}(q_l)} + r_{\text{high}(q_l)}}{2}.
\end{aligned}
$$

Due to interpolation, the approximate CDF for any quantile q is defined

$$
\begin{aligned}
q_{\text{below}(q)} &= \arg\min_{q_l} \{q - q_l | q_l \leq q\} \\
q_{\text{above}(q)} &= \arg\min_{q_l} \{q_l - q | q_l \geq q\} \\
\hat{F}(q) &= \hat{F}_{\{q_l\}}(q_{\text{below}(q)}) + \frac{q - q_{\text{below}(q)}}{q_{\text{above}(q)} - q_{\text{below}(q)}} \cdot \left(\hat{F}_{\{q_l\}}(q_{\text{above}(q)}) - \hat{F}_{\{q_l\}}(q_{\text{below}(q)})\right).
\end{aligned}
$$

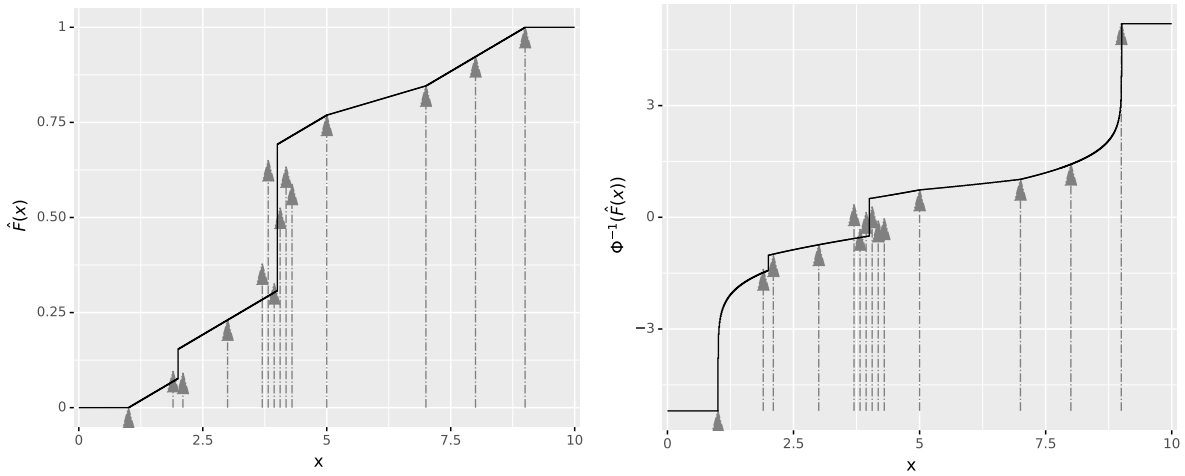
Similarly, due to interpolation, the inverse of the approximate CDF for any reference r is defined

$$
\begin{aligned}
r_{\text{below}(r)} &= \arg\min_{r_l} \{r - r_l | r_l \leq r\} \\
r_{\text{above}(r)} &= \arg\min_{r_l} \{r_l - r | r_l \geq r\} \\
\hat{F}^{-1}(r) &= \hat{F}^{-1}_{\{r_l\}}(r_{\text{below}(r)}) + \frac{r - r_{\text{below}(r)}}{r_{\text{above}(r)} - r_{\text{below}(r)}} \cdot \left(\hat{F}^{-1}_{\{r_l\}}(r_{\text{above}(r)}) - \{r_l\} F^{-1}(\hat{r}_{\text{below}(r)})\right),
\end{aligned}
$$

where $F^{-1}_{\{r_l\}}(r_l) = q_l$. The only change introduced by the Randomized Quantile Transformation is

$$
\begin{aligned}
r_{\text{high}(q)} &= \max\left\{\hat{F}(q), \left\{r_l | r_l = \hat{F}(q)\right\}\right\} \\
r_{\text{low}(q)} &= \min\left\{\hat{F}(q), \left\{r_l | r_l = \hat{F}(q)\right\}\right\} \\
u &\sim \text{Uniform}\left(\frac{-\delta_{\text{qtr\_spread}}}{2}, \frac{\delta_{\text{qtr\_spread}}}{2}\right) \\
\hat{F}^{\text{randomized}}(q) &= \hat{F}(q) + u \cdot \left(r_{\text{high}(q)} - r_{\text{low}(q)}\right),
\end{aligned}
$$

where $0 \leq \delta_{\text{qtr\_spread}} \leq 1$ is a hyperparameter that can be chosen by user. The inverse cumulative transformation, $\hat{F}^{-1}$, is unchanged in the Randomized Quantile Transformation. See Figure 4.2 for an example of the Randomized Quantile Transformation with uniform and standard normal output distribution, respectively. The sample vector $\mathbf{x}$ is the same for both Figure 4.1 and Figure 4.2 for easy comparison between the original quantile transformation and the Randomized Quantile Transformation.



(a) Quantile uniform transformation.

(b) Quantile normal transformation.

Figure 4.2: Example of Randomized Quantile Transformation with uniform and standard normal output distributions, respectively, given sample vector $\mathbf{x} = [1, 2, 2, 3, 4, 4, 4, 4, 4, 4, 5, 7, 8, 9]$. The solid black line represents the transformation function for a range of x-values, while the arrows represent the transformation for the initial samples in $\mathbf{x}$. For repeated values in $\mathbf{x}$ (for 2 and 4), the arrows are jittered a bit along the x-axis for easier visualization. The hyperparameter $\delta_{\text{qtr\_spread}}$ is set equal to 1.

Chapter 5

Counterfactual Synthesizing

Counterfactual explanations [8] is an emerging method within the toolbox of interpretable machine learning. A counterfactual explains a causal relationship by phrasing the argument on the form: "*if A did (or did not) happen, then B would (or would not) have happened*. For instance, if you burned your hand when you leaned on a hot stove, a counterfactual explanation could be: "*if you had not leaned on the stove, then you would not have burned your hand*".

Although counterfactual explanations in the context of interpretable machine learning is a quite new phenomena, counterfactual explanations date long back in social sciences such as philosophy and physchology [72]. Psychologists have shown that counterfactuals evoke causual reasoning in us humans [72] and counterfactuals are an important part of our cognitive lives [73]. The concept of using counterfactuals for causual reasoning has also been validated by philosophers [72]. In studies which compare counterfactual explanations to other explanation approaches, it has been shown that humans prefer counterfactual explanations over case-based reasoning (another reasoning approach based on giving examples) [72]. Fernández-Loría *et al.* [74] construct simple examples where counterfactual explanations perform better than feature importance methods.

The paper Verma *et al.* [75] list desirable properties of counterfactual explanations:

- **Actionability:** Counterfactual explanations should only consider changing features that are thought of as mutable and actionable. For instance a counterfactual explanation that change immutable features such as race or birth place might be of little value for many scenarios. We argue, however, that counterfactual explanations that modify such features can also be valuable with respect to detecting bias in a dataset [3] or just understanding the prediction model better.
- **Sparsity:** A counterfactual explanation is more understandable and actionable if it suggests to change only a few features, instead of many features. Verma *et al.* [75] argue that this holds true even if the magnitude of the individual changes are larger when changing only a few features.
- **Adherence to the data manifold:** If a counterfactual explanation is in close approximation to the data distribution used in training the underlying machine learning model, then research has suggested it is more likely to be actionable [75]. Intuitively, this makes sense. Imagine an 18 year-old just out of high school who receives a counterfactual observation where the only change is to change his job status from unemployed to retired. Even though only a single feature is changed, we argue that this counterfactual explanation is pretty useless for most scenarios. Most likely the 18 year old will have to wait at least 40 years before being able to retire. Additionally, since there are very few retired 18 year-olds, the counterfactual explanation is probably quite unrelatable to the 18 year old. Thus, it is neither very helpful as a suggested action nor as a tool for explaining or understanding the model decision.

- **Respect for causal relations:** If a counterfactual explanation is to be coherent with real-world relations, then a counterfactual explanation should comply with causal relations between features. For example, for many scenarios it will be unhelpful to propose a decrease in a person's age or academic degree. Additionally, the causal relations between features should be taken into account. Imagine an 18 year-old just out of high school who receives a counterfactual observation where the only change is a change in his level of education from high school to a PhD. If the counterfactual synthesizer should adhere to causal relations in the real-world, then it might make sense to suggest an increase in age as well, as the time required to get a PhD is normally multiple years.

Verma *et al.* [75] also list some desirable properties for the algorithm generating counterfactuals:

- **Black box and model agnostic counterfactual explanations:** An algorithm which is model agnostic and also applicable to black box models, is much more flexible than an algorithm that for instance requires access to gradients or only can be applied to a single type of models.
- **Fast generation of counterfactuals:** An algorithm which after a single optimization is able to generate counterfactual explanations for multiple datapoints (from the same data distribution and with the same prediction function), is both faster and easier to deploy [75].

Algorithm based counterfactual methods formulate the task of generating counterfactuals as an optimization problem that can be solved using methods for numerical optimization. Model based counterfactual methods, on the other hand, approach the task of generating counterfactuals by first learning the underlying probability distribution of the feature space and then using that probability distribution to generate counterfactual explanations. In Section 5.1 and Section 5.2, we give a brief overview of algorithm based and model based counterfactual methods, respectively. Although we mainly focus on model based counterfactual methods in this thesis, we believe algorithmic counterfactual methods are a natural part of an explanation of counterfactuals and give a unique insight into what a counterfactual explanation is and needs to be, compared to the insight provided by model based counterfactual methods. Finally, in Section 5.3, we proceed to describe the Wasserstein GAN based counterfactual framework introduced in this thesis.

Throughout this chapter we will use the following notation to discuss counterfactuals. Let $\hat{f} : \mathcal{X} \to \mathbb{R}$ be a prediction function for which we want to generate counterfactuals, where $\mathcal{X}$ is the feature space and $Y^* \in \mathbb{R}$ is a set of desired outcomes. The desired outcome Y^* can for instance be a single number, an interval or multiple intervals. The observation for which we want to generate counterfactual explanations we denote by $\mathbf{q}^*$ and a counterfactual observation we denote as $\mathbf{x}^*$.

5.1 A Brief Overview of Algorithm Based Counterfactual Methods

Perhaps the simplest optimization problem to generate a counterfactual explanation $\mathbf{x}^*$ corresponding to an observation $\mathbf{q}^*$ is

$$\arg\min_{\mathbf{x}^*, y^*} \hat{f}(\mathbf{x}^*) - y^*, \qquad y^* \in Y^*.$$

This optimization problem, however, does not reflect the fact that we generally want the counterfactual explanations to be close to the original observation. This can be incorporated into the optimization problem as

$$\arg\min_{\mathbf{x}^*, y^*} \max_{\lambda} \lambda(\hat{f}(\mathbf{x}^*) - y^*) + d(\mathbf{q}^*, \mathbf{x}^*), \qquad y^* \in Y^*, \tag{5.1}$$

where $d(\cdot,\cdot)$ is a function that punishes for diverging from the original observation $\mathbf{q}^*$ and λ balances the two terms. Let $\mathbf{X}^{obs}$ be the observed data and consist of the observations $\mathbf{x}^{[1]},\ldots,\mathbf{x}^{[n]}$, where each observation is a vector of size p. Wachter *et al.* [10] suggest using

$$d(\mathbf{x},\mathbf{x}^*) = \sum_{j=1}^{p} \frac{|x_j - x_j^*|}{MAD_j},$$
$$MAD_j = \text{Median}_{i\in\{1,\ldots,n\}}\left(\left|x_j^{[i]} - \text{Median}_{l\in\{1,\ldots,n\}}(x_j^{[l]})\right|\right),$$

but other functions can be used. Both of the above optimization problems are single-objective, despite Equation (5.1) wanting to minimize two goals, both proximity to original observation $\mathbf{q}^*$ and proximity to desired outcome Y^*. The solution to this was to optimize a collapsed weighted sum of two objectives. This process can be generalized to any number of objectives, but has the trade-off that the objectives must be balanced a-priori, which is increasingly difficult as we include more objectives. Other objectives that might be natural to include in the optimization is for instance the number of features that have been changed and proximity to other observed data points [11].

Counterfactual explanations with few features changed have the benefit of being easier to understand and if wanted, act upon. Proximity to other observed data points has the benefit of increasing the probability of the counterfactual explanations being plausible. This is an important aspect since implausible explanations can be less useful for understanding and much less useful for taking action. For instance a counterfactual explanation with a person of age 18 and ten years of work-experience is very unlikely. Since this combination is unlikely, it normally would be of little value even if it achieves one of the wanted outcomes. A related topic to this is actionability. Even if a counterfactual explanation is plausible, it might not be actionable for the observation to be explained. For instance it is impossible for a person to change race or become younger. Therefore, if actionability is needed or appreciated a solution could be to include restrictions to the parameter space in the optimization problem. For instance forcing race to be fixed and age to be fixed or at least bounded from below. This is often easily incorporated in algorithmic based counterfactual methods.

The difficult task of balancing objectives a-priori can be avoided by treating the generation of counterfactual explanations as a multi-objective optimization problem. If the objectives in the optimization problem are not ranked, there is a high possibility that there are cases where it is no longer possible to increase one of the objectives without decreasing at least one of the others. This is known as a Pareto frontier and is discussed in Section 5.1.1. For a multi-objective algorithm based generator of counterfactuals with non-ranked objectives, a natural choice might be to return samples attempting to represent the whole Pareto frontier.

5.1.1 Pareto Optimality

Given a set of criteria or objectives to optimize, Pareto optimality can be defined as a situation where none of the criteria can be optimized further without making any of the other criteria worse off [76]. A Pareto improvement is defined as a change in a situation where some objectives will become more optimal and none will become less optimal. If a Pareto improvement to a situation exists, then the situation is referred to as Pareto dominated. We also define that when a solution is a Pareto improvement compared to another situation, then it Pareto dominates the other situation. Notation-wise we define $p \prec q$ to be that p Pareto dominates q. The situation in which no Pareto improvements exists is called Pareto optimal. The set of all Pareto optimal situations are known as a Pareto frontier. In Figure 5.1 we plot an example of a Pareto frontier for the optimization problem

of minimizing both x_1 and x_2 when under the restrictions $x_2 + x_1^2 \geq 3$, $0 \leq x_1 \leq 4$ and $x_2 \leq 4$. The Pareto frontier for this optimization problem is $\{(x_1, 3 - x_1^2) | 0 \leq x_1 \leq 4\}$.

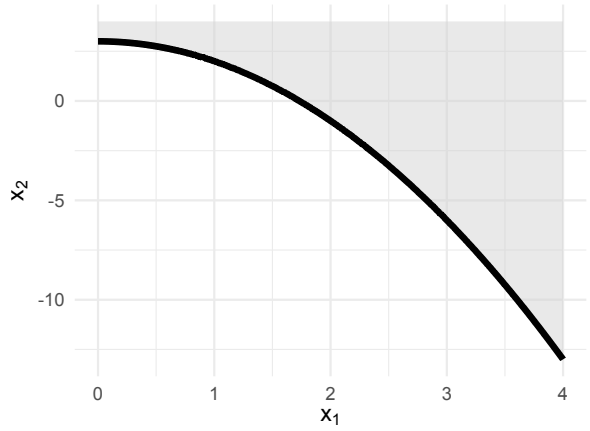


Figure 5.1: Example of Pareto optimality. The light grey area is the feature space of x_1 and x_2, while the thick black line is the Pareto frontier. The optimization problem in the example is to minimize both x_1 and x_2 when under the restrictions $x_2 + x_1^2 \geq 3$, $0 \leq x_1 \leq 4$ and $x_2 \leq 4$.

5.2 A Brief Overview of Model Based Counterfactual Methods

The process of fitting a model to approximate the feature space or counterfactual universe tend to be computationally intensive and time consuming. However, when the model is trained, it is usually very fast and efficient to generate new samples or counterfactuals, even for different observations to be explained (multiple $\mathbf{q}^*$). Some model based counterfactual methods only model the underlying distribution of the feature space $\mathcal{X}$ [4, 77] and use post-processing or optimization to find suitable counterfactual explanations for the observation $\mathbf{q}^*$. Other counterfactual methods model the feature space $\mathcal{X}$ in combination with the relation to the predictive function $\hat{f}$ [5, 12]. For this type of methods the observation to be explained $\mathbf{q}^*$ can be taken as input (along with potentially other queries) and then the model directly outputs relevant counterfactuals. Some post-processing or optimization can of course still be beneficial, but this is in theory not necessary.

The type of model used to capture the underlying data distribution differs. Two common approaches are, however, to use either GANs [5] or variational autoencoders (VAEs) [4, 12]. Generative adversarial networks are explained in Section 2.2, while variational autoencoders are explained in Section 3.1. Counterfactual methods using VAEs normally use the latent distribution as search space to generate counterfactuals. By utilizing the latent distribution as search space, the objective of closeness to other data points is usually automatically accomplished. Redelmeier *et al.* [77], on the other hand, utilize conditional inference trees to model the underlying data distribution. This is done by decomposing the probability distribution $P_{\mathbf{X}}(\mathbf{x})$ into products of conditional probability distributions $P_{X_j|X_1,\ldots,X_{j-1}}(x_j|x_1,\ldots,x_{j-1})$ and then using a conditional inference tree to model each conditional probability distribution. Redelmeier *et al.* [77] then utilize Monte Carlo sampling and a post-processing step to generate counterfactual explanations. Compared to using GANs or VAEs to model the data distribution, conditional inference trees are generally less computer intensive to train [77].

5.3 A Wasserstein GAN Based Counterfactual Synthesizer Framework – tabGANcf

In this section we present the author's take on using Wasserstein GAN to synthesize counterfactual observations. We restrict ourselves to the task of generating counterfactuals for a classification task prediction function $\hat{f}$, where the desired set of outcomes, Y^*, is to flip the prediction with respect to some threshold c^*. Thus, if the observation to be explained, $\mathbf{q}^*$, has a predicted value $\hat{f}(\mathbf{q}^*) < c^*$, then a generated counterfactual $\mathbf{x}^*$ should flip the prediction function such that $\hat{f}(\mathbf{x}^*) > c^*$ and vice versa. We refer to the two groups of observations that satisfy either $\hat{f}(\mathbf{x}) < c^*$ or $\hat{f}(\mathbf{x}) > c^*$ as being of different prediction classes. We further simplify the problem to only approximate the counterfactual universe for queries from a single prediction class. The problem of generating counterfactuals for queries from both prediction classes is easily solved by fitting two models. For simplicity of notation, we will for the rest of this chapter always assume we are generating counterfactuals to explain observations from the group $\hat{f}(\mathbf{x}) < c^*$, i.e. we want counterfactual observations that increase $\hat{f}$ above the threshold c^*.

When using tabular GAN as a counterfactual synthesizing method, we want the GAN to capture the counterfactual universe. That is, for an observation to be explained, $\mathbf{q}^*$, we want the generator to be able to produce a counterfactual $\mathbf{x}^*$ drawn from a distribution of observations that flip the prediction class, whilst still adhering to the original data distribution. We do not want counterfactual explanations that are very unlikely to have originated from the given data distribution (usually the same distribution as the prediction function $\hat{f}$ is trained on), because then they would, as discussed in the introduction to this chapter, probably be less actionable or relatable. Ideally, we also want the generated counterfactuals to be close to the original observation, $\mathbf{q}^*$, and suggested changes to be sparse.

The framework which we present in this section, attempts to capture the counterfactual universe in such a way. It builds upon the the data synthesizing framework defined in Section 3.2 and thus have much of the same customizability as the tabGAN framework, at least with respect to the functionality that is transferable to the counterfactual synthesizing application. We name this framework tabGANcf, although calling it a framework might be a bit of an exaggeration. It is very much still in the proof-of-concept phase and not all intended ideas are implemented yet. Thus, this section functions more as a collection of different possible implementations for a counterfactual synthesizer that the author believes have potential, instead of an overview of what is already implemented in the framework, unlike what we did in Section 3.2 for the tabGAN data synthesizing framework. We will, however, spend some time explaining what has been implemented, as we will use it for an exploratory analysis in Chapter 8. The idea of using tabular GAN as a model for capturing the counterfactual universe for a specific dataset and machine learning classifier is inspired by the papers Yang *et al.* [5] and Nemirovsky *et al.* [78].

In Section 5.3.1, we explain how we redefine the architecture, loss function and training function of the tabGAN framework to accomodate the counterfactual synthesizing task. Next, in Section 5.3.2 we present some possible extensions or ideas that perhaps could be benefical. Not all of them are implemented in the tabGANcf framework yet.

5.3.1 Implementation

Similar to the tabGAN framework, we perform preprocessing such that the data observations are on a format acceptable as both input and output to a neural network. Since the tabGANcf framework builds upon the tabGAN framework, we automatically include the same preprocessing options as described in Section 3.2.1. We continue with the notation from Chapter 3 where we used $'$ to denote

a preprocessed observation. Thus, we denote a counterfactual query (observation to be explained) in the preprocessed format as $\mathbf{q}^{*\prime}$, and a counterfactual explanation in the preprocessed format as $\mathbf{x}^{*\prime}$.

For the generator to be able to produce counterfactual explanations tailored to each specific observation to be explained, we believe a reasonable approach is to feed the generator the counterfactual query, $\mathbf{q}^{*\prime}$, in addition to the the latent noise vector, $\mathbf{z}$. Then, the generator can potentially learn to generate from the conditional distribution of each counterfactual query. To guide the generator into learning to generate counterfactuals, a natural approach is to change the loss function used to train the GAN network. Yang *et al.* [5] do this by adding a counterfactual loss term, L^{cf}. For a Wasserstein GAN, the loss function which the generator wants to minimize, can then be written as

$$L = E_{\mathbf{x}\sim P_{\mathbf{X}}}[D(\mathbf{x})] - E_{\substack{\mathbf{z}\sim P_{\mathbf{Z}} \\ \mathbf{q}^*\sim P_{\mathbf{Q}^*}}}\left[D(G(\mathbf{z}|\mathbf{q}^{*\prime})) + L^{\text{cf}}(G(\mathbf{z}|\mathbf{q}^{*\prime}))\right],$$

where we assume the queries are drawn from a distribution $P_{\mathbf{Q}^*}$ and for readability we omit the gradient penalty term that enforces the critic to be 1-Lipschitz, as it only affects the critic. For many applications, the counterfactual query and the data observations might be drawn from the same distribution, i.e. $P_{\mathbf{Q}^*} = P_{\mathbf{X}}$, but it can very well be drawn from a different distribution. Here, we include the counterfactual query as input to the generator only, but the counterfactual query can very well be given as input to the critic as well. This is done in Yang *et al.* [5]. However, as they do not specify how they define the counterfactual query, it might very well be differently defined than what we use in this thesis, where the counterfactual query $\mathbf{q}^*$ is the observation to be explained.

The paper Yang *et al.* [5] use cross-entropy loss to force the prediction function value of the counterfactual explanation, i.e. $\hat{f}(\mathbf{x}^*)$, to be of the correct class, as well as an unspecified regularization term to force the explanation to be close to the original observation. This is an elegant solution to the problem, but it forces the prediction function to be differentiable and the gradients of the prediction function to be available to the counterfactual synthesizer during training, at least if gradient-based learning is to be used. Additionally, another problem which Yang *et al.* [5] do not specify how they solve, is the process of calculating derivatives when the generated counterfactual explanation $\mathbf{x}^{*\prime}$ must be transformed to the original data format, i.e. becomes $\mathbf{x}^*$, before being input into the prediction function $\hat{f}$. Especially for the one-hot encoded discrete columns, this is a problem as they are transformed into a non-numerical data format and then gradients are not applicable. Additionally, any preprocessing steps done inside $\hat{f}$ would also have to be differentiable. Perhaps, Yang *et al.* [5] solve this by letting the prediction function take the input in the same preprocessed format as the neural networks, such that $\mathbf{x}^{*\prime}$ directly can be input into $\hat{f}$. Alternatively, a less restrictive solution could be to only let the discrete columns that are one-hot encoded be directly input into $\hat{f}$, while the numerical columns are transformed using a differentiable function. However, suddenly the solution appears less elegant and it puts restrictions on the input to the prediction function $\hat{f}$. Restrictions such as these may limit the number of suitable applications for the counterfactual synthesizer, or they might force the creators of the prediction function to plan ahead or make changes to the prediction function if they desire to fit a model based synthesizer to it.

Consequently, we desire to avoid calculating gradients through the prediction function. The paper Nemirovsky *et al.* [78] proposes an alternative approach for classifier prediction functions that are either non-differentiable or black-box. Of course, the method can also be used for differentiable classifiers. Instead of computing the gradients of the prediction function, $\hat{f}$, they propose to reweight each real data observations according to the prediction function value when inserted into the loss function. Nemirovsky *et al.* [78] do this for a residual GAN with standard loss (i.e. not Wasserstein

loss), but we believe this approach might be extendable to regular GAN and Wasserstein loss. For a batch of real observations $\left\{\mathbf{x}^{[i]}\right\}_{i=1,\ldots,n_{\text{batch}}}$ and a batch of queries $\left\{\mathbf{q}^{*\prime[i]}\right\}_{i=1,\ldots,n_{\text{batch}}}$, the loss function can be calculated as

$$L = \frac{\sum_{i=1}^{n_{\text{batch}}} \hat{f}(x^{[i]}) \cdot D\left(\mathbf{x}^{[i]}\right)}{\sum_{i=1}^{n_{\text{batch}}} \hat{f}\left(x^{[i]}\right)} - \frac{1}{n_{\text{batch}}} \sum_{i=1}^{n_{\text{batch}}} D\left(G\left(\mathbf{z}^{[i]}|\mathbf{q}^{*\prime[i]}\right)\right),$$

where $\left\{\mathbf{z}^{[i]}\right\}_{i=1,\ldots,n_{\text{batch}}}$ is a batch of latent noise vectors.

The reweighting gives more importance to the observations with larger prediction function. Recall that we for simplicity defined that the counterfactual queries are from the group $\hat{f}(\mathbf{x}) < c^*$ and that we want to generate counterfactual explanations from the group $\hat{f}(\mathbf{x}) > c^*$. In practice, this reweighting gives the same result as if we had drawn the real observations from the probability distribution $P_{C_{\hat{f}}}$ defined as

$$P_{C_{\hat{f}}}(\mathbf{x}) = C \cdot \hat{f}(\mathbf{x}) \cdot P_{\mathbf{X}}(\mathbf{x}),$$

where C is a normalizing constant[1]. In fact, Nemirovsky *et al.* [78] prove that if the generator has sufficient capacity and the discriminator is systematically allowed to reach its optimum, then the output distribution of the generator converges to $P_{C_{\hat{f}}}$.

We struggle a bit with understanding why specifically this reweighting scheme was chosen by Nemirovsky *et al.* [78], other than perhaps for its simplicity. With this reweighting choice the optimal output distribution will probably contain some samples with a low prediction function value. In theory, any function g could be used to reweight the real data samples. Of course, Nemirovsky *et al.* [78] did not specify that they just wanted to flip the prediction class with respect to some threshold as we do, but we believe it is reasonable to assume that this will be a common usage of counterfactual explanations. For instance, returning to the example from the introduction of this thesis with a bank that uses a black box model to decide who receives a loan, a counterfactual explanation would generally be desired to just flip the prediction. In the event wherein counterfactual explanations with higher prediction function value are preferred even after flipping the prediction class, then a potential reweighting function could be $g(\mathbf{x}) = I\left(\hat{f}(\mathbf{x}) > c^*\right) \cdot \hat{f}(\mathbf{x})$.

If the only preference with respect to the prediction function is to flip the prediction class, then we believe a reasonable choice is to use $g(\mathbf{x}) = I\left(\hat{f}(\mathbf{x}) > c^*\right)$ as the reweighting function. In practice, this would be the same as if we only fed the critic observations from the real data distribution of the correct prediction class. It would also be more efficient, since with the reweighting function $g(\mathbf{x}) = I\left(\hat{f}(\mathbf{x}) > c^*\right)$ all real observations of the wrong prediction class contribute nothing to the loss function. Along with the original reweighting option used by Nemirovsky *et al.* [78], we also implement the option to only feed the critic samples from the correct prediction class.

Until now, we have only modified the loss function to influence the generator to produce samples of the correct prediction class. However, similar to Nemirovsky *et al.* [78] and Yang *et al.* [5], we include the option for a regularization term that will influence the generator to synthesize counterfactual observations that are close to the counterfactual query. Nemirovsky *et al.* [78] use a linear combination of the 1-norm and the squared 2-norm applied to the residual changes (difference between counterfactual query and generated counterfactual explanation). We do something similar, but we want to penalize the discrete variables differently. Reusing much of the same notation as Chapter 3, we let the continuous columns of $\mathbf{q}^{*\prime}$ be denoted by $\mathbf{c}^{*\prime}$ and the one-hot encoded columns by $\mathbf{d}^{*\prime} = \mathbf{d}_1^{*\prime} \oplus \ldots \oplus \mathbf{d}_{N_D}^{*\prime}$. Additionally, we let the counterfactual explanation produced by

[1]Explicitly, the normalizing constant $C = \left(\int \hat{f}(\mathbf{x}) \cdot P_{\mathbf{X}}(\mathbf{x}) dx\right)^{-1}$, but it does not need to be computed for this application.

the generator as explanation for observation $\mathbf{q}^{*\prime}$, be denoted $G\left(\mathbf{z}|\mathbf{q}^{*\prime}\right) = \hat{\mathbf{c}}^{*\prime} \oplus \hat{\mathbf{d}}^{*\prime}$, where $\hat{\mathbf{c}}^{*\prime}$ is a vector with the continuous columns and $\hat{\mathbf{d}}^{*\prime} = \hat{\mathbf{d}}_1^{*\prime}, \oplus \ldots \oplus \hat{\mathbf{d}}_{N_D}^{'}$ is a vector of the one-hot encoded discrete columns. Then, the regularization term we use can for a single observation be written as

$$l\left(G\left(\mathbf{z}|\mathbf{q}^{*\prime}\right), \mathbf{q}^{*\prime}\right) = \alpha\|\hat{\mathbf{c}}^{*\prime} - \mathbf{c}^{*\prime}\|_1 + \beta\|\hat{\mathbf{c}}^{*\prime} - \mathbf{c}^{*\prime}\|_2^2 + \gamma \sum_{j=1}^{N_d} \sum_{j=1}^{l_j} d_{j,k}^{*\prime} \cdot \left(1 - \hat{d}_{j,k}^{*\prime}\right),$$

where α, β and γ are hyperparameters chosen by the user. We have not done enough experimenting to recommend good default values for these. However, for the exploratory analysis in Chapter 8 we mainly utilize $\alpha = 1$, $\beta = 0$ and $\gamma = 2$. It is a bit arbitrarily chosen, but we believe perhaps $\alpha = 1$ might by itself reason okay with the approximation of the Earth-Mover distance which we make the critic approximate by being 1-Lipschitz, as the Earth-Mover distance intuitively is a measurement of the mass that must be transported to transform one probability distribution into another. In **??** We set $\beta = 0$, as we believe sparse counterfactuals are preferable and increasing the β parameter might be a bit counterintuitive to this, as the squared 2-norm penalizes less harshly for changes close to the original value compared to the same change far away from the original value. We set $\gamma = 2$ to approximately penalize the discrete variables the same as the continuous variables, but, again, we stress that this is not based on a hyperparameter search, unlike the default values in the tabGAN framework.

5.3.2 Extensions

In the previous section, we introduced two options for guiding the generator to produce samples correctly classified by a black-box classifier, $\hat{f}$; either reweighting the real data observations to be proportional to their prediction function value, or only feeding the critic real data observations with the correct prediction class without any reweighting. We also introduced a regularizing term to be added to the loss function to guide the generator in producing counterfactual explanations near the counterfactual query, $\mathbf{q}^*$. The discrete nature of the discrete columns makes it easier for the generator to produce sparse changes for the discrete variables of the observations, however, for the continuous column values it is more difficult. The 1-norm regularization penalty might contribute to the generator wanting to also produce sparse changes for the continuous columns, but the nature of a neural network will practically make it impossible for it to recreate the values of $\mathbf{q}^{*\prime}$ exactly, even if it desired to do so.

As a fix to this, we propose two interconnected extensions. Firstly, we propose to change the generator architecture to also have a stochastic nature for the numerical output layer, not only for the subparts of the discrete output layer with their Gumbel activation functions. This change would, for each numerical output value, consist of having a certain probability of being set to the same value as the counterfactual query, $\mathbf{q}^{*\prime}$. Each of these probabilities could be a learnable parameter by the neural network, such as a bias node, but we believe a better option is that it is calculated for each generated sample as a linear combination of values from the last hidden layer, where the weights and bias in the linear combination are the learnable parameters. Potentially, the neural network might also utilize the difference between each intended numerical output value (without the stochastic numerical component) and the corresponding value of the counterfactual query, when determining the probabilities. The architecture of the generator could for example be of the form

$$
\begin{cases}
\textit{Input: } \mathbf{z}, \left(\mathbf{q}^{*\prime} = \mathbf{c}^{*\prime} \oplus \mathbf{d}^{*\prime}\right) & \text{(latent vector and counterfactual query)} \\
\mathbf{h}_1 = BN\left(GELU\left(FC_{l_z \to l_{h,g}}\left(\mathbf{z} \oplus \mathbf{q}^{*\prime}\right)\right)\right) \\
\mathbf{h}_2 = BN\left(GELU\left(FC_{l_{h,g} \to l_{h,g}}\left(\mathbf{h}_1\right)\right)\right) \\
\hat{\mathbf{c}}^{*\prime} = FC_{l_{h,g} \to l_{N_C}}\left(\mathbf{h}_2\right) \\
\mathbf{h}_{\text{logits}} = FC_{l_{h,g} \to N_C}\left(\mathbf{h}_2 \oplus \left(\hat{\mathbf{c}}^{*\prime} - \mathbf{c}^{*\prime}\right)\right) \\
\mathbf{h}_{\text{probs}} = \sigma\left(\mathbf{h}_{\text{logits}}\right) \\
\mathbf{b} = Bernoulli\left(\mathbf{h}_{\text{probs}}\right) \\
\hat{\mathbf{c}}^{*\prime}_R = \mathbf{b} \cdot \hat{\mathbf{c}}^{*\prime} + (1 - \mathbf{b}) \cdot \mathbf{c}^{*\prime} \\
\hat{\mathbf{d}}^{*\prime} = Gumbel_\tau\left(FC_{l_{h,g} \to l_1}\left(\mathbf{h}_2\right)\right) \oplus \cdots \oplus Gumbel_\tau\left(FC_{l_{h,g} \to l_{N_D}}\left(\mathbf{h}_2\right)\right) \\
\textit{Output: } \hat{\mathbf{c}}^{*\prime}_R, \hat{\mathbf{d}}^{*\prime},
\end{cases}
$$

where $\mathbf{h}_{\text{logits}}$ is the logits (logarithmic odds) of the output numerical values not being set to the original value from the counterfactual query. We let $Bernoulli(\cdot)$ denote a function that elementwise samples from a Bernoulli distribution with the input probability, while σ is the sigmoid function previously defined. The sigmoid function transforms the logit values into probabilities. The new numerical output from the generator neural network is then $\hat{\mathbf{c}}^{*\prime}_R$, where some elements may be stochastically chosen to be set to values from the counterfactual query, $\mathbf{q}^{*\prime}$. With this, we believe the generator would stand a better chance of producing sparse changes also for the numerical values. However, to further encourage sparse changes, we believe an option is to include a term in the loss function that penalizes for high values in the layer $\mathbf{h}_{\text{logits}}$, i.e. the generator would be penalized for choosing high probabilities of the numerical output values being different from those of the counterfactual query. As of now, this is fully theoretical and has not been implemented or tested.

An extension we have implemented is the option to mix outputs from the generator in the loss function with real observations of the correct prediction class. The idea behind this, is that in the previously introduced approaches, we encourage the generator to replicate the distribution of the reweighted real data distribution or alternatively the distribution of the real observations with correct prediction class. However, we see no reason for why the counterfactuals produced by the generator should completely follow this distribution. Even though we want the generated counterfactuals to not be improbable, this is not the same as replicating the entire distribution. For example, imagine a large square with a small circle inside it. Let the inside of the circle be uniformly filled with observations from one prediction class and the outside of the circle, but still inside the square, be uniformly filled with observations from the second class. Imagine we want to create a counterfactual synthesizer that given a counterfactual query from inside the circle, spits out a counterfactual observations from outside the circle. A successful counterfactual synthesizer might then map each counterfactual query to just outside the circle. Intuitively, we see no reason why we should desire the counterfactual synthesizer to map to the far away corners of the square as often as it maps close to the circle boundary. By mixing the generator samples in the loss function with real observations of the correct prediction class, we intuitively hope that this will allow the counterfactual synthesizer to map less often to the undesirable parts of the real data distribution (such as the far away corners of the square in the example).

Another extension, which is almost the opposite of the last optionable extension, is to mix outputs from the generator in the loss function with real observations of the wrong prediction class. This is intended as an option when we only feed the critic real observations with the correct prediction class. The idea is that, when we only feed the critic a subset of the real data distribution, we potentially miss out on utilizing information that could help the critic in its guiding of the gener-

ator to produce realistic counterfactuals of the correct class. By showing the critic more examples of what is not desired, we could potentially make it easier for the generator to learn. However, we are unsure of how effective this will be in practice and especially for Wasserstein GAN.

Due to constraints of space and time, we unfortunately, perform no testing of the first and third extension explained in this section and only very informal testing of the second extension explained in this section. Thus, these are mostly just theoretical ideas that the author think might be interesting to explore at a later time. For completeness, we also briefly list a few more possible extensions, such as the ability to keep some columns unchanged. This could be decided before training the model or more flexibly at sampling time, either way it would help keep the counterfactuals actionable. Another extension could be to employ a different sampling technique to better approximate the counterfactual universe for the rare queries, as done with the umbrella sampling technique in Yang *et al.* [5]. Yang *et al.* [5] also investigate including known causal dependence between attributes into models by changing the generator architecture.

Chapter 6

Datasets

In this chapter we introduce the datasets that we will use to evaluate and compare our implemented methods against state-of-the-art methods. In total we will use two synthetic (simulated) datasets and four real datasets. The two simulated datasets are explained in Section 6.1 and Section 6.4. Out of the real datasets, the Adult dataset from the UCI Machine Learning Repository [79] is the one we will use the most. Therefore, we give a quite detailed explanation of the Adult dataset in Section 6.2. For the remaining three datasets, which we will use solely to evaluate machine learning efficacy, we give a brief explanation in Section 6.3.

6.1 Syn2D_3cats Dataset

In this thesis, we create the Syn2D_3cats dataset, which is a simulated dataset specifically designed for visual evaluation of the training progress of a GAN on tabular data. The dataset has two numerical columns, x_1 and x_2, and one categorical column x_3. The dataset has three variables, but since one of them is categorical, the dataset is easily plotted in two dimensions. In total the dataset has 1000 observations.

The discrete column, x_3, is uniformly sampled from the list $\{cat1, cat2, cat3\}$. Conditioned on x_3 being either $cat1$ or $cat2$, the first numerical column is sampled uniformly from the range $[0, 10]$, else x_1 is sampled uniformly from the range $[1, 3]$. Mathematically this can be described as

$$[x_1|x_3] \sim \begin{cases} Uniform[0, 10] & \text{if } x_3 \in \{cat1, cat2\} \\ Uniform[1, 3] & \text{if } x_3 = \{cat3\}. \end{cases}$$

The distribution of the second numerical column is dependent upon the values of x_1 and x_3. The conditional distributions of x_2 given x_1 and x_3 are defined as

$$[x_2|x_1, x_3] \sim \begin{cases} \mathcal{N}(x_1^2 - 50x_1 + 25, x_1^2) & \text{if } x_3 = cat1 \\ \mathcal{N}(x_1^3 - 13x_1^2 + 31x_1 + 45, x_1^2) & \text{if } x_3 = cat2 \\ Uniform[20, 10 + 10x_1] & \text{if } x_3 = cat3, \end{cases}$$

where $\mathcal{N}(\mu, \sigma^2)$ is a normal distribution with mean μ and variance σ^2. In Figure 6.1, we plot a subset of the training observations of the Syn2D_3cats dataset. The plot visualizes clearly how the joint distribution of (x_1, x_2) is dependent upon the x_3 value.

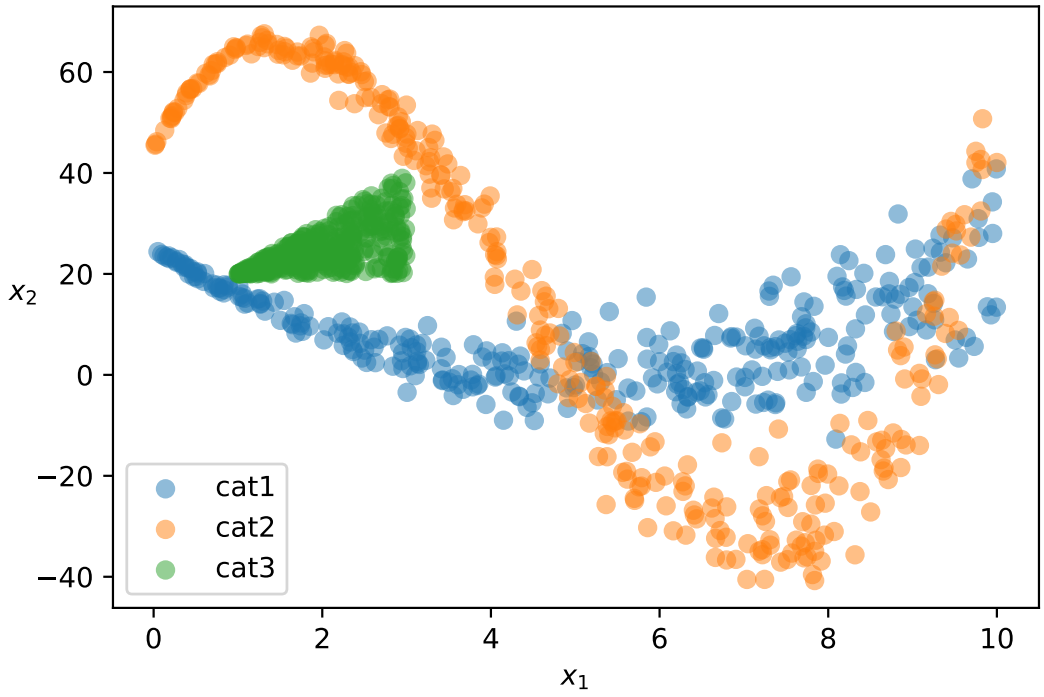


Figure 6.1: Visualization of a subset of 1000 observations from the Syn2D_3cats dataset with x_1 on the horizontal axis and x_2 on the vertical axis. The x_3 variable value gives the color of each plotted observation. The observations have opacity set to 0.5 to give a better visualization of the density of observations.

6.2 Adult (Census Income) Dataset

The Adult dataset, also known as the Census Income dataset, is extracted from the 1994 Census bureau database [79]. The column intended as variable of interest is the annual income (binarized to $> 50K$ and $\leq 50K$ US dollars). The dataset consists of 48842 observations and 15 columns. Two thirds of the observations are pre-classified as part of the training set and one third as part of the test set. This results in a training set of size 32561 and a test set of size 16281. Both train and test sets do, however, contain missing values. We choose to remove all observations with at least one missing value. The new sizes of the training and test sets are respectively 30162 and 15060, in total 45222 observations. We refer to the training and test observations of the Adult dataset as respectively the Adult train dataset and the Adult test dataset.

Some of the discrete columns in the Adult dataset are very imbalanced, for instance the "native_country" column where 91% of the observations are from the United-States whilst the other categories each account for less than a single percent. To simplify the dataset a bit we merge some categories. This entails for instance merging all non-US countries into one category for the "native_country" column and combining *Married-AF-spouse*, *Married-civ-spouse* and *Married-spouse-absent* to a single category *Married* for the "marital_status" column. We also choose to delete the "relationship" column that represents each observation's role in the family, as its categorical definitions were a bit unclear and the information was already partially contained in the "marital.status" column and the "gender" column. For the full description of the preprocessing steps we refer to Appendix D. In Table D.1 in Appendix D, we also give a complete list of columns and categories

for the unprocessed (except for removing missing values) Adult dataset. The remaining columns after preprocessing are listed in Table 6.1. In the table we give the value range for each numerical column as well as the categories with percentage shares for each discrete column.

Table 6.1: Overview of each column in the Adult dataset. For numerical columns the value range is reported, while for discrete columns the categories along with percentages are reported. The percentages for all categories of a single discrete column sum to 1.

Column	Values (numerical range or categories)
age	[17, 90]
workclass	Government (14%), Private (74%), Self-emp-inc (3.6%), Self-emp-not-inc (8.4%), Without-pay (0.046%)
fnlwgt	[13492, 1490400]
education	<=12th (13%), Assoc-acdm (3.3%), Assoc-voc (4.3%), Bachelors (17%), Doctorate (1.2%), HS-grad (33%), Masters (5.6%), Prof-school (1.7%), Some-college (22%)
educational_num	[1, 16]
marital_status	Divorced (14%), Married (48%), Never-married (32%), Separated (3.1%), Widowed (2.8%)
occupation	Adm-clerical (12%), Armed-Forces (0.031%), Craft-repair (13%), Exec-managerial (13%), Farming-fishing (3.3%), Handlers-cleaners (4.5%), Machine-op-inspct (6.6%), Other-service (11%), Priv-house-serv (0.51%), Prof-specialty (13%), Protective-serv (2.2%), Sales (12%), Tech-support (3.1%), Transport-moving (5.1%)
race	Native-American-Inuit (0.96%), Asian-Pac-Islander (2.9%), Black (9.3%), Other (0.78%), White (86%)
gender	Female (32%), Male (68%)
capital_gain	[0, 99999]
capital_loss	[0, 4356]
hours_per.week	[1, 99]
native_country	Non-US (8.7%), US (91%)
income	<=50K (75%), >50K (25%)

Many of the columns in Table 6.1 are self-explanatory, but we will explain those that are not. The column name "fnlwgt" is an abbreviation for final weight and is a continuous measurement for how many units in the target population the observation represents. "capital_gain" and "capital_loss" columns represent respectively capital gained and capital lost from other investment sources than salary/wage. "hours_per_week" are the number of hours worked per week.

We observe that even after preprocessing there are still discrete columns with a lot of categories. The Adult dataset can therefore be a challenging dataset with respect to learning the underlying data distribution.

6.3 Three Other Real Datasets

In this section we present the three real datasets, the Covertype dataset[1] found in the UCI Machine Learning Repository [79], the Credit Card Fraud Detection dataset[2] from Kaggle [3], and the Online

[1]Url to the Covertype dataset: `https://archive.ics.uci.edu/ml/datasets/covertype`
[2]Url to the Credit Card Fraud Detection dataset: `https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud`
[3]An online data science platform. Url: `https://www.kaggle.com/`

News Popularity dataset[4] also found in the UCI Machine Learning Repository.

6.3.1 Covertype Dataset

The Covertype dataset is created from data originally obtained through US Geological Survey (USGS) and US Forest Survey (USFS) data [79]. The response variable in the dataset is forest covertype. Each observation represents a $(30x30)\text{m}^2$ cell in four wilderness areas situated in the Roosevelt National Forest of northern Colerado. The cover types are likely to be the result of mostly ecological processes (not forest management practices), since the areas' exposure to human intervention is minimal. The response variable was collected from US Forest Service (USFS) Region 2 Resource Information System (RIS) data , while the other variables originate from processed versions of data from collected through US Geological Survey (USGS) and US Forest Survey (USFS) data [79]. In addition to the response variable "Cover_Type", there are two other categorical variables. These are originally one-hot encoded as 40 binary columns for soil type and 4 binary columns for wilderness area. To inform the data synthesizers that these columns are for a single variable, we reverse the one-hot encoding for each of them and store the information in two columns "Soil_Type" and "Wilderness_Area". The response variable is originally label encoded as integers from 1 to 7, however, we convert this to seven categories *CoverType1*, ..., *CoverType7* to highlight for the data synthesizers that this is a single categorical column, instead of being of continuous or integer type. For column descriptions we refer to Table E.1 in Appendix E.1. In the same section a list of value range for each numerical columns and categories for each discrete column can be found as well. The dataset has $581,012$ observations in it.

6.3.2 Credit Card Fraud Dataset

The Credit Card Fraud Detection dataset, or as we will refer to it from now on, the Credit Card Fraud dataset, represents transactions from two days in September 2013 by European cardholders. The response variable "Class" represents whether the transaction was a fraud or not. The dataset is very imbalanced. Out of the $284,807$ transactions (observations) in the dataset, only 492 (0.172%) are frauds. Due to privacy reasons, most of the original features are transformed using principal component analysis, where the first 28 principal components are included. These columns are named $V1, \ldots, V28$. The only two features which are not transformed, is "Time" and "Amount". The feature "Amount" contains the transaction amount for each observation, while the feature "Time" represents the time each transaction is executed. It is measured in seconds since the first transaction in the dataset. Originally, the response variable was on the format 1 for fraud and 0 otherwise, however, to signify to the data synthesizers that this is a discrete column, we transform the column such that *Class1* represents fraud and *Class0* represents not fraud.

6.3.3 Online News Popularity Dataset

The Online News Popularity Dataset was collected by Fernandes *et al.* [80] and donated to the UCI Machine Learning Repository. The dataset contains information about articles posted on the news and media website Mashable[5] between the 7th of January 2013 and 7th of January 2014. A small portion of the articles were discarded due to not following the standard HTML structure (and therefore requiring specific steps to process), but most were processed and added to the dataset. In total the dataset contains 39000 rows (articles). The authors of [80] extracted a total of 47 features. The response variable is the number of shares on Mashable and has the column name "shares" in

[4]Url to the Online News Popularity dataset: `https://archive.ics.uci.edu/ml/datasets/online+news+popularity`
[5]Url to Mashable: `https://mashable.com/`

the dataset. In the UCI Machine Learning Repository, the dataset is associated with a classification task of whether "shares" ≥ 1400. There are $18,490$ observations with shares < 1400 and $21,154$ observations with shares ≥ 1400. For a description of each column, we refer to Table 2 in Fernandes *et al.* [80]. This article also gives more information of how the data are fetched and processed. In Table E.4 in Appendix E.2, we give an overview of the value range for each numerical column and the categories for each discrete column in the Online News Popularity dataset.

6.4 Syn_Moons Datasets

We simulate a dataset with two continuous columns, x_1 and x_2, that consists of two interleaving half circles. The observations are simulated as a position on one of the half circles plus some Gaussian noise with standard deviation equal to 0.08. In total we choose to simulate 5000 observations. A dataset with two noisy interleaving half circles is a common dataset type used in counterfactual syntheszing papers such as Yang *et al.* [5] and Nemirovsky *et al.* [78], and we use the function *make_moons* in the scikit-learn package to simulate the dataset. In Figure 6.2 we plot a subset of 2500 observations from the Syn_Moon dataset. As background we plot a contour plot of the estimated probabilities of a support vector machine model trained to classify which moon each observation belongs to (see introduction to Chapter 8).

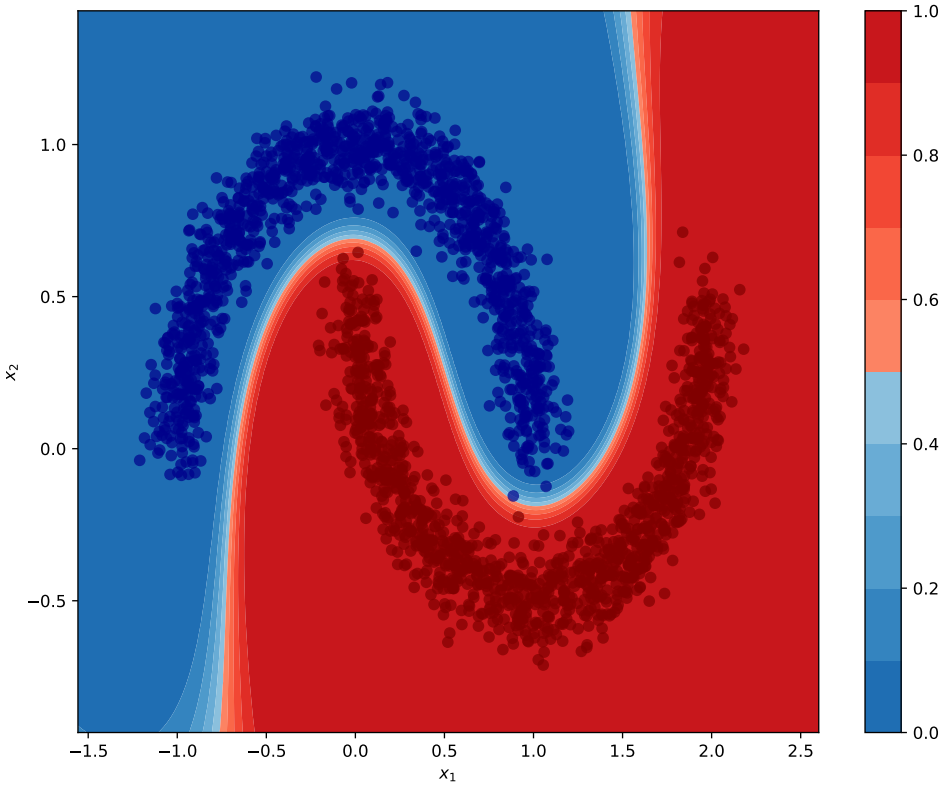


Figure 6.2: Visualization of a subset of 2500 observations from the Syn_Moon dataset with x_1 on the horizontal axis and x_2 on the vertical axis. The observations have opacity set to 0.7 to give a better visualization of the density of observations. As background, a contour plot of estimated probabilities of a support vector machine model trained to classify which moon each observation belongs to is plotted (see introduction to Chapter 8).

Chapter 7

Experiments – Data synthesizers

In this chapter, we compare our data synthesizer framework, tabGAN, against the state-of-the-art models defined in Section 3.3. We give quite detailed explanations of how the experiments will be set up (with hyperparameters etc.), as this is an important step other comparable papers sometimes skips. This detailed setup will hopefully make it easy to reproduce the results if necessary. We will also briefly discuss why we choose to use these specific experiments. As we in this chapter perform many different experiments, we find it more structured to present results and discussion for each separate experiment together.

From the tabGAN framework we select 6 models to include in the comparison. Three of these are of the regular WGAN implementation, tabGAN, while the remaining three are of the conditional WGAN implementation, ctabGAN. Within each of these two groups, it is the preprocessing step that is different for the three methods; either standardization, quantile transformation or the Randomized Quantile Transformation. We name the six models tabGAN-sd, tabGAN-qt, tabGAN-qtr, ctabGAN-sd, ctabGAN-qt and ctabGAN-qtr, where we have used the naming regime [method]-[preprocessing step]. Here, method refers to either tabGAN or ctabGAN, while sd, qt and qtr are abbreviations for respectively standardization, quantile transformation and the Randomized Quantile Transformation.

The other data synthesizer methods that we will use for comparison are CTGAN, TVAE, GaussianCopula, CopulaGAN and TabFairGAN. For the CTGAN method we include two different configurations, one with packing and one without packing. We name these CTGAN-pac1 (without packing) and CTGAN-pac10 (with ten packed samples). We also include two configurations of TVAE: TVAE-orig and TVAE-mod, as described in Section 3.3.2. Throughout this chapter we will unless otherwise stated, train each model for 300 epochs with a batch size of 500.

7.1 Model Customization and Hyperparameter Choice

Throughout the experiments in this chapter, we will primarily use the data synthesizer configurations described in the following subsections. For the methods from the tabGAN framework this will mostly overlap with the default choices in the framework, but since the explanation of the default hyperparameter choice is a bit scattered across Section 3.2, we will give a compressed explanation here for clarity. The default choices in the tabGAN framework are based on an iterative, semimanual, and cautious hyperparameter search. We describe this process in Section 7.1.1 before we define hyperparameter choices for the tabGAN models in Section 7.1.2 and hyperparameter choices for the remaining data synthesizer models in Section 7.1.3.

7.1.1 Hyperparameter Search

The hyperparameter search was carried out as follows. We began with a hyperparameter combination we believed to be a reasonable choice based on gut feeling, hyperparameter choices in other papers such as Xu *et al.* [1], Xu and Veeramachaneni [13], Rajabi and Garibay [3], and Yang *et al.* [5], as well as trial-and-error on simple simulated datasets. Then, we began a more systematic approach in which we used the Adult dataset and a metric known as machine learning efficacy (see Chapter 2) in order to more formally assess the performance of the model for different combinations of hyperparameters. For each hyperparameter, we ran the model for a grid of values (with the other hyperparameters kept constant) and plotted the result. For some of the hyperparameters we believed to be dependent, we ran the model for a multidimensional grid and plotted the result for combinations of hyperparameters instead. As the results of such tests are not deterministic, we ran the model multiple times for each combination of hyperparameters and averaged the results. However, even with this procedure, we observed fluctuations due to randomness. If time was infinite, we could have run each grid point enough times until it stopped fluctuating. However, this was, of course, not the case, and we usually only ran each model between 10 and 25 times only. This was, however, enough to obtain quite stable results. Only after having run all of the hyperparameter grid searches of interest, we considered changing the default hyperparameters, and we only changed one or a few of the most promising hyperparameters. We found the promising hyperparameter choices by looking for trends in the plotted grid searches. We did not merely choose the best performing combination, but instead used a more catious approach, trying to avoid two types of overfitting. Firstly, we hoped to avoid wrong choices due the randomness of the model training, but more importantly, we wanted the hyperparameter choice to generalize well to other datasets. Thus, we only acted upon clear trends, and if in doubt, we either ran the model more times for each grid point or chose the conservative option.

We count the process just described as a single iteration of the iterative hyperparameter tuning used in this thesis. In total, we did eight of these iterations before landing on the hyperparameter choices listed as the default for the tabGAN framework presented here. This required thousands upon thousands of computing hours and was only made possible through the usage of the Idun computing cluster, which is a NTNU provided high-performing computational resource which also provide storage options. In fact, the EPIC research infrastructure, which is part of the Idun cluster, is one of Norway's largest general-purpose graphic processing unit (GPGPU) enabled infrastructure [81]. With the Idun cluster, we were able to run many tasks in parallel. This is not to say that a single script was run using parallel computing on a GPU, but instead that multiple scripts were run simultaneously, each with its own CPU and GPU. Although the Idun cluster allows multiple CPUs and GPUs to be used for a single job, we found it more efficient to instead submit more jobs simultaneously. We reason that this is due to the tabular data with our chosen batch sizes not being very straining for a single GPU, and therefore little is gained from adding more GPUs. The author also expresses gratitude for access to the Markov computational server belonging to the Department of Mathematical Sciences at NTNU, which provided faster hyperparameter tuning before access to the Idun cluster was granted.

7.1.2 Architecture and Hyperparameter Choice for Models from tabGAN Framework

For both the critic and the generator, we will use the Adam optimizer with initial learning rate $\alpha = 0.0002$ and exponential decay rates $\beta_1 = 0.7$ and $\beta_2 = 0.999$. We will use $n_{\text{critic}} = 10$ updates of the critic for each update of the generator and set the Wasserstein gradient penalty parameter equal to $\lambda = 10$. We will use Gumbel layers with softmax temperature $\tau = 0.1$ for tabGAN methods and $\tau = 0.5$ for ctabGAN methods. For the generator, we let the latent noise input layer be of size

$l_z = 128$. We give both the generator and the critic two dense hidden layers of size $l_{h,g} = l_{h,d} = 256$. After each hidden layer, we apply the GELU activation function. Additionally, for the generator, we use batch normalization after the activation function for each hidden layer.

For the quantile transformation in tabGAN-qt and tabGAN-qtr we use $n_{quantiles} = 1000$ quantiles to create the transformer for each column. In method tabGAN-qtr we only use the randomized version of the quantile transformation for values that contain more than 5% of the total observations. We set the spread parameter (see Section 4.3) of the Randomized Quantile Transformation equal to $\delta_{qtr,spread} = 1$, so that the randomization procedure uses all of the quantile space allocated for each of the values to which it is applied. As we believe it is disadvantageous to apply the Randomized Quantile Transformation to values which are only repeated a few times, we choose to only apply the randomized part of the Randomized Quantile Transformation to column values which are assigned more than 5% of the quantiles. For the one-hot encoding of the discrete variables, we include some uniform noise for the tabGAN methods by setting the parameter $\delta_{\text{oh}-\text{noise}}$, defined in Section 3.2.1, equal to 0.01. We do not include noise in the one-hot encoding for the ctabGAN methods.

7.1.3 Architecture and Hyperparameter Choice for Baseline Data Synthesizers

To make the comparison between the models fair, we set $n_{\text{critic}} = 10$ for each of the other GAN based methods, i.e CTGAN, CopulaGAN and TabFairGAN. Although CTGAN, CopulaGAN, and TabFairGAN by default have a lower value for n_{critic}, we can safely increase it since n_{critic} is not a tuning parameter. Setting the n_{critic} parameter to a higher value should only increase performance. Also by using a high number of critic updates we hope to gain more stable results for the methods. The only theoretical downside of increasing n_{critic} is increased training time.

By default, the CTGAN and CopulaGAN methods use two hidden layers of size 256 for both the generator and the discriminator, as well as a latent noise layer of size 128 for input to the generator. As this is exactly the same as for our methods from the tabGAN framework, we leave it as default. We do this for both CTGAN methods, CTGAN-pac1 (without packing) and CTGAN-pac10 (with packing). For TabFairGAN, the sizes of the hidden layers and the latent noise layer are automatically set to be equal to the dimensions of the transformed and one-hot-encoded data. As this is equal to only 50 for the Adult dataset, we created a modified version of the TabFairGAN script where the hidden layer and latent noise layer size can be set using input arguments. We ran the experiments which we will describe later in this chapter on modified versions of TabFairGAN, where we sat the generator and critic to have hidden layers of size 256, latent noise layer to be of size 128 and experimented with adding a second hidden generator layer. However, it appeared that the TabFairGAN architecture did not scale well and as we did not want to make drastic changes to the TabFairGAN architecture, we chose to merely use the default TabFairGAN architecture for the experiments in this thesis even though it uses smaller layer sizes. We feel this is a fair choice, since we let the TabFairGAN method use the best performing architecture we found from limited testing, while still not penalizing the other methods for the TabFairGAN method's lack of scalability. The TVAE method uses a latent layer size of 128 and hidden layer sizes of 128 by default. To ensure a fair comparison, we include, as described in Section 3.3.2, two versions of TVAE which we name TVAE-orig and TVAE-mod. The TVAE-orig method uses default layer sizes, while TVAE-mod sets the hidden layer sizes to be equal to 256 for both the encoder and decoder.

All other hyperparameters we leave as default, as we assume that they are chosen for a good reason or through a hyperparameter search. We note that CTGAN, CopulaGAN, and tabFairGAN use the Adam optimizer with the same learning rate as our chosen learning rate $\alpha = 0.0002$, although the parameter choice for β_1 and β_2 differs. CTGAN and CopulaGAN use $(\beta_1, \beta_2) = (0.5, 0.9)$, while TabFairGAN uses $(\beta_1, \beta_2) = (0.5, 0.999)$. The usage of the same optimizer, Adam, with

similar parameter choices contributes to making the comparison of preprocessing, model architecture, and training process choices fair. Also, the two TVAE implementations described in Section 3.3.2, TVAE-orig and TVAE-mod, use Adam optimizer. TVAE uses a learning rate of 0.001 and $(\beta_1, \beta_2) = (0.9, 0.999)$.

7.2 Visual Evaluation of tabGAN Methods on the Syn2D_3cats Dataset

As a first experiment we want to visually confirm if and how our data synthesizing framework is capable of learning a data distribution. To do this we will use the Syn2D_3cats dataset as it is easy to visualize in two dimensions. We only include the tabGAN-sd and ctabGAN-sd methods in this experiment in order to not overwhelm the reader with plots. Additionally, since Syn2D_3cats is a simple dataset, quantile transformation might not be needed, or perhaps even beneficial to use, as the preprocessing method for the two continuous columns. We will train tabGAN on the Syn2D_3cats dataset using 5000 epochs and a batch size of 500. As the training dataset only has 2500 observations, this equals to 5 batches per epoch. For the 20 chosen epochs evenly distributed between and including 0 and 5000, we plot a generated data set of the same size as the Syn2D_3cats training dataset for tabGAN-sd and ctabGAN-sd. In this way, we can visually inspect how the generator improves as the number of epochs trained increases. In addition to this, we will plot an estimate for the EM distance averaged over each epoch for each of the two methods. The approximation for the EM distance is calculated using Equation (2.7).

In Figure 7.1 and Figure 7.2, we plot the training progress on the Syn2D_3cats dataset for, respectively, tabGAN-sd and ctabGAN-sd. The result is pretty good, but not perfect. From the figures, it is clear that both methods definitely learn most of the defining features of the dataset. For instance, they both learn to recreate category $cat1$ almost exactly, with the shape of its mean being a second order polynomial and the variance increasing steadily from $x_1 = 0$ to $x_1 = 10$. The tabGAN-sd method, however, appears to be somewhat superior in its approximation, since it captures that the variance is close to zero when close to $x_1 = 0$. Moreover, both learn the location of category $cat3$ and its uniform distribution inside a triangle. Unsurprisingly, both struggle to recreate exactly the shape and sharp lines of the triangle. The method ctabGAN-sd appears to be a bit more successful in its recreation of the triangle shape and sharp lines; however, they both create a decent, although a bit distorted approximation. Both methods also appear to partially capture the conditional distribution given $cat2$. They have both learned that it has the shape of a third order polynomial. The tabGAN-sd method appears to better recreate the polynomial shape, but does not adequately capture the variance increase from $x_1 = 0$ to $x_1 = 10$. Meanwhile, the ctabGAN-sd method recreates the shape slightly more distorted, especially between $x_1 = 0$ and $x_1 = 2$, but appears to better capture that the variance is lower from $x_1 = 2$ to $x_1 = 6$ than from $x_1 = 6$ to $x_1 = 10$. However, it does not manage to recreate the gradual increase in variance. Additionally, the ctabGAN-sd method has much more noise inside the parabola part of the $cat2$ distribution located between $x_1 = 2$ and $x_1 = 10$ than the tabGAN-sd method, and the tabGAN-sd method better generates samples between $x_1 = 8$ and $x_1 = 10$ for category $cat2$. If we had run the training of the methods for more epochs, then the recreated distribution might have been better and a bit less noisy for both methods, but given the training progress seen in Figure 7.1 and Figure 7.2 and the plots of the approximated Earth-Mover distance in Figure 7.3 and Figure 7.4, we reason that they might be close to convergence.

At epoch 0, before any training has taken place, the generated samples are merely an unsystematic mix of different categories for both methods. At epoch 526 we observe that there has been some separation between the categories and the conditional distributions given $cat1$ or $cat3$ resemble the true conditional distributions. We observe that ctabGAN-sd appears to learn quicker than the tabGAN-sd method. Already at epoch 1578 the data distribution is pretty much recovered,

although a bit distorted version of it, while tabGAN-sd has still not learned the conditional distribution given $cat2$. After epoch 1578, we actually do not observe much progress made by the ctabGAN-sd method. The fact that the ctabGAN-sd method learns the distribution so quickly might be a possible reason why the plot of the approximate Earth-Mover distance for ctabGAN in Figure 7.4 appears a bit strange. The first peculiarity can be explained by the break-in period for the critic where it has to learn to separate the real and fake distribution (or more accurately measure the distance between real and fake distribution). In both Figure 7.3 and Figure 7.4, we observe that the approximate Earth-Mover distance increases quickly for the first couple of epochs. We reason that this increase is due to the critic learning to actually approximate the correct distance. Then, for both tabGAN-sd and ctabGAN-sd, there is a sharp decrease in the estimated EM distance. We reason that this is due to the generator having a steep learning curve at the beginning when it goes from producing noise to something that actually resembles samples from the real distribution. For the tabGAN-sd method, the rapid decrease slows down before it reaches zero and we observe a slow, although noisy, convergence to zero from the positive numbers side. However, for the ctabGAN-sd method the Earth-Mover distance approximation actually becomes negative, before we observe a quick convergence back to zero from the negative numbers side. We reason that this might be due to the generator learning more quickly than the critic is able to keep up a good approximation of the Earth-Mover distance. Consequently, the critic might be the potentially limiting factor for the learning of the ctabGAN-sd method for this specific dataset and this specific run. We note that for tabGAN-sd the Earth-Mover distance approximation follows reasonably well the observed quality for different epochs in Figure 7.1. This highlights one of the potential benefits of using Wasserstein GAN, where the loss function actually can be a useful representation of the training progress. However, as we see in Figure 7.4, with the negative Earth-Mover approximation and convergence to zero after around 250 epochs, the approximation can not be trusted blindly. Perhaps with an ever higher value for n_{critic} we might have gotten a more accurate Earth-Mover distance approximation. The large fluctuations we observe for both methods in the estimate of the Earth-Mover distance is likely due to the small number of batches in each epoch which we average over.

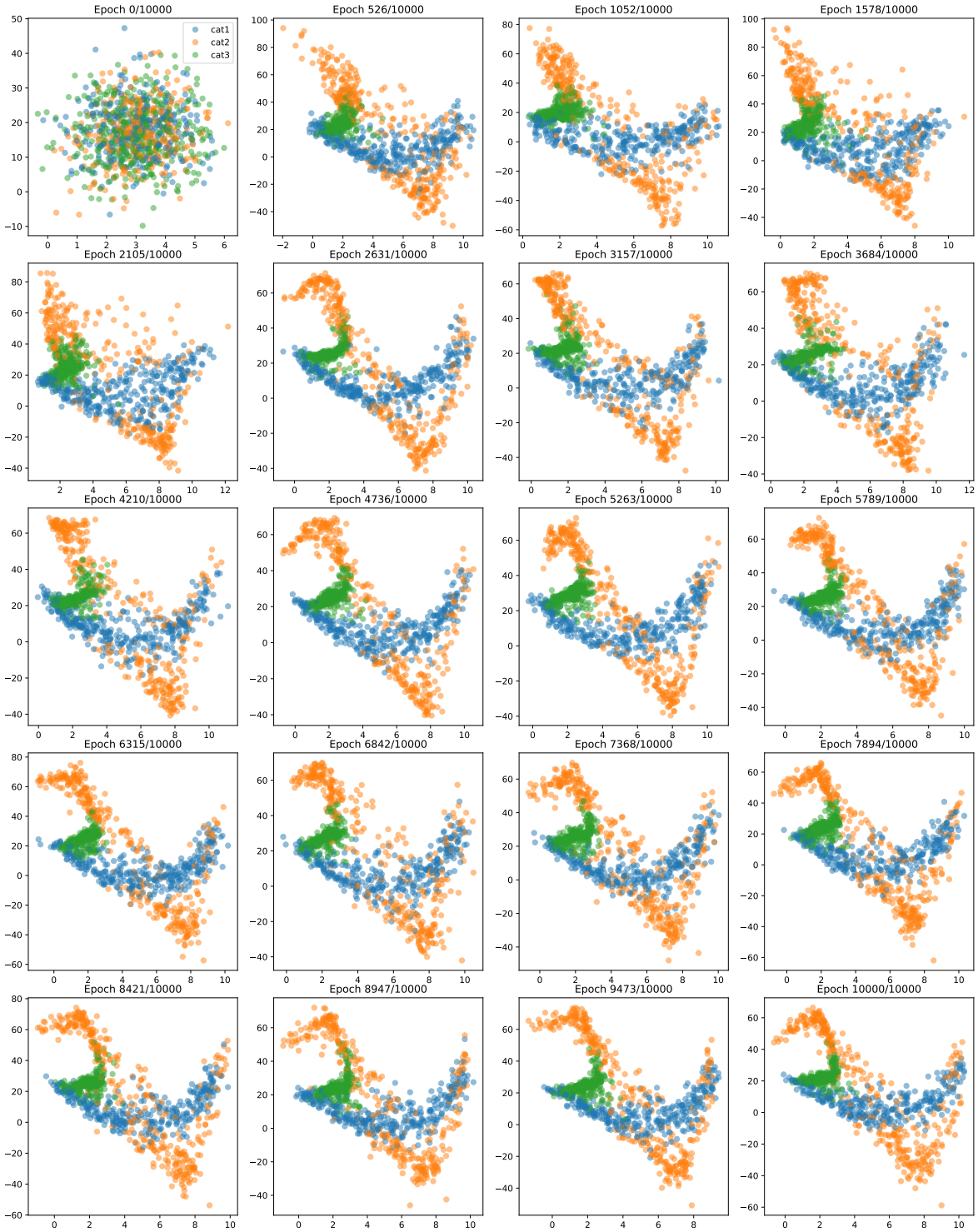


Figure 7.1: Visualization of the training progress for **tabGAN-sd** when learning to generate samples from the Syn2D_3cats dataset defined in Section 6.1.

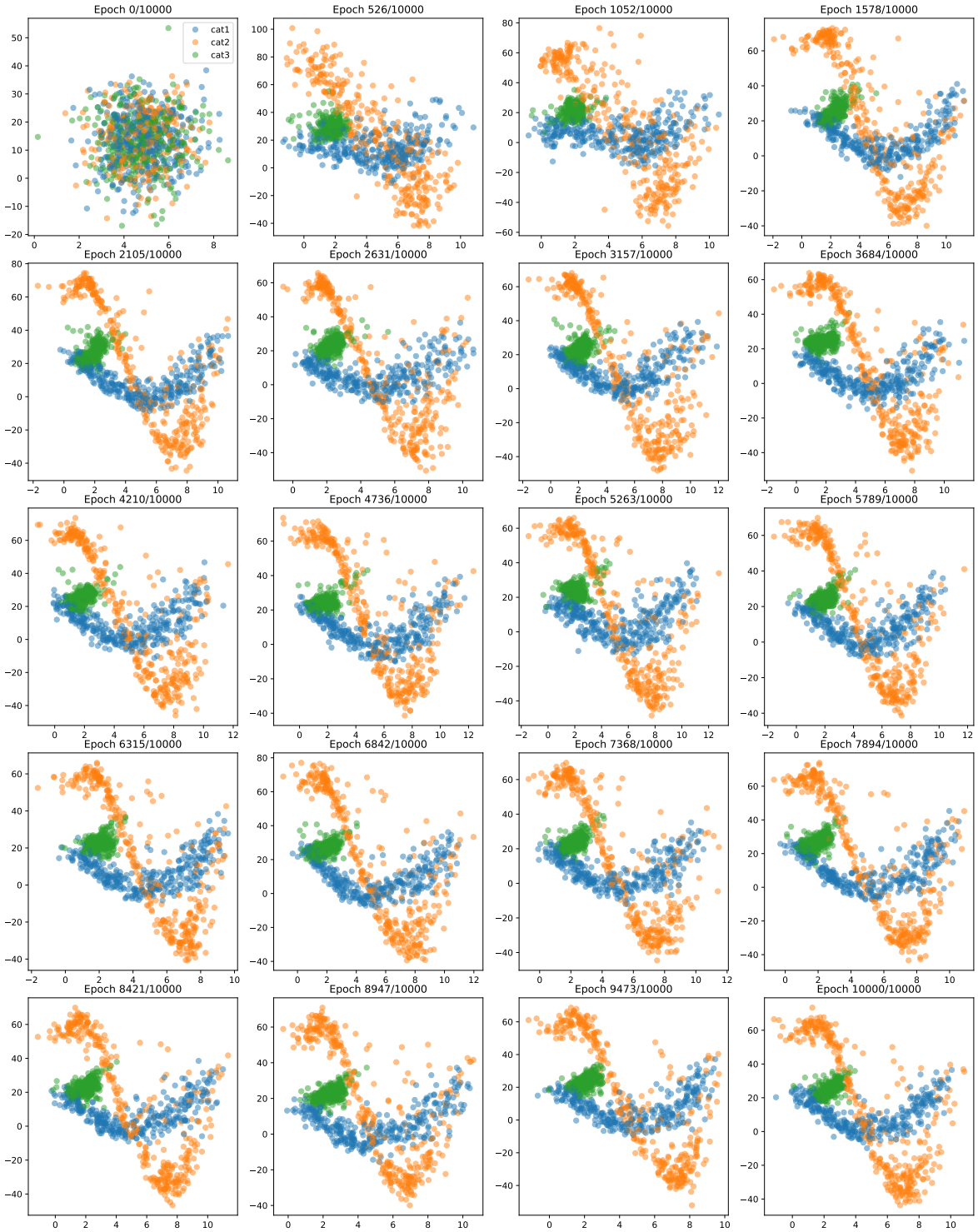


Figure 7.2: Visualization of the training progress for **ctabGAN-sd** when learning to generate samples from the Syn2D_3cats dataset defined in Section 6.1.

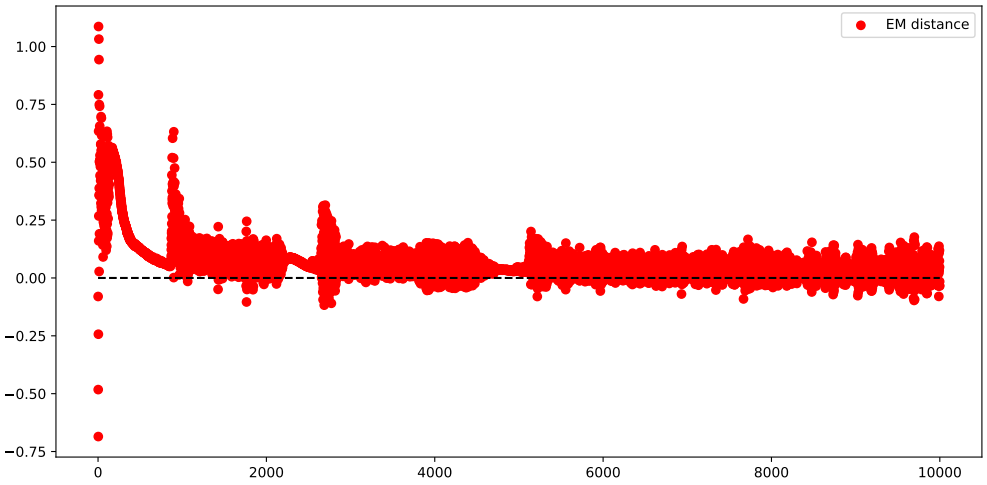


Figure 7.3: Estimate of the Earth-Mover distance during training for the **tabGAN-sd** method when learning to generate samples from the Syn2D_3cats dataset defined in Section 6.1. The Earth-Mover distance approximation is averaged over all batches in each epoch.

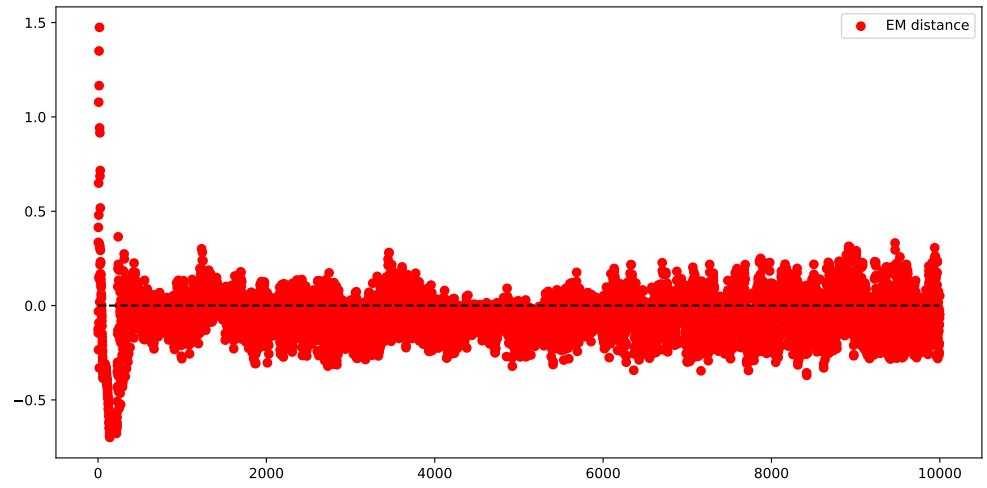


Figure 7.4: Estimate of the Earth-Mover distance during training for the **ctabGAN-sd** method when learning to generate samples from the Syn2D_3cats dataset defined in Section 6.1. The Earth-Mover distance approximation is averaged over all batches in each epoch.

7.3 Comparison of Marginal Distributions on Adult Dataset

To evaluate how well the methods capture the marginal probability distribution of each column in the Adult dataset, we will plot for each column the histogram for a synthesized training set together with the histogram of the observations from the true training set. If a data synthesizer method captures the correct marginal distributions, then the histograms should overlap. We will do this for each method. For the plots to be easily comparable between methods, we use a fixed bin size and bin placement for each column. For a continuous column this is done in the following way. First, we plot 20 bins between the minimum value of the original dataset and the maximum value of the original dataset. This is used to determine the bin size and bin placement. To accommodate cases where the synthesizers generate values outside of the minimum and maximum of the original dataset, we extrapolate the bins if needed. For discrete columns the categories are listed alphabetically. To avoid overflowing the main part of the thesis with very similar plots, we will move the plots for ctabGAN-sd, ctgabGAN-qt, CTGAN-pac10, CopulaGAN and TVAE-mod to Appendix C. The reason we chose these plots, is that they are very similar to the plots for some of the other methods discussed in this section.

7.3.1 Results and Discussion of Marginal Distributions

In Figures 7.5 to 7.7, we plot the histograms for tabGAN-sd, tabGAN-qt, and tabGAN-qtr, respectively. We observe that all three methods do a very satisfactory job of recreating the marginal distributions. For tabGAN-qt and tabGAN-qtr, the marginal distributions are almost perfectly recreated for every column, with the exception of a single category *Doctorate* in the column "education", which for some reason is the only one not sampled. It is not very surprising, as it is the category with the least representation amongst the categories in the "education" column, but tabGAN-qt and tabGAN-qtr still manage to correctly sample categories with even less representation from other columns, such as the *Priv-house-serv* category from the "occupation" column and *Native-American-Inuit* and *Other* from the "race" column. If we were to make an educated guess, we would believe a possible explanation for why *Doctorate* is more difficult than the other categories just listed, could be the strong correlation between "education" and some of the other columns. This is in contrast to the "race" column, which hardly has any correlation with any of the other columns (see Figure 7.13 in Section 7.4). By definition, there is a strong correlation between "education" and "education_num", but as later seen in Figure 7.13 in Section 7.4, the column "education" is strongly correlated to other columns as well.

We observe that tabGAN-qtr does a slightly better job of correctly sampling the marginal distributions for the continuous columns with many repeated values. This is the case for "educational_num", "capital_gain", "capital_loss", and "hours_per_week". This indicates that the Randomized Quantile Transformation is working as intended. It makes it easier for the generator to not oversample the values which are repeated many times. This is, for instance, the case for the value 40 for the "hours_per_week" column and the values 9, 10, and 13 for the column "educational_num". For the columns "capital_gain" and "capital_loss", almost all the observations have value 0. Consequently, the distribution of the few samples which are not are equal to zero, are easy for the generator to "forget" to sample from. We observe that tabGAN-qt does not recreate enough samples with "capital_gain" $>$ 15000 and for the small cluster of values near 2000 for the "capital_loss" column it undersamples the values smaller than 2000. tabGAN-qtr, on the other hand, correctly generates such samples. The only cluster from which tabGAN-qtr fails to sample is the few observations that work around 80 hours per week.

As expected, the tabGAN-sd method struggles with the opposite problem as tabGAN-qt. The tabGAN-sd method has a hard time not slightly undersampling the values that are repeated many

times. This is due to the standardization transformation mapping all observations to an infinitesimal range, unlike the quantile transformation, which maps unrepeated values to an infinitesimal range and repeated values to a finite range proportional to the number of repeated values. This is clearly seen for the columns "capital_gain" and "capital_loss", where we observe that a significant share of the samples are given values slightly below zero instead of zero. Note, however, that the bin placement makes it extra obvious and may perhaps be a bit misleading. The values are probably only mapped to just below zero; however, as there is a bin separation at zero, they get placed in their own bin. Just like tabGAN-qt, we observe that tabGAN-sd also struggles with the recreation part of the distribution for both "capital_gain" and "capital_loss" which is much larger than zero. For the integer columns "educational_num" and "hours_per_week" we also observe that tabGAN-sd struggles a bit with the distributions that are either not continuous or very far from normally distributed. Again, we observe that it undersamples the values which are repeated many times. Additionally, it of course is unable to generate only integers, and therefore also generates decimal numbers lying between each integer. This could, however, have been easily fixed with a post-processing step. In fact, this is actually implemented for the methods in the Stochastic Data Vault package (with CTGAN, CopulaGAN, GaussianCopula, and TVAE), where it automatically detects the appropriate decimals to round. Although we have been a bit critical of the tabGAN-sd method, overall it makes a pretty good job of recreating the marginal distributions. It actually perfectly recreates all of the marginal distributions for the discrete columns, even the "Doctorate" category which the tabGAN-qt and tabGAN-qtr methods were unable to correctly sample.

In Figure C.1, Figure C.2, and Figure 7.8, we plot the histograms for ctabGAN-sd, ctabGAN-qt and ctabGAN-qtr, respectively. The marginal distributions for ctabGAN-sd are practically the same as for tabGAN-sd. Not surprising, since the goal of the conditional GAN architecture and conditional sampling is to better represent the rare categories. However, as tabGAN-sd already perfectly recreated the marginal distributions of the discrete columns, no gain was possible. Adding the conditional architecture and training process did not address the problems for tabGAN-sd that we described above. The plot for ctabGAN-qt is also very similar to the plot for tabGAN-qt. The only notable differences are that now the number of samples with the "Doctorate" category is noticeable in the histogram, although it is still very much undersampled, and now it completely fails to generate the tiny cluster of samples with "capital_loss" value around 2000. This indicates that the conditional architecture and training process helps a bit with representing and learning to generate rare categories. The plot for the ctabGAN-qtr method is almost equal to the plot for the tabGAN-qtr method, with one important exception, now the "Doctorate" category is correctly sampled as well. Consequently, we find no error with the marginal distributions produced by the ctabGAN-qtr method.

In Figure 7.9, we plot the histogram for the GaussianCopula method. It is obvious that this method struggles with all of the discrete columns and the numeric columns which do not resemble any of the parametric distributions available to it. The "age" column and the "fnlwgt" columns are pretty well reconstructed. However, the rest is very badly approximated. The "hours_per_week" column appears to have had a normal distribution with mean around 40 fitted to it, but the normal distribution completely fails to replicate the large number of observations with the value 40. The same problem can be seen in the "educational_num" column. The "capital_gain" column and the "capital_loss" column appear to have had an exponential distribution or perhaps a gamma distribution fitted to it. By doing this, the GaussianCopula is partially able to capture the large number of samples near 0, but it still undersamples. Additionally, we assume that not all samples collected in the bin near 0 are actually that close to zero, as the bin is quite wide. As a consequence of the massive undersampling of values at zero, the GaussianCopula method massively oversamples the larger than zero values for both "capital_gain" and "capital_loss". As expected, this method is

also not able to capture the tiny cluster near 2000 for the column "capital_loss". For the discrete columns, a few categories of each class are massively oversampled, whereas many categories are not sampled at all. Again, this is to be expected since the GaussianCopula method's solution to discrete variables is merely one-hot encoding and then treating each new one-hot encoded column as a regular continuous variable. Clearly, this is not a very successful strategy for real datasets with discrete variables.

The histograms for the CTGAN-pac1, CTGAN-pac10 and CopulaGAN methods are given in Figure 7.10, Figure C.3, and Figure C.4, respectively. The marginal histograms for CTGAN-pac1 and CTGAN-pac10 are almost indistinguishable, although CTGAN-pac1 appears to have a slight advantage overall. All of the columns are recreated satisfactory. However, we observe that compared to the methods from the tabGAN framework, category shares are not accurately recreated. This is an overall trend for all the discrete columns, but especially for "occupation", "race" and "native_country". Additionally, we observe that both CTGAN-pac1 and CTGAN-pac10 fail to correctly recreate the tiny cluster around 2000 for the column "capital_loss". We observe that they both oversample at the middle of the cluster and undersample at the edges of the tiny cluster. It is reasonable to assume that the preprocessing step of CTGAN, which fits a variational Gaussian Mixture Model, assumes there is a Gaussian distribution centered in the middle of the cluster and this is the result of what the generator was able to pick up on. The marginal histograms for the CopulaGAN method appear quite similar to those for CTGAN-pac1 and CTGAN-pac10. The most notable difference is that the CopulaGAN method generally is more successful with recreating the correct category shares, however, unlike the CTGAN methods, the CopulaGAN method undersamples the *Doctorate* category.

In Figure 7.11, we present the marginal distribution histograms for the TabFairGAN method. It does a very successful job of recreating the marginal distributions of the discrete columns, definitely on par with the ctabGAN-qtr method with respect to that. The marginal distribution of the numerical columns are also pretty well recreated, but not as successfully as ctabGAN-qtr, CTGAN-pac1, CTGAN-pac10 and CopulaGAN. Unsurprisingly, TabFairGAN shares some of the issues as tabGAN-qt and ctabGAN-qt, as they all rely on regular quantile transformation. All these methods have a tendency to oversample the column values that are repeated many times. We observe this for the columns "hours_per_week, "educational_num", "capital_loss" and "capital_gain". Additionally, the recreation of the "age" column is surprisingly poor. For the "age" column it is especially obvious that TabFairGAN oversamples the edge values, but the same is apparent in all of the other numerical columns as well. We believe it reasonable to assume that this is due to TabFairGAN using uniform quantile transformation and no activation function to help guiding the numerical output nodes from the space $(-\infty, \infty)$ to the space $[0, 1]$. Thus, the neural network that constitutes the generator is forced to learn the mapping to the $[0, 1]$ space, but clearly fails to not generate too many samples larger than 1 or smaller than 0. The tabGAN framework avoids this issue by using quantile transformation with Gaussian output distribution instead of uniform output distribution. However, a potential solution for TabFairGAN might be to utilize a sigmoid activation function after the numerical output layer.

In Figure 7.12 and Figure C.5, we plot the marginal distributions of TVAE-orig and TVAE-mod, respectively. Both of the two variational autoencoder methods perform quite well, but there are a couple of issues. First, the edge cases for "age" and "fnlwgt" are undersampled. Additionally, the category shares are a bit off, especially for TVAE-mod and the "occupation" column. The *Separated* category of the "marital_status" column is massively undersampled for both methods. On a positive note, the marginal distributions of the remaining numerical columns are very well reconstructed, and they even partially managed to create samples for the tiny cluster around value 2000 for the "capital_loss" column.

To sum up, all methods except the GaussianCopula method managed to decently recreate the

marginal distributions. The best performing method in this comparison we would say is ctabGAN-qtr, but with CTGAN-pac1, CTGAN-pac10, CopulaGAN, tabGAN-qt, tabGAN-qtr and ctabGAN-qt close behind. This comparison is based on a single run and naturally there will be some randomness involved, however, we observed much of the same patterns when rerunning multiple times.

Figure 7.5: Histograms of generated samples from **tabGAN-sd** plotted against histograms of observations from the Adult training dataset. One histogram for each column. The synthesized dataset is of the same size as the Adult training dataset.

Figure 7.6: Histograms of generated samples from **tabGAN-qt** plotted against histograms of observations from the Adult training dataset. One histogram for each column. The synthesized dataset is of the same size as the Adult training dataset.

Figure 7.7: Histograms of generated samples from **tabGAN-qtr** plotted against histograms of observations from the Adult training dataset. One histogram for each column. The synthesized dataset is of the same size as the Adult training dataset.

Figure 7.8: Histograms of generated samples from **ctabGAN-qtr** plotted against histograms of observations from the Adult training dataset. One histogram for each column. The synthesized dataset is of the same size as the Adult training dataset.

Figure 7.9: Histograms of generated samples from **GaussianCopula** plotted against histograms of observations from the Adult training dataset. One histogram for each column. The synthesized dataset is of the same size as the Adult training dataset.

Figure 7.10: Histograms of generated samples from **CTGAN-pac1** plotted against histograms of observations from the Adult training dataset. One histogram for each column. The synthesized dataset is of the same size as the Adult training dataset.

Figure 7.11: Histograms of generated samples from **TabFairGAN** plotted against histograms of observations from the Adult training dataset. One histogram for each column. The synthesized dataset is of the same size as the Adult training dataset.

Figure 7.12: Histograms of generated samples from **TVAE-orig** plotted against histograms of observations from the Adult training dataset. One histogram for each column. The synthesized dataset is of the same size as the Adult training dataset.

7.4 Comparing Joint Distributions on Adult Dataset

After having evaluated the capabilities of the data synthesizer methods to replicate marginal distributions, we will proceed to determine how well they capture the correlation structures in the training dataset. For each pair of column combinations, we want to determine whether the joint distribution appears to be replicated. This can be done by plotting 3D histograms or correlation matrices for each unique column combination of length two. However, this would for the Adult dataset result in $\sum_{i=1}^{14} i - 1 = 91$ plots or matrices, which would be overwhelming and not very useful for comparison, especially when taking into consideration that these 91 plots would have to be plotted for each of the thirteen methods of interest. We could of course only plot a selected subsection of these plots or matrices, but then we would lose information, and at the same time the results would be difficult to interpret and compare. Thus, inspired by Xu and Veeramachaneni [13], we choose to use mutual information as a quantitative evaluation tool for correlation between columns. More specifically, we compute the pairwise mutual information between columns and for ease of comparison and visualization we use a normalized version. Mutual information is a quantitative measurement of how dependent two variables are. More precisely, what mutual information criteria do is quantify how much information is gained about one random variable by observing another random variable. For ease of computation, we discretize each numerical column into 40 buckets with ideally 2.5% of the observations in each (unfortunately, as some of the numerical columns have repeated values that represent more than 2.5% of the observations, some of the numerical columns will have fewer than 40 columns and some buckets will contain more than 2.5% of the observations). In this thesis, we choose to use the following version of normalized mutual information

$$NMI(\mathcal{X},\mathcal{Y}) = \frac{2}{\text{Entropy}(\mathcal{X}) + \text{Entropy}(\mathcal{Y})} \sum_{x\in\mathcal{X}} \sum_{y\in\mathcal{Y}} P_{\mathcal{X},\mathcal{Y}}(x,y) \log\left(\frac{P_{\mathcal{X},\mathcal{Y}}(x,y)}{P_{\mathcal{X}}(x)P_{\mathcal{Y}}(y)}\right), \tag{7.1}$$

where x and y are the categories of the discretized columns $\mathcal{X}$ and $\mathcal{Y}$. The probabilities of $P_{\mathcal{X}}$, $P_{\mathcal{Y}}$ and $P_{\mathcal{X},\mathcal{Y}}(x,y)$ are respectively the (estimated) marginal distribution of column $\mathcal{X}$, the (estimated) marginal distribution of column $\mathcal{Y}$ and the (estimated) joint distribution of columns $\mathcal{X}$ and $\mathcal{Y}$. The entropy of $\mathcal{X}$ and $\mathcal{Y}$ are calculated as

$$\text{Entropy}(\mathcal{X}) = \sum_{x\in\mathcal{X}} P_{\mathcal{X}}(x) \log\left(\frac{1}{P_{\mathcal{X}}(x)}\right)$$
$$\text{Entropy}(\mathcal{Y}) = \sum_{y\in\mathcal{Y}} P_{\mathcal{Y}}(y) \log\left(\frac{1}{P_{\mathcal{Y}}(y)}\right).$$

Note that our definition of normalized mutual information is symmetric, such that $NMI(\mathcal{X},\mathcal{Y}) = NMI(\mathcal{Y},\mathcal{X})$. Direct insertion into Equation (7.1) also shows that $NMI(\mathcal{X},\mathcal{X}) = 1$. If there really is no correlation between two columns and we use the true probability densities for the estimation, then the NMI value would be equal to zero. In Figure 7.13, we plot the NMI matrix for the training observations of the Adult dataset. As expected, we observe that there is a very strong correlation between the "education" and the "educational_num" column. Additionally, we see a bunch of weaker correlations such as between "age" and "marital_status", between "income" and "marital_status", between "occupation" and "gender". These and more are correlation structures that we want the data synthesizers to capture.

We will for each method plot the NMI matrix of a synthesized dataset the same size as the Adult training dataset. As a reference, we will once again plot the NMI matrix of the Adult training

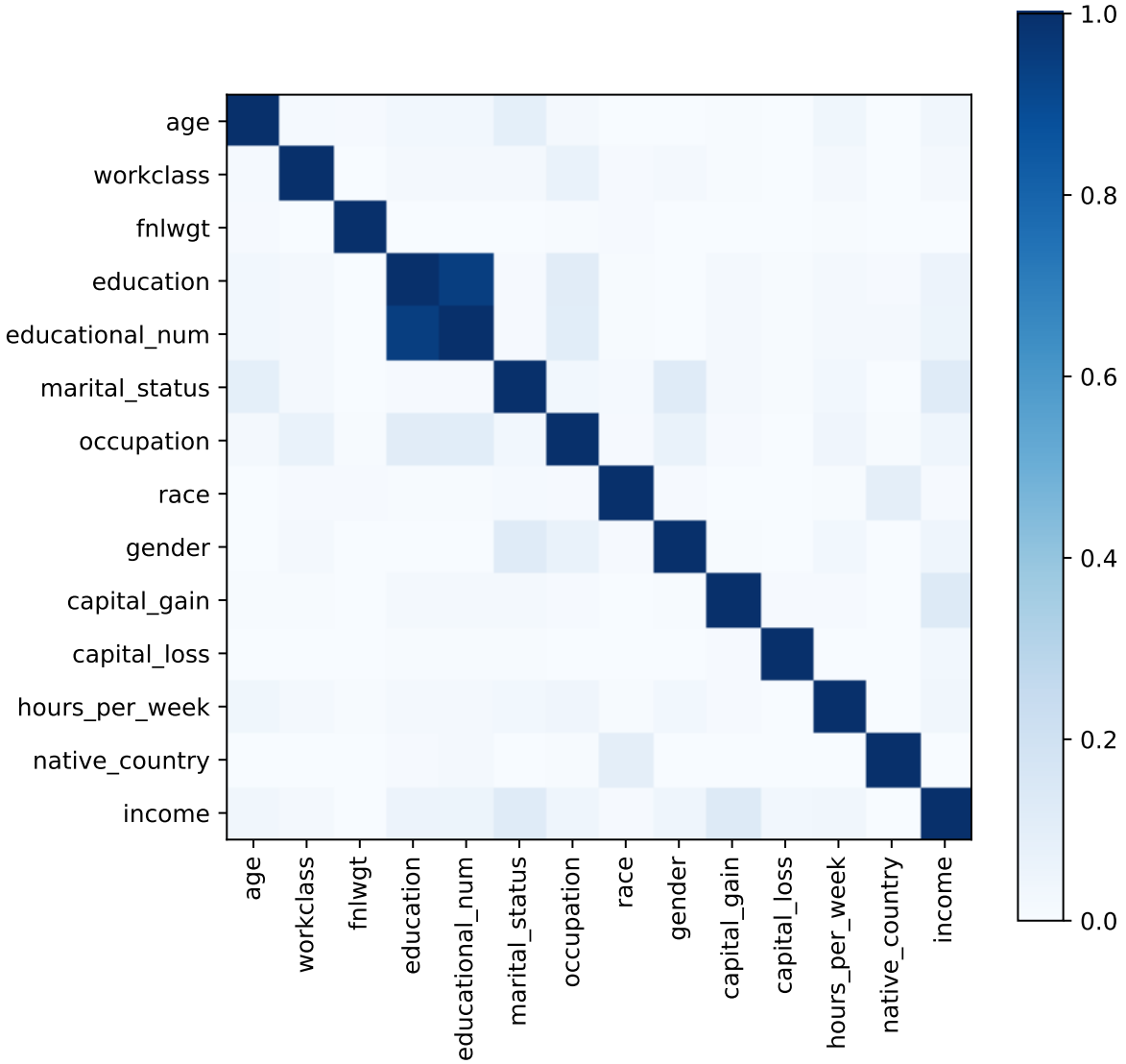


Figure 7.13: Normalized mutual information (NMI) matrix for the training observations of the Adult dataset introduced in Section 6.2.

dataset. For ease of comparison, we will also plot the difference between the NMI matrix of the original Adult training dataset and synthesized test datasets (of same size as the true test Adult dataset). As reference we plot the NMI matrix of the original Adult training dataset subtracted from the NMI matrix of the original Adult test dataset, thus we gain an example of the difference in NMI matrix that can be due to randomness.

7.4.1 Results and Discussion of Joint Distributions

In Figure 7.14, we plot the NMI matrices for the methods from the tabGAN framework. It is clear that all of these methods have successfully managed to capture the joint distribution between each column pair. The NMI matrices are almost indistinguishable from the NMI matrix of the original Adult training dataset. The most obvious deviation is the NMI value between the "education" and "education_num" columns. All of the methods from the tabGAN framework successfully capture that

there is a high correlation between the two education columns, but they fail to capture just how strong the correlation is. We observe that the methods with quantile transformation does a slightly better job of capturing the high correlation than the methods with standardization as preprocessing method. The method that manages to estimate the highest correlation is tabGAN-qt. Other than the education columns, the methods with quantile transformation appear to be extremely successful in recreating the joint distributions. tabGAN-sd and ctabGAN-sd are also very successful, but we notice that they introduce a correlation between "capital_loss" and the education columns that does not exist in the original Adult training dataset. tabGAN-sd introduces extra correlation that should not be there to a larger extent than ctabGAN-sd. This is easier to spot in Figure 7.16, where we plot the difference between the original Adult training dataset and the synthetically generated datasets. From Figure 7.17 we observe that there are no difference in the NMI matrix calculated for the original Adult training dataset and the NMI matrix calculated for the original Adult test dataset. Figure 7.16 also reveals that tabGAN-sd and ctabGAN-sd fail to capture some of the correlation between the columns "capital_gain" and "income". This is also slightly the case for tabGAN-qtr and ctabGAN-qtr.

We plot the NMI matrices for GaussianCopula, CTGAN-pac1, CTGAN-pac10, CopulaGAN, TabFairGAN, TVAE-orig and TVAE-mod in Figure 7.15. At first sight, it is obvious that the GaussianCopula method has competely failed to learn the joint distributions. Except for the diagonal, which by definition is equal to 1, and the single tile between "age" and "marital_status", the NMI matrix is blank. Consequently, the GaussianCopula has failed to learn any of the joint distributions. The other synthesizers, however, are much more successful. We observe that CTGAN-pac1, CTGAN-pac10, TVAE-orig and TVAE-mod are close to capturing the true strength of the correlation between the education columns, with TVAE-mod being the most successful and really close to the actual value. CopulaGAN and TabFairGAN also capture the correlation, but are less successful in capturing its strength. We observe that CTGAN-pac1 and CTGAN-pac10 create more correlation than there should be between the columns "race" and "native_country", while CopulaGAN does not create enough correlation. From Figure 7.15, it is clear that the TVAE methods introduce many correlation structures that are not present in the original Adult training dataset. This is even more of a problem for TVAE-orig than TVAE-mod, but the issue is very much present for both methods. Looking at Figure 7.17, where we plot the the difference between the synthesized datasets and the original Adult training dataset, the trend is even more obvious. The plot of the differences in each NMI matrix shows that the TabFairGAN and CopulaGAN methods do not create any extra correlation patterns not present in the original dataset; however, they do slightly underestimate the correlation between "income" and "capital_gain" and as mentioned earlier, they are less successful in capturing the true strength of the correlation between the education columns. CopulaGAN slightly better approximates the correlation between "education" and "education_num", but in return slightly underestimates the correlation patterns between "native_country" and "race" as well as the correlation between "gender" and the two columns "marital_status" and "occupation". CTGAN-pac1, CTGAN-pac10, TVAE-orig and TVAE-mod also underestimate the correlation between "capital_gain" and "income". It is obviously a difficult correlation to capture correctly, since all the methods except for tabGAN-qt and ctabGAN-qt fail to capture the correlation successfully. We believe the difficulty may potentially be caused by the extreme overweight of repeated 0 values for "capital_gain", which makes it difficult to correctly estimate the cases with "capital_gain" larger than zero.

Overall, the GAN synthesizers are the best performing in joint distribution comparison. To be fair, the TVAE-mod method was clearly the best at capturing the strong correlation between the education columns, but for this dataset, the variational autoencoder synthesizers were too prone to introduce correlation structures not present in the original dataset. We believe the tabGAN-qt method overall was the best performing method for this comparison, with ctabGAN-qt, tabGAN-qtr,

ctabGAN-qtr, CTGAN-pac1, CTGAN-pac10, CopulaGAN and TabFairGAN not far behind.

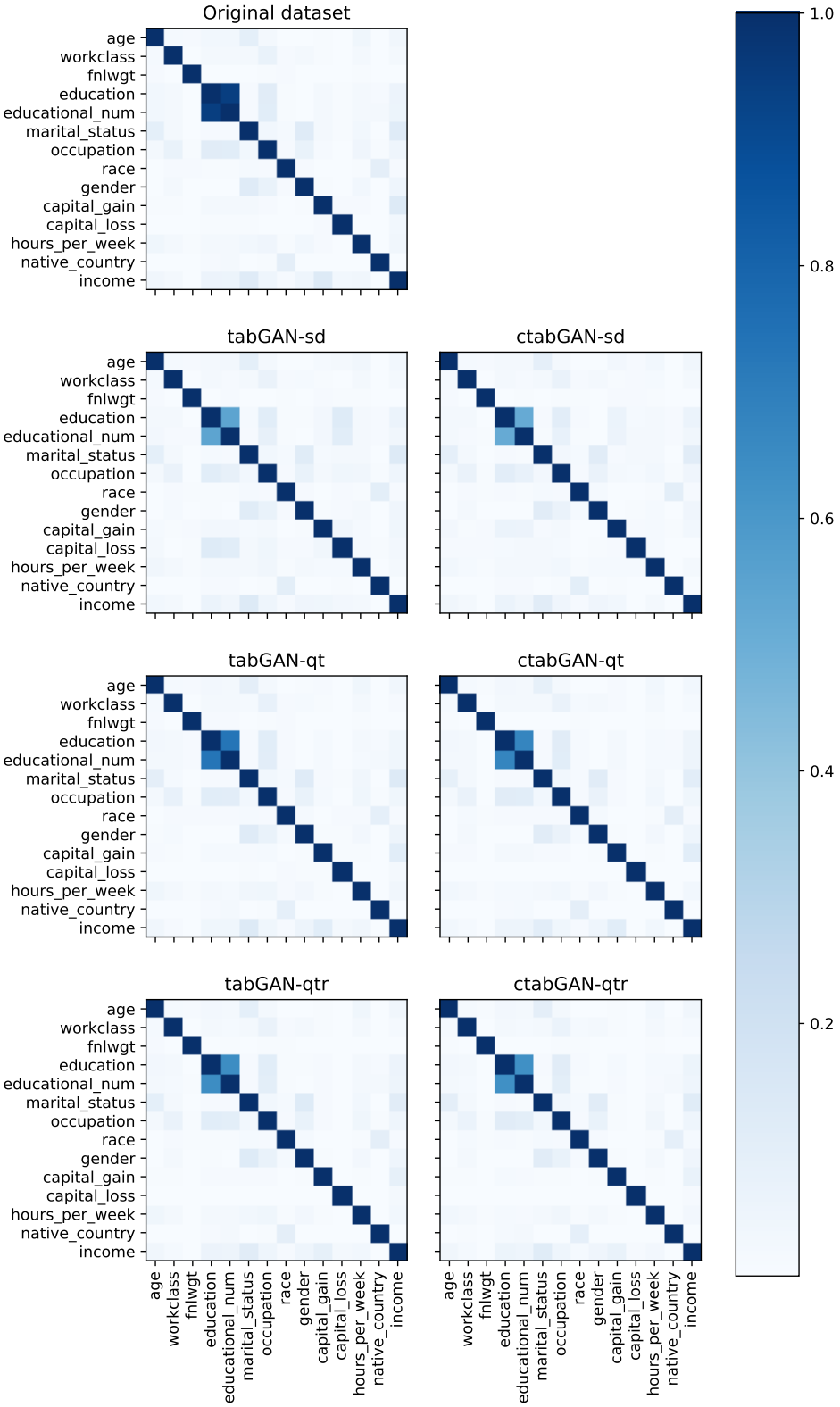


Figure 7.14: NMI matrices for the Adult training dataset and the synthesized datasets generated by each of the methods from the tabGAN framework: tabGAN-sd, tabGAN-qt, tabGAN-qtr, ctabGAN-sd, ctabGAN-qt and ctabGAN-qtr. The synthesized datasets were of the same size as the Adult training dataset.

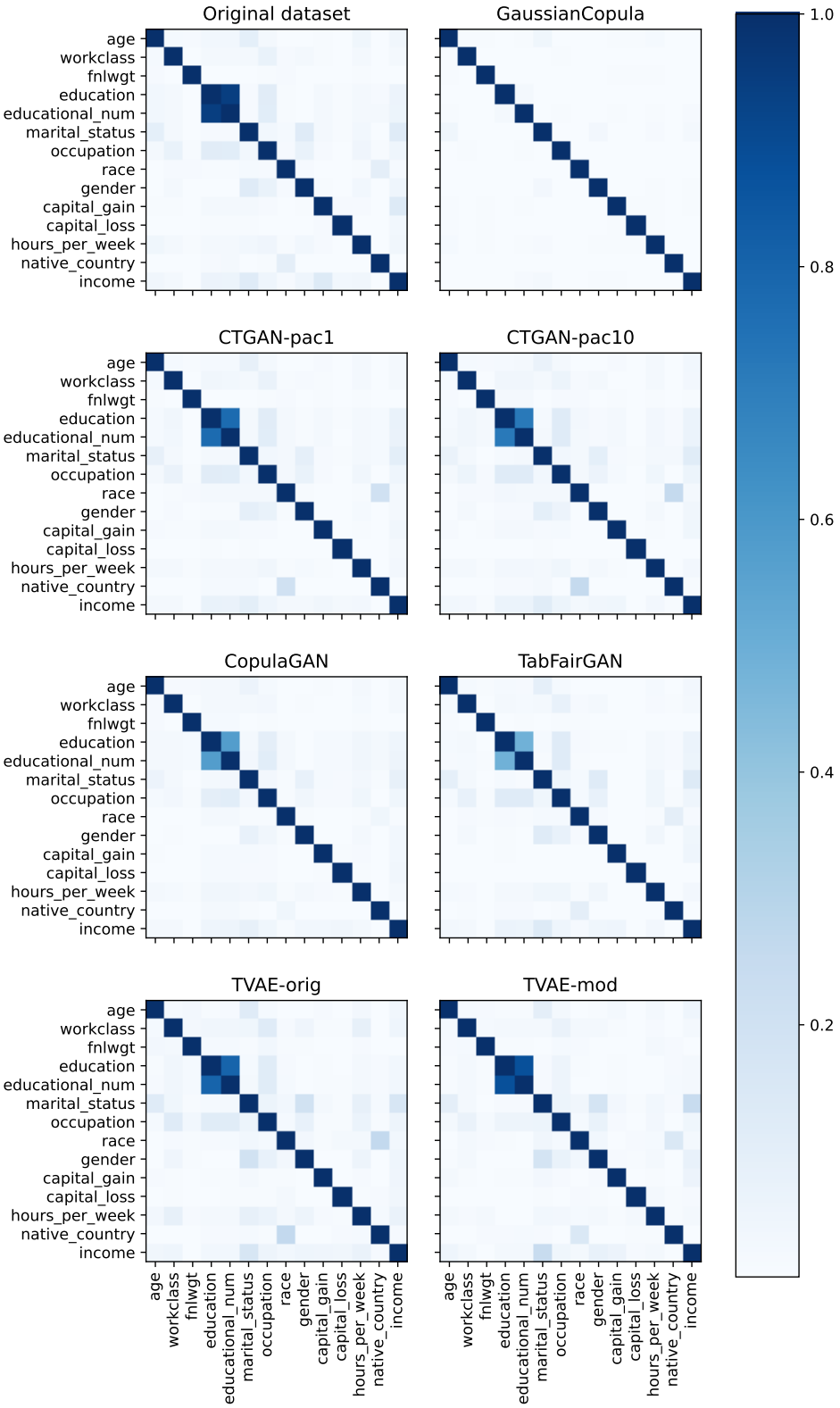


Figure 7.15: NMI matrices for the Adult training dataset and the synthesized datasets generated by each of the methods GaussianCopula, CTGAN-pac1, CTGAN-pac10, CopulaGAN, TabFairGAN, TVAE-orig, and TVAE-mod. The synthesized datasets were of the same size as the Adult training dataset.

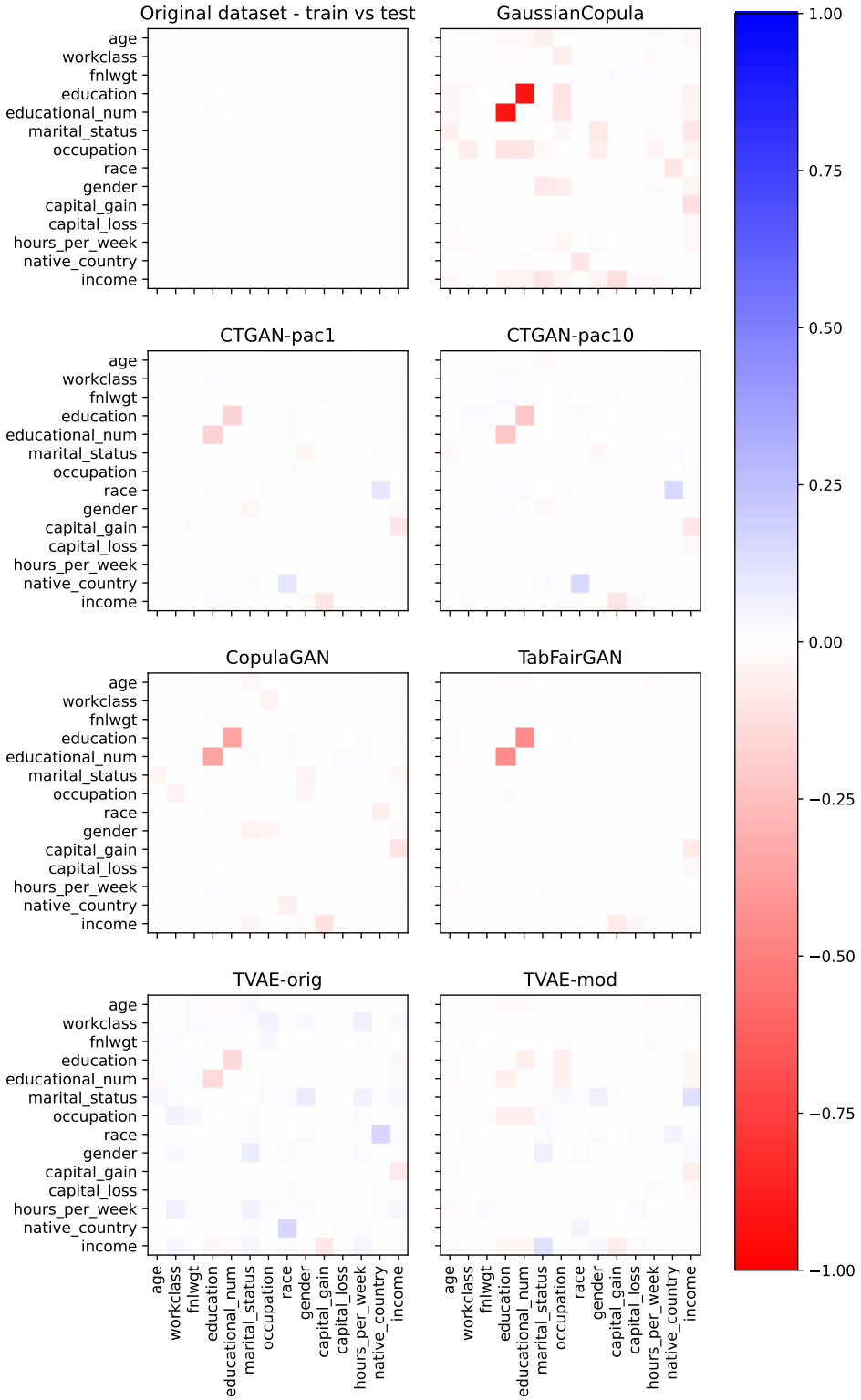


Figure 7.16: Difference between the NMI matrix for the Adult training dataset and the NMI matrices for the Adult test dataset and the synthesized datasets generated by each of the methods from the tabGAN framework: tabGAN-sd, tabGAN-qt, tabGAN-qtr, ctabGAN-sd, ctabGAN-qt, and ctabGAN-qtr.The synthesized datasets were of the same size as the Adult test dataset. With the chosen colormap, samples close to zero are shown as white even if they are not exactly zero.

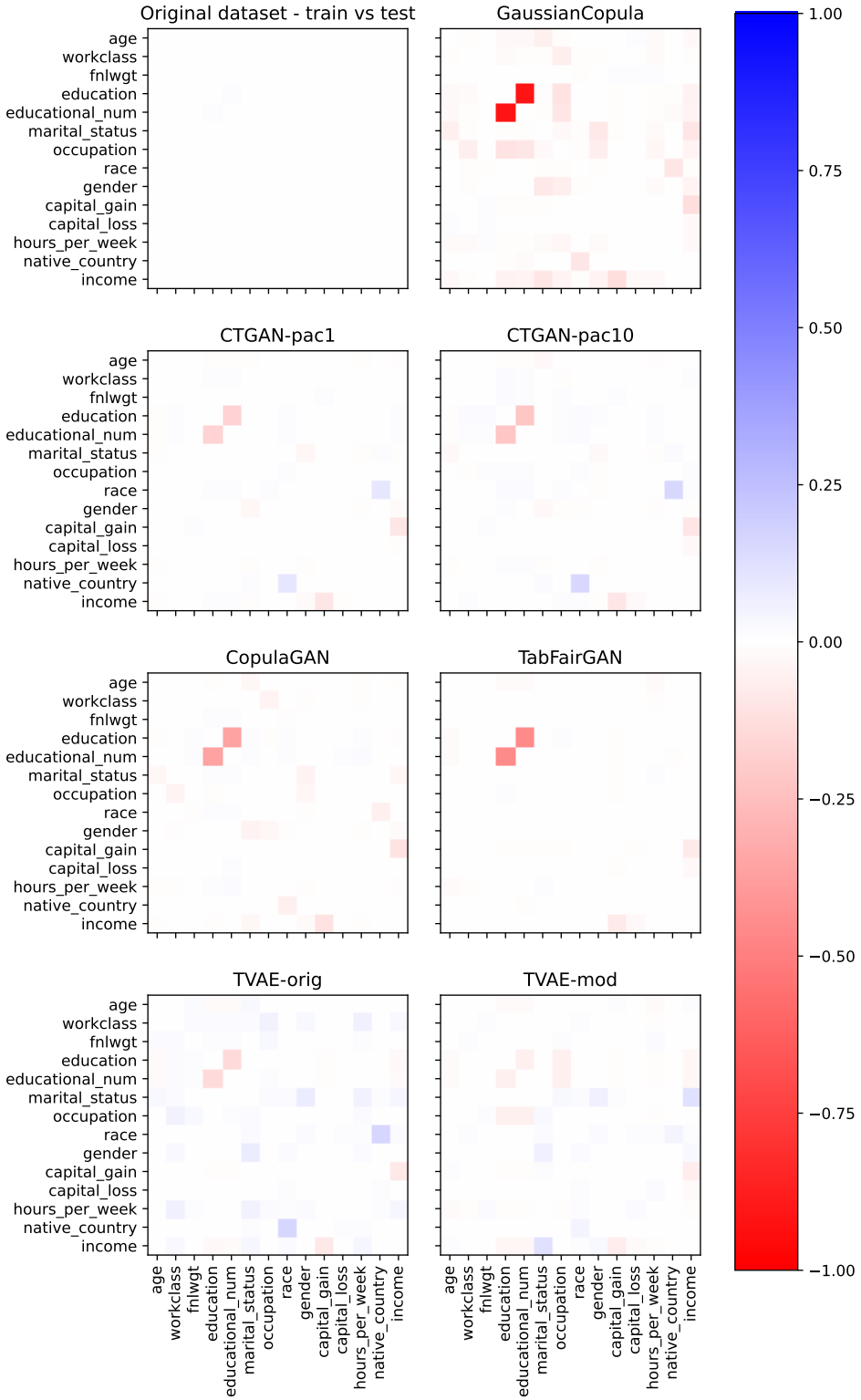


Figure 7.17: Difference between the NMI matrix for the Adult training dataset and the NMI matrices for the Adult test dataset and the synthesized datasets generated by each of the methods GaussianCopula, CTGAN-pac1, CTGAN-pac10, CopulaGAN, TabFairGAN, TVAE-orig, and TVAE-mod. The synthesized datasets were of the same size as the Adult test dataset. With the chosen colormap, samples close to zero are shown as white even if they are not exactly zero.

7.5 Comparing Machine Learning Efficacy on Adult Dataset

A recognized quantitative method for evaluating data synthesizers is to compare machine learning efficacy [1, 3, 13]. The process of evaluating machine learning efficacy can be described as follows. First a data synthesizer is trained on a training dataset. Then, the synthesizer generates a synthetic data set of the same size as the original training dataset. Finally, the machine learning model fitted to the synthesized dataset is evaluated on the original test dataset. If a data synthesizer is skilled at replicating the true data distribution, the model trained on the synthesized training dataset should be almost as good as the model trained on the true training dataset. For the Adult dataset we perform this process for each of the data synthesizers listed in this thesis. As response variable we set "income", which is the intended response variable by the UCI Machine Learning Repository where we fetched the Adult dataset. We replicate the process 50 times to obtain more stable results and obtain an estimate of the variability of the results. In this thesis, we use XGBoost (see Section 2.3) as the machine learning model. We choose XGBoost as it has been shown to perform very well on a wide range of datasets with very little hyperparameter tuning [38]. It might be even better to include an ensemble of machine learning models and average the results with each model when comparing machine learning efficacy, but we believe that the high performance of XGBoost at least partially makes up for it. An advanced prediction model is important because if the model does not utilize advanced patterns in the data, then the synthesizers that actually manage to replicate these advanced patterns do not get the appropriate credit for it. The paper [1] uses an ensemble of four models fitted to the Adult dataset, but the ensemble is outperformed by an XGBoost model without any hyperparameter tuning. Therefore, we are confident in the choice of XGBoost as our prediction model when evaluating the efficacy of machine learning. Still, we acknowledge that different types of models might potentially pick up on different types of data patterns.

We will use three classification metrics in the machine learning comparison; accuracy, AUC and F1 score. To accommodate the random components in the fitting and sampling process of a data synthesizer and get an overview of the variability, we will for each metric report the median and a centered 90% interval based on the percentiles after 50 runs.

7.5.1 Results and Discussion of Machine Learning Efficacy Comparison on Adult Dataset

In Table 7.1, we present the results of the benchmarking on the Adult dataset. The two clear winners are tabGAN-qt and tabGAN-qtr. The tabGAN-qtr method appears to have a tiny edge, with a higher median value for the accuracy and F1 score metrics, but the two methods are too close to be sure if it is not just due to coincidence. We observe that tabGAN-qtr appears to be slightly more stable, with the 5% percentile for tabGAN-qtr being, for each of the metrics, consistently higher than the 5% percentile for tabGAN-qt. Thus, we reason the Randomized Quantile Transformation might have some stabilizing properties compared to regular quantile transformation. The third best performing synthesizer is ctabGAN-qtr, which outperforms the ctabGAN-qt method. It appears that the randomization component was more important in combination with the conditional architecture than without it. We also note that for the methods from the tabGAN framework, the conditional methods in the framework, ctabGAN, consistently perform worse than the regular methods, tabGAN. This might, however, be specific to the Adult dataset and this machine learning task. It it interesting that the Randomized Quantile Transformation gave such a performance boost for the ctabGAN method and not for the tabGAN methods. We hypothesize that for the Adult dataset, the Randomized Quantile Transformation makes it easier for the generator to learn and that perhaps the effect decreases the better the rest of the synthesizer is. This would at least be consistent with what we have experienced throughout the hyperparameter tuning. In the beginning, the random-

ized version of the quantile transformation gave a significant boost to performance also for the tabGAN methods. However, as we continued to improve the rest of the synthesizer, such as architecture and hyperparameters, the effect decreased. If the Randomized Quantile Transformation truly makes it easier for the generator to learn, then it could potentially be the case that tabGAN-qtr reaches convergence faster during training than tabGAN-qt (even though the valued converged to is similar). This would be interesting to investigate.

The method tabGAN-sd is the fourth best synthesizer. It has lower median macro F1 score than ctabGAN-qt, but significantly outperforms ctabGAN-qt with respect to Accuracy and AUC. It is almost tied between the CTGAN-pac1 method and the ctabGAN-qt method, but CTGAN-pac1 appears to be the better performing of them, as it has slightly higher accuracy and macro F1 score. All the remaining methods perform a lot worse. It is interesting that ctabGAN-sd performs so much worse than ctabGAN-qt and ctabGAN-qtr, but an explanation could be that the conditional WGAN architecture and training process are a bit more difficult than the regular WGAN implementation for the Adult dataset. Consequently, the ability of the quantile transformation to transform continuous distributions to become more normal, is more valueable when the initial task is hard.

Among the six named methods from the tabGAN framework, the methods ctabGAN-sd, ctabGAN-qt, and ctabGAN-qtr are the ones that most closely resemble the CTGAN-pac1 method. They all use a conditional architecture with many similarities and the same conditional sampling strategy during training. They also use the same preprocessing method for discrete columns, i.e. one-hot encoding, and do not use packing. Thus, for an evaluation of the preprocessing method used in CTGAN for continuous columns, i.e. variational Gaussian mixture models (VGMM), these are the most useful methods as frames of reference. We observe that the method with Randomized Quantile Transformation (QTR), ctabGAN-qtr, outperforms CTGAN-pac1. It is, however, difficult to determine for certain how much of this is due to the transformation itself or instead, perhaps, a better architecture and hyperparameter choice. However, it is in indicator that the QTR may be more suited as preprocessing for the numerical columns in the Adult dataset. Although, even if that is the case, we do not know if it is more suitable for all of the numerical columns, or only some of them. Potentially, it could be that a few of the columns benefit more from VGMM than QTR, but this benefit is overshadowed by a few of the other columns that greatly benefit from the QTR as preprocessing function. ctabGAN-qt with regular quantile transformation performed on par with the CTGAN-pac1 method, but the ctabGAN-sd method was vastly outperformed. This might indicate that using variational Gaussian mixture models is better as a preprocessing step for the Adult dataset than mere standardization. Standardization can be linked to Gaussian mixture models by setting the number of models in the mixture to 1.

Another interesting aspect is the massive performance drop for the CTGAN method when packing is used. Consistent with what we observed for the methods in the tabGAN framework during hyperparameter tuning, packing is not beneficial for the Adult training dataset. Neither, does parametric quantile transformations appear to be beneficial as a preprocessing step before being input into the CTGAN framework, as CopulaGAN has worse performance than CTGAN-pac10. The only difference between CopulaGAN and CTGAN-pac10 is the parametric quantile transformation preprocessing step. Potentially, this also indicates that rank-based quantile transformation is superior to parametric quantile transformation for GAN structures trained on the Adult dataset.

Unsurprisingly, the GaussianCopula is the worst performer by a large margin in this category as well. More surprisingly is the lower performance of TVAE-orig and TVAE-mod. Variational autoencoder have been shown to be successful data synthesizers [1, 82, 83] and in the Xu *et al.* [1] paper the TVAE method outperformed CTGAN for most of the tests. It could, however, be that the Adult dataset did not suit the TVAE implementation. We observe that TVAE-mod outperforms TVAE-orig; apparently, the larger hidden layer sizes for both encoder and decoder are useful. The TabFairGAN

Table 7.1: Machine learning efficacy benchmarking for the different data synthesizers on the Adult dataset. The performance is calculated with respect to three metrics; Accuracy, AUC and Macro F1 score. For each metric we report the median and the interval defined by the 5% and 95% percentiles based on 50 runs for each synthesizer. Note that the interval reported is not a confidence interval measuring the variance of the experiment itself, as is quite common, but instead a measurement of the stability of the data synthesizing process. More accurately, the interval also includes the variability of the XGBoost fitting process, but as seen from the Train dataset row, this variability is negligible. For each metric and percentile we highlight the value of the best performing method in bold.

Method	Accuracy	AUC	Macro F1
Train dataset	0.8653 (0.8653, 0.8653)	0.9229 (0.9229, 0.9229)	0.8087 (0.8087, 0.8087)
tabGAN	0.8395 (0.8167, 0.8444)	0.8941 (0.8789, 0.8980)	0.7665 (0.7290, 0.7817)
tabGAN-qt	0.8427 (0.8385, **0.8459**)	0.8968 (0.8932, 0.8989)	0.7774 (0.7700, **0.7835**)
tabGAN-qtr	**0.8430** (**0.8401**, 0.8458)	**0.8976** (**0.8951**, **0.8999**)	**0.7789** (**0.7739**, **0.7835**)
ctabGAN	0.8296 (0.8090, 0.8386)	0.8838 (0.8685, 0.8924)	0.7542 (0.6817, 0.7701)
ctabGAN-qt	0.8374 (0.8323, 0.8405)	0.8897 (0.8860, 0.8920)	0.7672 (0.7589, 0.7733)
ctabGAN-qtr	0.8412 (0.8377, 0.8448)	0.8958 (0.8933, 0.8979)	0.7762 (0.7721, 0.7818)
GaussianCopula	0.7557 (0.7549, 0.7572)	0.8289 (0.7899, 0.8507)	0.4364 (0.4325, 0.4437)
CTGAN-pac1	0.8376 (0.8328, 0.8422)	0.8869 (0.8828, 0.8908)	0.7685 (0.7508, 0.7756)
CTGAN-pac10	0.8304 (0.8248, 0.8367)	0.8814 (0.8750, 0.8855)	0.7583 (0.7384, 0.7668)
CopulaGAN	0.8272 (0.8202, 0.8319)	0.8678 (0.8581, 0.8745)	0.7474 (0.7258, 0.7568)
tabFairGAN	0.8243 (0.8158, 0.8306)	0.8749 (0.8687, 0.8810)	0.7578 (0.7463, 0.7666)
TVAE	0.8177 (0.8046, 0.8290)	0.8714 (0.8617, 0.8800)	0.751 (0.7362, 0.7611)
TVAE-mod	0.8261 (0.8187, 0.8356)	0.8784 (0.8706, 0.8877)	0.7596 (0.7436, 0.772)

method performs on par with or perhaps a little bit better than CopulaGAN; however, when comparing it to tabGAN-qt, which is the method most resembling it, there is clear indication that the TabFairGAN method could easily have been improved upon. During hyperparameter tuning we explored using quantile transformation with uniform output distribution instead of Gaussian output distribution, but found that it reduced performance by a bit. However, we still observed much better performance than that of TabFairGAN. When combining this with the scalability issues discussed in Section 7.1.3, we believe that the TabFairGAN method could be improved by either switching to a Gaussian output distribution for the quantile transformation, or changing its architecture to better match that of tabGAN in the tabGAN framework (or better yet both).

The methods tabGAN-qt, tabGAN-qtr and ctabGAN-qtr appear to be the best performing by a large margin. However, until now we have only based the comparison on a single dataset, and we are interested in whether the results generalize well to other datasets. Additionally, the methods from the tabGAN framework might have an unfair advantage since their hyperparameter tuning was performed on the Adult dataset.

7.6 Comparing Machine Learning Efficacy on Three Other Real Datasets

In this section, we investigate whether the results from the previous section generalize to new datasets. We will compare machine learning efficacy for three new datasets, in a similar manner to that described in Section 7.5. However, unlike for the Adult dataset, the training and test portions are not predetermined. As some methods potentially could prefer certain subsets of the data, we

redraw the training and test selection for each run. To ensure fairness, the data synthesizers are given the same training and test split for a specific run. We repeat this 25 times for each data synthesizer. The three datasets are the Covertype dataset, the Credit Card Fraud dataset, and the Online News Popularity dataset, all described in Section 6.3. As machine learning task for each dataset, we will use the ones intended by UCI Machine Learning Repository for the Covertype and Online News Popularity dataset and the one listed on Kaggle for the Credit Card Fraud dataset. For the Covertype dataset this results in predicting the "Cover_Type" column, which is a classification problem with seven classes. As machine learning task for the Credit Card Fraud we attempt to predict the column "Class", i.e whether the transaction was a fraud or not. For the Online News Popularity dataset the task is to predict whether an article is popular or not. Popular is determined as those articles with more than 1400 shares on Mashable. When we train a data synthesizer on the Online News Popularity dataset, we will train it with the continuous column "shares" and only later on when we fit the XGBoost model to the synthesized dataset will we transform the "shares" column to be a discrete column with the two categories *>1400* and *<=1400*.

For the Covertype machine learning task, we will utilize accuracy and macro F1 score as metrics, as it is a multiclass classification problem with imbalanced classes. As the Credit Card Fraud dataset is extremely imbalanced with 284315 cases of not fraud and only 492 cases of fraud, we will use the metric area under the Precision-Recall curve (AUPRC) with fraud as the positive class. For the Online News Popularity dataset we will use accuracy and AUC as metrics, as it is a binary classification problem with balanced classes. When calculating the AUC for the Online News Popularity dataset we set *>1400* to be the positive class.

We choose to use these three datasets specifically due to their inclusion in the machine learning efficacy comparison in the CTGAN paper [1]. Though great datasets for a diverse evaluation of data synthesizers, unfortunately these datasets are not very suited for evaluating the Randomized Quantile Transformation method. Neither the Credit Card Fraud dataset nor the Online News Popularity dataset have any continuous columns with as much as a single repeated value. Thus, the tabGAN-qtr and ctabGAN-qtr implementations will be exactly the same as the tabGAN-qt and ctabGAN-qt implementations, respectively. The Covertype dataset has a single column with eight repeated values that each constitute a bit more than 5% of the observations and two other columns, each with a single repeated value that constitutes just above 5% of the observations. Here the Randomized Quantile Transformation will be applied, but the end result for the whole dataset will be pretty similar to the regular quantile transformation. Additionally, we believe that the Randomized Quantile Transformation is most beneficial for columns with disproportionately many repeated observations for a few values, such as in the Adult dataset. Thus, even though we will report the benchmarking scores for the tabGAN-qtr and the ctabGAN-qtr methods for all three datasets, the Randomized Quantile Transformation will barely be tested in this section.

7.6.1 Results and Discussion of Machine Learning Efficacy Comparison

In Table 7.2 we present the results of the benchmarking on the three datasets. The tabGAN framework continues to perform very well. All the best performing methods for each dataset and metric combination are from the tabGAN framework. For the Covertype dataset the six methods from the tabGAN framework all perform very similar, but they outperform all the other data synthesizers. ctabGAN-sd is the best performing method, and ctabGAN-qtr the second best. Although the results for the tabGAN framework methods are so similar that we can not rule out that the ranking order amongst themselves could change if the experient was rerun, the small percentile intervals indicate that the ranking order can be trusted to at least some degree. We note that the ctabGAN-qtr method outperforms ctabGAN-qt. The difference is quite small, but we observe that the 5% percentile of the ctabGAN-qtr method is the same as the median of the ctabGAN-qt method for both the accuracy

metric and the macro F1 score metric. This indicates that for the only dataset in this comparison for which the Randomized Quantile Transformation is applied, it leads to a slight increase in performance for the ctabGAN architecture. Similar to the comparison of machine learning efficacy for the Adult dataset in Table 7.1, we do not observe the same increase for the tabGAN architecture. As the methods from the tabGAN framework outperform all the other data synthesizer methods irrespective of preprocessing choice, it is an indication that it is the tabGAN framework architecture or hyperparameter choices that are superior to the other GAN based synthesizers. As one of the main differences between the ctabGAN implementation and the CTGAN implementation is the choice of GELU as activation function, this could potentially be an influencing factor for the better performance seen by the tabGAN framework methods.

Amongst the data synthesizers not belonging to the tabGAN framework, TVAE-mod is the best performer. Thereafter comes the the TVAE-orig method, closely followed by the tabFairGAN method. We observe that CopulaGAN performs slightly better than the CTGAN models, while CTGAN-pac1 has a tiny edge compared to CTGAN-pac10. Again, the model that performs the worst is GaussianCopula by a large margin. However, this is to be expected, as the Covertype dataset is a dataset with many discrete categories, and GaussianCopula does not have a good approach to tackle discrete columns. It should also be mentioned that the GaussianCopula method, for more than half of the runs fails to produce a dataset where all the categories of the "Cover_Type" column are represented. As the median and percentiles are calculated only from the runs in which it is possible to fit an XGBoost model, those runs were not used in the estimation of the median and percentiles. In Table 7.3 we give for each dataset an overview of the fraction of times each method fail to generate samples from all categories in the response variable. We observe that GaussianCopula is the only method that sometimes fails for the Covertype dataset.

For the Credit Card Fraud dataset, we see a drastic reorganization of the ranking. Most notably, is the fact that all of the GAN synthesizers without a conditional architecture and training process, struggle immensely. This is not surprising, as doing well in this machine learning task, requires the synthesizer to capture the distribution of the fraud observations in the dataset and they are an extreme minority. The regular GAN methods clearly fail to recreate the distribution of the fraud variables. This can be due to the synthesizers almost ignoring the fraud category or not managing to capture the distinguishing features of these fraud observations, or perhaps a combination of both. The tabGAN-sd method fails completely and receives a median AUPRC value close to zero. We observe, however, that the 95% percentile for the method is a decent value, thus for a few runs, the method must have managed to capture the data distribution better. Also the tabGAN-qt method receives a median AUPRC value close to zero, while the tabGAN-qtr method receives a better median AUPRC value. It would be easy to make the mistake of concluding that the tabGAN-qtr method performs better than the tabGAN-qt method on the Credit Card Fraud dataset. However, as we mentioned in the introduction to this section, the QTR method is identical to the regular quantile transformation for the Credit Card Fraud dataset since there are no repeated values for any of the numerical columns. Thus, this difference in AUPRC value must be due to chance. The CopulaGAN also performs very poorly for this dataset. The next three methods with a tied median AUPRC value are TabFairGAN, TVAE-orig, and TVAE-mod. Neither of them have a very impressive score, but looking at the percentile intervals, the TabFairGAN method is definitively the worst performing of them. Moving on to the first methods that perform decently, we have ctabGAN-sd, CTGAN-pac1 and CTGAN-pac10. Perhaps a bit suprisingly, GaussianCopula is one of the best-performing models for this dataset. This is probably due to the Credit Card Fraud dataset having mostly numerical columns. The only discrete column is the "Class" column and it only has two categories. However, from Table 7.3 we observe that for more than half of the runs, the GaussianCopula method did not produce samples from both categories of the "Class" column, thus making the performance a bit

less impressive. The best performing methods, however, are ctabGAN-qt and ctabGAN-qtr, which for this dataset are identical methods.

The most successful methods for the Online News Popularity dataset are tabGAN-qt, tabGAN-qtr, ctabGAN-qt, and ctabGAN-qtr. To avoid confusion, we repeat that tabGAN-qtr and ctabGAN-qtr are identical to tabGAN-qt and ctabGAN-qt for this dataset. The data synthesizers are much more evenly matched for this dataset, and the remaining methods are all relatively close in measured performance. A potential reason for why we see such little difference in machine learning efficacy for this dataset, is that the associated machine learning problem is pretty hard. It can very well be that the first gain in classification performance is quite easy to get, but that the remaining gap in performance up to the true training dataset is out of reach for all of the data synthesizers. However, this is merely speculation.

Table 7.2: Machine learning efficacy benchmarking for the different data synthesizers on the three real datasets described in Section 6.3. The performance is calculated with respect to different metrics suitable to each specific machine learning task. For each metric we report the median and the interval defined by the 5% and 95% percentiles based on 25 runs for each synthesizer. Note that the interval reported is not a confidence interval measuring the variance of the experiment itself, as is quite common, but instead a measurement of the stability of the data synthesizing process and the impact of the split into training and test sets. More accurately, the interval also includes the variability of the XGBoost fitting process, but this variability is negligible. For readability, we refer to the Credit Card Fraud dataset merely as Credit Card in the header. For each metric and percentile we highlight the value of the best performing method in bold.

	Covertype		Credit Card	Online News Popularity	
Model	Accuracy	Macro F1	AUPRC	Accuracy	AUC
Train dataset	0.870 (0.868, 0.872)	0.866 (0.864, 0.869)	0.867 (0.842, 0.890)	0.657 (0.650, 0.661)	0.715 (0.709, 0.722)
tabGAN	0.791 (0.788, 0.793)	**0.802** (**0.798**, **0.804**)	0.129 (0.001, 0.686)	0.594 (0.545, 0.615)	0.650 (0.602, 0.663)
tabGAN-qt	0.788 (0.785, 0.790)	0.798 (0.795, 0.800)	0.076 (0.001, 0.563)	**0.629** (**0.622**, **0.636**)	**0.677** (**0.668**, 0.684)
tabGAN-qtr	0.788 (0.786, 0.790)	0.798 (0.796, 0.800)	0.501 (0.001, 0.697)	0.625 (0.619, 0.635)	0.672 (0.666, **0.685**)
ctabGAN	**0.795** (**0.792**, **0.798**)	0.800 (0.797, **0.804**)	0.753 (0.681, 0.806)	0.589 (0.531, 0.619)	0.644 (0.597, 0.668)
ctabGAN-qt	0.790 (0.788, 0.794)	0.795 (0.793, 0.798)	**0.794** (0.746, **0.836**)	0.623 (0.617, 0.633)	0.673 (0.663, 0.680)
ctabGAN-qtr	0.790 (0.789, 0.793)	0.795 (0.794, 0.798)	0.792 (**0.754**, **0.836**)	0.625 (0.617, 0.635)	0.673 (0.661, 0.683)
GaussianCopula	0.578 (0.509, 0.646)	0.509 (0.336, 0.681)	0.787 (0.783, 0.791)	0.585 (0.572, 0.601)	0.619 (0.602, 0.634)
CTGAN-pac1	0.668 (0.653, 0.682)	0.702 (0.694, 0.716)	0.745 (0.701, 0.796)	0.602 (0.579, 0.617)	0.651 (0.637, 0.665)
CTGAN-pac10	0.635 (0.623, 0.640)	0.683 (0.664, 0.690)	0.740 (0.689, 0.786)	0.603 (0.572, 0.616)	0.645 (0.632, 0.653)
CopulaGAN	0.693 (0.685, 0.697)	0.720 (0.715, 0.724)	0.344 (0.119, 0.543)	0.605 (0.592, 0.618)	0.646 (0.632, 0.663)
tabFairGAN	0.717 (0.709, 0.724)	0.737 (0.728, 0.742)	0.501 (0.001, 0.501)	0.592 (0.581, 0.613)	0.630 (0.613, 0.650)
TVAE	0.718 (0.708, 0.731)	0.741 (0.730, 0.749)	0.501 (0.487, 0.761)	0.576 (0.550, 0.596)	0.639 (0.610, 0.658)
TVAE-mod	0.736 (0.720, 0.742)	0.753 (0.744, 0.760)	0.501 (0.380, 0.684)	0.575 (0.557, 0.598)	0.645 (0.631, 0.662)

Table 7.3: Overview of the fraction of times each method failed to synthesize a dataset that contained all of the categories in the response variable. The fractions of failures are reported separately for each dataset.

Methods	Covtype	Credit Card Fraud	Online News Popularity
tabGAN-sd	0.00	0.00	0.0
tabGAN-qt	0.00	0.00	0.0
tabGAN-qtr	0.00	0.00	0.0
ctabGAN-sd	0.00	0.00	0.0
ctabGAN-qt	0.00	0.00	0.0
ctabGAN-qtr	0.00	0.00	0.0
GaussianCopula	0.56	0.64	0.0
CTGAN-pac1	0.00	0.00	0.0
CTGAN-pac10	0.00	0.00	0.0
CopulaGAN	0.00	0.00	0.0
TabFairGAN	0.00	0.04	0.0
TVAE	0.00	0.20	0.0
TVAE-mod	0.00	0.20	0.0

7.7 Comparison of Training Speed

As an additional analysis we investigate how much time is required to train each data synthesizer method for 300 epochs with batch size set to 500 on the Adult training dataset. In the measured time, we also include the time required to synthesize a fake dataset of the same size. As hardware for this test we will use a Dell XPS 15 9570 with Intel Core i7-8750H processor, 16GB RAM and Nvidia GeForce GTX 1050 Ti with 4 GB RAM. The results are given in Figure 7.18. The first thing we notice is that the methods based on the CTGAN implementation, that is CTGAN-pac1, CTGAN-pac10 and CopulaGAN, are by far the most time-consuming methods to train. They each take around an hour to train for the Adult training dataset, which compared to the Covertype dataset and the Credit Card Fraud dataset is a quite small dataset. From Figure 7.18 we observe that the synthesizers which are not GAN based, are the fastest to train. This is not surprising as GAN based methods are generally computationally intensive to train and even more so Wasserstein GANs. For each update to the generator, we perform n_{critic} updates to the critic. For all the WGAN based methods in this thesis, that is TabFairGAN, CTGAN-pac1, CTGAN-pac10, CopulaGAN and all of the methods from the tabGAN framework, we set $n_{\text{critic}} = 10$. Thus, we perform a lot of training steps for the critic for each of the generator updates. This probably explains why the variational autoencoder methods, which are also regarded as quite computationally intensive to train, use less than half the time of the tabGAN methods, a quarter of the time of the ctabGAN methods and less than a tenth of the time used by CTGAN-pac1, CTGAN-pac10 and CopulaGAN. The fastest method, however, is the GaussianCopula method. This is unsurprising as the process of fitting marginal distributions, transforming each column to be Gaussian distributed and calculating the covariance matrix, is a lot less computationally heavy than training two neural networks, as done by both the GAN and the variational autoencoder based methods. Training the GaussianCopula took around 170 seconds compared to a bit less than 250 seconds for the TVAE methods. The three tabGAN methods, that is tabGAN-sd, tabGAN-qt, and tabGAN-qt, each take between 550-610 seconds to train. The TabFairGAN method, however, takes around 800 seconds to train. It is quite impressive that the tabGAN methods are faster than the TabFairGAN method, as the TabFairGAN implemenation uses a less advanced architecture for the neural networks and, more importantly, much smaller layer sizes.

We observe that the training time almost doubles when using ctabGAN instead of tabGAN. We believe this increase is almost entirely caused by the process which fetches observations from the real training dataset that matches the conditional queries, i.e the the specific category for a single column which is conditioned on (as described in Section 3.2.3). This process is quite time consuming as for each conditional vector $\mathbf{m}'$, the matching observations in the training dataset must be identified and then uniformally sampled from. This quickly adds up to a lot of extra time when it must be done for each real observation input to the critic. We did put effort into optimizing this process and ended up with a solution where we give all of the possible mask vectors a unique identifier id, which we use as key in a dictionary that stores the indices of the matching observations in the train dataset. The indices of the matching observations can then be easily identified through a single dictionary lookup, instead of each time having to search an entire dataset column for matching observations. As we receive a batch of conditional vectors at a time during the training phase, we vectorize the dictionary lookup and sampling process using numpy such that we avoid the slowness of "for loops" in Python. We believe this method is quite effective as we observe that the ctabGAN methods are around four times faster than the CTGAN methods, which employ the same conditional sampling process. Still, we believe there may be room for improvement. For the tabGAN methods, i.e not ctabGAN methods, we employ the whole data fetching process entirely in Tensorflow and we allow for the CPU to prefetch batches while the GPU is working. Since the data fetching process for ctabGAN involves a dictionary lookup, we were not able to do the same for

ctabGAN. However, perhaps there exists a workaround or another method altogether that will give an even larger performance boost. The fetching of real observations from conditional vectors does at least appear to be the current performance bottleneck, and it is where we believe any optimization attempt should be focused.

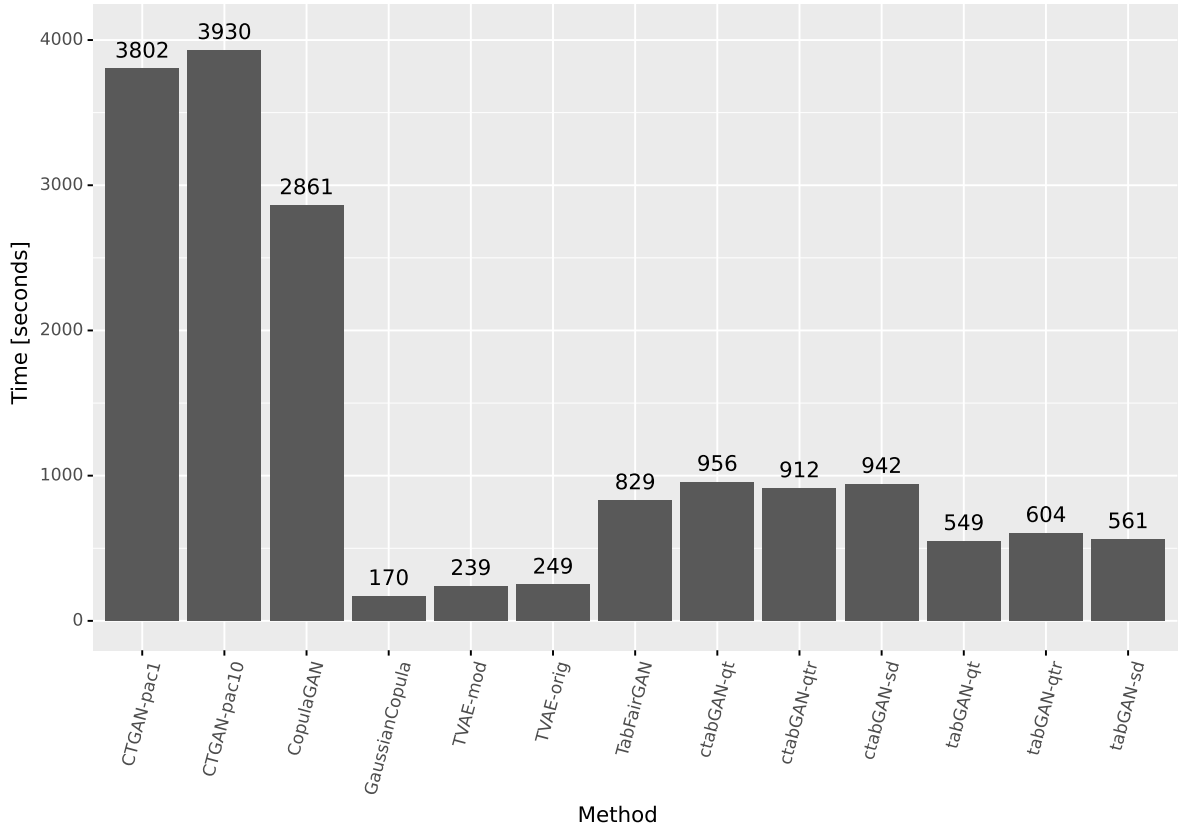


Figure 7.18: Total time used by each data synthesizing method when training for 300 epochs on the Adult train dataset with batch size set to 500 on a Dell XPS 15 9570 with Intel Core i7-8750H processor, 16GB RAM and Nvidia GeForce GTX 1050 Ti with 4 GB RAM. Each method was only timed for a single run and therefore a bit subject to random fluctuations.

Chapter 8

Experiments - Counterfactual Synthesizer

In this chapter, we perform some very simple experiments with the counterfactual framework described in Section 5.3. The experiments are very informal and constitute more of an exploratory analysis or proof-of-concept verification. We will use the dataset Syn_Moons described in Section 6.4. To further challenge the counterfactual synthesizers, we include a discrete dummy variable with two categories to the dataset (for each observation the category is randomly chosen with equal probability for each category). As prediction function, $\hat{f}$, we will use a support vector machine (SVM) model [84] trained to classify which of the moons (half circles) an observation belongs to. SVM is a supervised machine learning algorithm that is based on the idea of finding a hyperplane that best separates two classes. It has later been extended to regression tasks. A SVM uses a kernel to map the observations into high-dimensional feature spaces where a separating hyperplane can be found, thus facilitating non-linear decision boundaries. The choice of kernel varies. For this application we use a radial basis function kernel [85].

Throughout this chapter we will use a simple setup for the counterfactual methods. We will use quantile transformation as preprocessing method for the numerical variables in the dataset. For both generator and critic we will use two hidden layers of size 256 with the GELU activation function. As input to the generator we will use a latent noise layer of size 128 in addition to the counterfactual query (observation to be explained). We will use the Adam optimizer with parameters $\beta_1 = 0$ and $\beta_2 = 0.999$. The reason we choose to use $\beta_1 = 0$ instead of the default value of 0.7 used in the data synthesing framework tabGAN, is that $\beta_1 = 0$ appears to give more stable results for the counterfactual methods on this simple toy dataset for some reason. We will train each counterfactual method for 1000 epochs with a batch size of 500. As the Syn_Moons dataset has 5000 observations, this equals to 10 batches per epoch. As regularization coefficients we will mainly use $\alpha = 1$ for the numerical 1-norm penalty, $\beta = 0$ for the numerical squared 2-norm penalty, and $\gamma = 2$ for the discrete regularization penalty. Throughout this chapter we will use the same 10 counterfactual queries to evaluate the counterfactual methods.

8.1 Reweight Data Observations or Only Feed the Critic Observations of the Correct Prediction Class

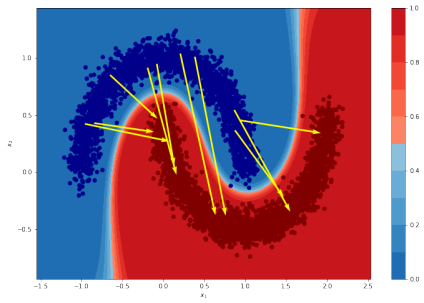
In Section 5.3.1, we introduced two options for guiding the generator to produce samples correctly classified by a black-box classifier, $\hat{f}$; either reweighting the real data observations to be proportional to their prediction function value, or only feeding the critic real data observations with the

correct prediction class without any reweighting. In this section we will informally compare these two approaches. We denote the two approaches tabGANcf-reweight and tabGANcf-onlyFeed, respectively. After training the two methods on the Syn_Moons dataset, we generate counterfactual explanations for the ten counterfactual queries of interest, two times for each method. The counterfactual explanations are plotted in Figure 8.1.

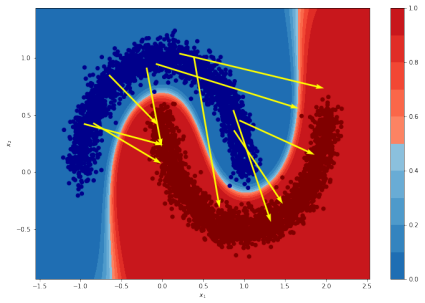
In Figure 8.1a and Figure 8.1b, we plot two different tabGAN-reweight synthesized versions of counterfactual explanations for the ten counterfactual queries of interest. The tabGANcf-reweight method appears to perform decently. In Figure 8.1a, all the counterfactual observations manage to flip the prediction function to the correct prediction class and they are plausible, as they are located in densely packed regions of the original data distribution. Additionally, they are actually relatively close to their corresponding counterfactual queries. In Figure 8.1b, the generated counterfactuals are a bit less successful. All of them still manage to flip to the correct prediction class (although just barely for one of them), but two of the generated counterfactuals are located way outside the data manifold. Still, 18 out of 20 decent counterfactuals are pretty good. We also observe some variety in the counterfactual explanations produced from the same query. The two different tabGAN-onlyFeed synthesized versions of counterfactual explanations for the same ten counterfactual queries of interest, are given in Figure 8.1c and Figure 8.1d. The tabGAN-onlyFeed method also appears to perform very decently. All of the twenty counterfactual explanations are of the correct prediction class and they adhere to the data manifold. Actually, compared to the tabGAN-reweight method, the counterfactuals from the tabGAN-onlyFeed method adhere better to the original data distribution. The explanations from tabGAN-reweight are more often located in the outskirts of the densely located regions, while the explanations from tabGAN-onlyFeed are generally more often located towards the middle of the densely packed regions. Additionally, we observe a bit more variety in the explanations from tabGAN-onlyFeed, but not necessarily all in a good way. We observe that the counterfactuals from the tabGAN-onlyFeed method generally are further from the corresponding counterfactual queries than from the tabGAN-reweight method.

A trend we observe in the plots for both methods, is that the counterfactual observation arrows tend to all move in a rightward direction. While this is natural for the counterfactual queries situated at the left part of the blue half moon, for the counterfactual queries located in the right half of the blue moon, it would make just as much sense for the counterfactual explanation to move leftwards. The reason we observe this pulling to the right, is that we force the counterfactual explanations to recreate the original data distribution of the red moon. Consequently, the generator finds that the easiest compromise is to generally produce counterfactual observations that are located downwards and to the right of the counterfactual queries. This is the problem we discussed in the second extension in Section 5.3.2.

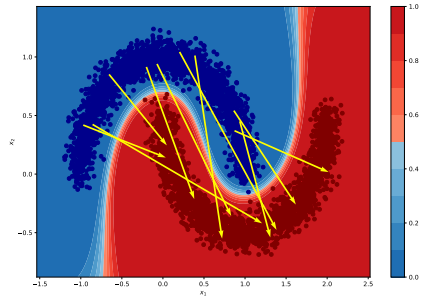
We are currently a bit unsure why the counterfactual observations from the tabGAN-reweight method are further away from the densely packed regions and closer to the counterfactual observations than those from the tabGAN-onlyFeed method. In practice, with this prediction function that is practically equal to either 0 or 1 for all input, the optimal generator distribution should be very similar for both tabGAN-reweight and tabGAN-onlyFeed. The current hypothesis of the author, is that the large share of real observations (approximately half) that have their weight set to zero in the loss function, although the weight of the remaining real observations are doubled, somehow makes it harder for the critic to learn the correct output distribution. With the critic being less certain of the correct output distribution, the generator gets less correct feedback and the regularizing loss term thus has greater influence on the learning of the generator. This is, however, mere speculation and more investigation is needed.



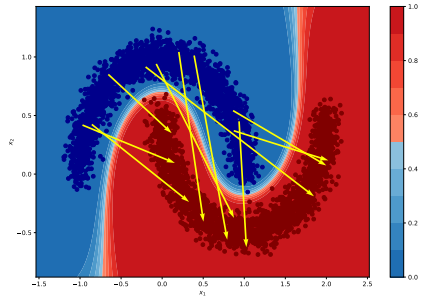
(a) tabGANcf-reweight – Example 1.



(b) tabGANcf-reweight – Example 2.



(c) tabGANcf-onlyFeed – Example 1.



(d) tabGANcf-onlyFeed – Example 2.

Figure 8.1: Example of generated counterfactuals from the two methods tabGAN-reweight and tabGAN-onlyFeed. For each method we give two examples of a synthesized set of counterfactual explanations. For comparability, we use the same ten counterfactual queries for each method and example. Each counterfactual is denoted by an arrow that stretches from the counterfactual query to the corresponding counterfactual observation. The arrows are color coded as yellow if the counterfactual explanation correctly flips the prediction class and color coded as magenta if it does not.

Nr.	x1	x2	dummy
0	0.936726	0.450238	dummy0
1	-0.655622	0.850763	dummy1
2	-0.071853	0.940222	dummy1
3	0.871999	0.369955	dummy0
4	-0.201285	0.914596	dummy0
5	-0.856179	0.423720	dummy0
6	0.863761	0.541687	dummy1
7	0.389646	1.012091	dummy1
8	0.203196	1.042454	dummy0
9	-0.969989	0.418297	dummy0

Table 8.1: The ten counterfactual queries repreatedly used in this chapter.

Nr.	x1	x2	dummy
0	0.936726	0.450238	dummy0
1	-0.655622	0.850763	dummy1
2	-0.071853	0.940222	dummy1
3	0.871999	0.369955	dummy0
4	-0.201285	0.914596	dummy0
5	-0.856179	0.423720	dummy0
6	0.863761	0.541687	dummy1
7	0.389646	1.012091	dummy1
8	0.203196	1.042454	dummy0
9	-0.969989	0.418297	dummy0

Table 8.2: Ten counterfactual explanations generated by tabGAN-onlyFeed from the counterfactual queries listed in Table 8.1.

8.2 Are Unnecessary Changes Made to the Discrete Column Values for the Counterfactual Explanations?

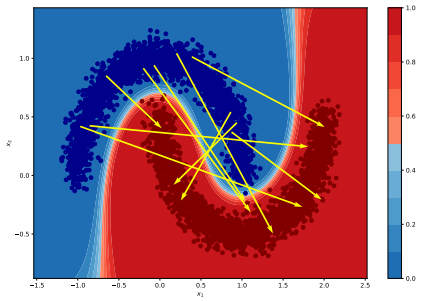
In this section we explore if the tabGAN-onlyFeed method manages to understand that the discrete dummy column added to the Syn_Moons dataset, has no effect on the prediction function $\hat{f}$ and therefore definitely should not be changed in the generated counterfactual explanations. In Table 8.1 we list the values for the ten counterfactual queries repeatedly used in this chapter, and in Table 8.2 we list the ten corresponding counterfactual explanations generated by tabGAN-onlyFeed. We observe that none of the counterfactual explanations suggest to change the value in the "dummy" column, thus indicating that the counterfactual method is able to not make unnecessary changes for discrete columns. To make it a bit more formal, we repeat this for 5000 queries and 5000 corresponding counterfactual explanations, none of them which suggest to change the discrete column value.

8.3 Investigation of the Impact of the 1-norm Numerical Penalty Coefficient

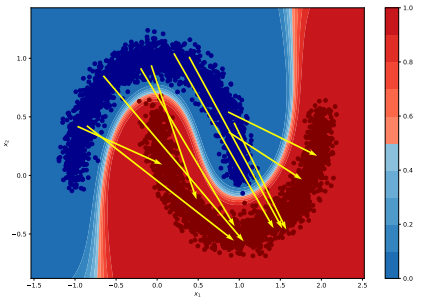
We want to better understand how the coefficient value of the numerical 1-norm regularization, α, affects the counterfactual observations. In Figure 8.2 we plot ten counterfactuals for the tabGAN-onlyFeed method for four different α values: 0, 0.5, 1 and 1.5. As expected, we observe from Figure 8.2a that without the 1-norm regularization (and also with no squared 2-norm regularization), the counterfactual synthesizer very well manages to flip the prediction class, but do not care how close the counterfactual explanation is to the counterfactual query. We assume the generator merely disregards the counterfactual query and instead generates samples from the red moon data distribution. Coincidentally, this is also the reason why we see the first counterfactual explanations that are directed leftwards. With $\alpha = 0.5$, the rightwards pattern in the counterfactuals is back and the counterfactuals explanations tend to be closer to the counterfactual queries. Still, we observe that many of the counterfactual explanations are quite far from the corresponding counterfactual queries. The counterfactuals for $\alpha = 0.5$ and $\alpha = 1$ appear to be quite similar. However, we observe as expected that the counterfactuals from the method with $\alpha = 1$ tend to be a bit closer to the queries than those for the method with $\alpha = 0.5$. For the counterfactuals from the method with $\alpha = 1.5$ plotted in Figure 8.2d, we observe a drastic change compared to the method with $\alpha = 1$. Now, two of the counterfactual explanations fail to flip the prediction class and out of the remaining eight, most are located in the less densely packed regions. Although none of the α values give methods with very good counterfactuals, we feel $\alpha = 1$ appears to strike the best compromise between adhering to the data distribution and synthesizing counterfactual explanations close to the data distribution.

8.4 Investigation of the Impact of Mixing Real Correct Samples with the Generator Samples

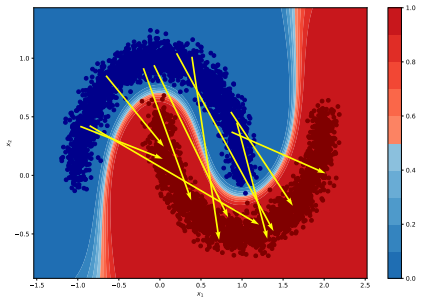
In this section we informally explore if the second extension mentioned in Section 5.3.2 can be beneficial, that is to mix samples from the generator in the loss function with real observations of the correct prediction class. We plot the result of a single run with and without this extension in Figure 8.3. For the extended method we add another 500 real observation of correct class mixed in between the generator samples in the loss function. To compensate for doubling the number of "samples from the generator", we give each of them half the weight they normally would have in the loss function. From the figure we observe that the extension does not appear to provide the desired difference. We still observe counterfactual observations mapped unnecessarily far away from the counterfactual queries. Additionally, there might be a small indication that the method with extension actually maps more to less densely packed regions. If this is actually the case, it could be due to the extra noise introduced by mixed-in samples, but it is hard to tell from this plot alone if this actually is the case. When we rerun the experiment, we continue to observe a slight indication of less mapping to the center of the densely packed regions. Nevertheless, the extension does not appear to provide the intended change. It might, however, be a bit too quick to dismiss the extension after a tiny single experiment. The number of added samples can be considered a hyperparameter that can be tuned, therefore further investiations might want to investigate other proportions of added observations. Additionally, it could potentially be better to keep the total number of generated and added samples constant, instead of just adding more samples as we did in this experiment.



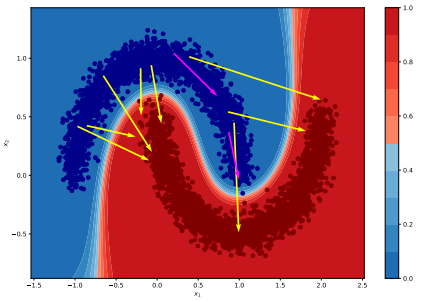
(a) tabGANcf-onlyFeed with $\alpha = 0$.



(b) tabGANcf-onlyFeed with $\alpha = 0.5$.

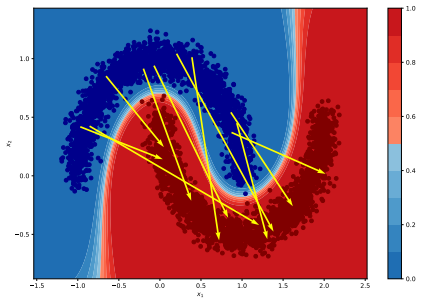


(c) tabGANcf-onlyFeed with $\alpha = 1$.

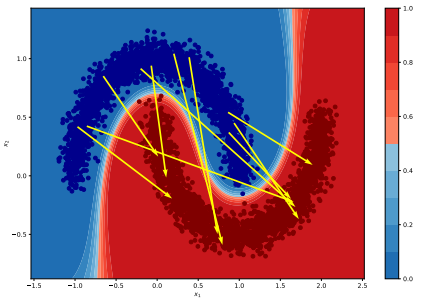


(d) tabGANcf-onlyFeed with $\alpha = 1.5$.

Figure 8.2: Example of generated counterfactuals from the tabGAN-onlyFeed method with four different values for the coefficient of the numerical 1-norm regularization term: 0, 0.5, 1 and 1.5. Each counterfactual is denoted by an arrow that stretches from the counterfactual query to the corresponding counterfactual observation. The arrows are color coded as yellow if the counterfactual explanation correctly flips the prediction class and color coded as magenta if it does not.



(a) tabGANcf-onlyFeed without extension.



(b) tabGANcf-onlyFeed with extension.

Figure 8.3: Example of generated counterfactuals from the tabGAN-onlyFeed method with and without an extension where real observations of the correct prediction class is mixed with the generator samples in the loss function. Each counterfactual is denoted by an arrow that stretches from the counterfactual query to the corresponding counterfactual observation. The arrows are color coded as yellow if the counterfactual explanation correctly flips the prediction class and color coded as magenta if it does not.

Despite this, the author feel less convinced that this extension has merit. Perhaps, it is a bit flawed to hope that adding samples matching the original data distribution will help. Maybe it would have been better to create an algorithm that only mixes in samples located in the high-density areas where it is undesirable for the synthesizer to map counterfactuals. However, this leaves the difficult task of deciding where such high-density areas are. An idea that comes to mind, is to automate this process of generating samples for the undesirable parts of the data distribution with correct prediction class, by adding a second generator. This generator could have the simple architecture with only a latent noise vector as input. The second generator would also have no regularization loss terms. The two generators might then collaborate to reproduce the entire real data distribution of the correct prediction class. Since the second generator do not have a regularization term, it would be more beneficial for it to map to the parts of the real data distribution not suited for counterfactual explanations, than it would be for the first and original generator with the regularization term that penalizes for mapping away from the counterfactual queries. The proportion of samples produced by each generator would be a hyperparameter.

Chapter 9

Discussion and Conclusion

In this thesis, we introduce a data synthesizing framework known as tabGAN and later modify it to create a model-based counterfactual synthesizer framework, which we call tabGANcf. The counterfactual framework is more a proof-of-concept, while the data synthesizing framework is more complete, with a lot of customization available and default values based on extensive hyperparameter tuning. From the results of the experiments performed in Chapter 7 we find that various diverse methods from the tabGAN framework are all very capable data synthesizers across multiple datasets. We verify this visually, with respect to marginal and joint distributions, and with respect to machine learning efficacy.

The three numerical preprocessing methods (standardization, quantile transformation and the Randomized Quantile Transformation) and the two main architectures (tabGAN and ctabGAN) appear to have different advantages. The default combination of tabGAN and standardization is a simple well-performing method that might even outperform the other more advanced methods for datasets with numerical column distributions that are approximately Gaussian. As indicated by the results, we believe that quantile transformation is more beneficial for numerical columns that are not approximately normally distributed, for instance due to having a different shape or not being continuous. Quantile transformation also appears to be a superior choice as it outperforms standardization on three out of four datasets, and whilst on the remaining dataset it is slightly beaten, it basically performs on par with standardization. We believe this is due to quantile transformation for columns with Gaussian distributions in practice being a very similar transformation to standardization, thus also being a good option for columns with already approximately Gaussian distribution. Building on this, we believe the Randomized Quantile Transformation (QTR) is the best-performing preprocessing method in this analysis and the superior choice. If there are no repeated values (or less repeated values than some user determined percentage), the QTR is identical to quantile transformation. However, when there are columns with many repeated values, we believe, as indicated by our result, that the QTR will usually outperform the regular quantile transformation. At least this was very much true for the Adult dataset. This should, however, be verified on more datasets with repeated values, as unfortunately the three additional datasets we used in the machine learning efficacy comparison barely had any repeated values. Additionally, it would be interesting to investigate whether QTR leads to faster convergence than regular quantile transformation. With the QTR the generator does not need to be as exact in its mapping to the repeated values, and the critic is not given an unfair advantage where gaps in the original data distribution might be easy to detect for the critic but hard for the generator to replicate.

Of the two main architectures, ctabGAN appears to be the safest choice and what we recommend by default. Although tabGAN actually performed slightly better on three out of the four datasets, it completely failed for the Credit Card Fraud dataset. The reason it failed is that the Credit Card

Fraud dataset has extremely imbalanced classes for the response variable, and the tabGAN methods are not as good for learning the distribution of rare categories as the ctabGAN methods. We also observed this while plotting the marginal distributions of the original Adult dataset against synthesized versions. This also highlights the limitations of using machine learning efficacy as the sole measurement of the performance of a data synthesizer. If a column or category is not an important variable in the prediction of the response variable in the chosen machine learning problem, a synthesizer might not be appropriately penalized for recreating it incorrectly or not being able to recreate the advanced patterns associated with this column or category. We believe it is important that a good synthesizer recreates all of the marginal distributions correctly and does not forget to sample from some of the rare cases. Thus, we do not use machine learning efficacy as the sole determinant of synthesizing skill, and we conclude that overall ctabGAN is a better choice than tabGAN. As we observe in this thesis, with a diverse choice of machine learning tasks, we are none the less able to uncover such limitations. A benefit of using tabGAN methods over ctabGAN methods is, as we see in Section 7.7, a faster training speed. The sampling of real observations from the original dataset conditioned on a specific column category is the reason for the slower training speed. Perhaps, with further optimization, this difference in training speed could be reduced.

When we compare the data synthesizers from the tabGAN framework with other state-of-the-art data synthesizers, we find that the methods from the tabGAN framework generally outperform all the other data synthesizers. This is very promising as the CTGAN and CopulaGAN are, excluding the methods introduced in this thesis, the best performing GAN based synthesizers the author is aware of. Even more, as we show in Section 7.7, the tabGAN framework implementation is actually a lot faster than both CTGAN and CopulaGAN. This, despite the fact that CTGAN and CopulaGAN have a similar model architecture and use the same conditional training process as the ctabGAN methods. It would be interesting to further investigate the underlying reasons for the better performance of the tabGAN framework. The current hypothesis of the author is that the usage of GELU as an activation function instead of LeakyReLU and ReLU is very beneficial. Additionally, we believe that for the Adult datset with repeated values, quantile transformation and QTR are more beneficial than variational Gaussian mixture model as preprocessing transformation. This might also be the case for the other datasets, but since both tabGAN and ctabGAN with standardization as numerical preprocessing also outperform CTGAN, we believe that GELU activation is a much more beneficial choice for the activation function of the hidden layers. Of course, there could potentially be other parts of the CTGAN (and thus also CopulaGAN) architecture that are less beneficial than the default architectures in the tabGAN framework. For instance, CTGAN uses batch normalization before applying the activation function, while we use batch normalization after applying the activation function to the hidden layers. Perhaps the most likely contributors to the difference in performance, except for the choice of the activation function for hidden layers, are the use of tanh by CTGAN as an activation function for the numerical output and the concatenation of each new dense hidden layer with the previous layer given as input to the new dense layer.

Another interesting conclusion that we can draw from the experiments in this thesis is that the default choice of using packing in the CTGAN method is not very beneficial. This is supported by the hyperparameter tuning we performed on the tabGAN framework. Consequently, the tabGAN framework by default do not use packing even though it is implemented. It would be interesting to investigate both for the CTGAN method and for the tabGAN framework methods if packing can be useful for datasets with extremely many modes such as for instance the MNIST3 dataset mentioned in Tolstikhin *et al.* [86], where each observation is obtained by concatenating three randomly chosen MNIST images [87] to form a three-digit number in the range between 0 and 999. It would also be interesting to investigate how the tabGAN framework performs on high-dimensional binary datasets such as for instance the binarized MNIST dataset used by Xu *et al.* [1]. Although the

methods from the tabGAN framework consistently outperform the CTGAN methods, we believe that the variational Gaussian mixture model used by CTGAN would be beneficial to include as a preprocessing option in the tabGAN framework. It could very well be that the variational Gaussian mixture model is a more beneficial preprocessing method for some datasets. Additionally, with the inclusion of the variational Gaussian mixture model preprocessing in the tabGAN framework, we could give a more clean comparison of it to the preprocessing options already implemented in CTGAN. It is difficult to draw a fair conclusion about the best preprocessing option, as CTGAN uses another model architecture, and therefore more than one thing is changed at once. Additionally, it would be useful to investigate the combination of different preprocessing options. For example, it might be beneficial to use QTR for numerical columns with many repeated values or not continuous distributions and use variational Gaussian mixture models for the remaining numerical columns. The author encourages future investigations into this.

The experiments in Chapter 8 provide a proof of concept for using GAN as a counterfactual explanation synthesizer. Although Yang *et al.* [5] and Nemirovsky *et al.* [78] already have provided a proof-of-concept, this is the first proof-of-concept we are aware of with public code available. Additionally, unlike Yang *et al.* [5], this implementation can be used by black-box classifiers. The implementation is quite a bit different from that described in Nemirovsky *et al.* [78], the most obvious difference being the usage of a Wasserstein GAN instead of a residual GAN. However, one of the options we implement in the tabGANcf framework is a modified version of the counterfactual loss function of Nemirovsky *et al.* [78] intended for black-box classifiers. For a toy dataset we show that two different methods from the tabGANcf framework are consistently able to generate counterfactual observations that adhere to the original data manifold and successfully flip the predicted class of a prediction function. Out of the two methods, we believe the method of only feeding the critic real data observations that are of the correct prediction class is both more efficient and beneficial than reweighting the real data observations in the loss function. However, more thorough investigation is needed.

An issue we observe for all methods of the tabGANcf framework is the lack of sparse changes for the continuous columns. We believe this is a inherent problem due to the chosen model architecture. In Section 5.3.2 we propose an extension consisting of two modifications that might enable sparse changes for the continuous columns. This is, however, not implemented due to constraints of time and space. We encourage future investigations into this and the other extensions proposed in the same subsection. We informally investigated one of the mentioned extensions, which was to mix real observations of the correct prediction class with the generator samples; however, the results were not very promising. A much more promising idea, we think, is to instead include a second generator without the regularization term and query input, such that the second generator can produce samples for the parts of the data distribution to where it is not beneficial for the main generator to map counterfactual explanations.

The counterfactual synthesizing framework tabGANcf is built on top of the data synthesizing framework tabGAN (as a Python class that inherits). While this was a very efficient option during the work leading up to this thesis, in which we could avoid to make the same code change two places, we believe the tabGANcf framework would benefit from being built from scratch. The code would be much more readable and simple if it could be rewritten to facilitate only the counterfactual synthesizing. This would definitely also make it easier to include future extensions, since the task of synthesizing counterfactuals is quite different from the task of synthesizing a dataset. Therefore, including new extensions usually requires rewriting the code even more generally, a task which becomes increasingly complex the more generalization is required. This would also hold true for the tabGAN framework. Additionally, we believe that if the tabGAN framework is to be useful as a package for others, it might actually benefit from fewer customization options than what is

implemented today. Besides the numerous customization options which we have described in this thesis, which we believe are beneficial, we also include many other customization options mainly for use during the hyperparameter tuning phase. It might also be advantageous to change some variable and helper function names and include some more user-friendly helper functions. If these things are fixed and it is published as a Python package, we believe it would be a good, and perhaps even preferable alternative, to the CTGAN Python package. An even better idea would be to, in the same way as CTGAN, include the tabGAN framework as a synthesizer method in the Synthetic Data Vault (SDV) package. Then it would automatically benefit from the other functionalities of the SDV package, such as handling of primary key columns (which must be unique for each observations) and tools for anonymizing personally identifiable information such as address [2].

Additionally, steps can be taken to further improve the tabGAN data synthesizing framework. The paper Tolstikhin *et al.* [86] proposes a boosting approach for GAN models, where a new component is added iteratively to a mixture GAN model. Each new component is added by running a GAN model on a reweighted dataset that emphasizes samples not already well covered by the mixture model. This mixture method can be implemented on top of all the GAN-based data synthesizers described in this thesis. Thus, it could be implemented as a separate framework on top of the tabGAN framework. Another promising possible improvement, could be to include a gradient layer in the generator architecture as described in Nitanda and Suzuki [88]. The gradient layer attempts to overcome the restrictions of a specific finite-dimensional model structure by seeking a descent direction in an infinite-dimensional space. Currently, the tabGAN framework is only capable of performing synthesis for a single dataset. In the future, it would be worth investigating if it can be extended to multiple related datasets or time-series datasets.

Bibliography

[1] L. Xu, M. Skoularidou, A. Cuesta-Infante and K. Veeramachaneni, *Modeling tabular data using conditional gan*, 2019. arXiv: 1907.00503 [cs.LG].

[2] N. Patki, R. Wedge and K. Veeramachaneni, 'The synthetic data vault,' in *2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, 2016, pp. 399–410. DOI: 10.1109/DSAA.2016.49.

[3] A. Rajabi and O. O. Garibay, *Tabfairgan: Fair tabular data generation with generative adversarial networks*, 2021. arXiv: 2109.00666 [cs.LG].

[4] M. Pawelczyk, K. Broelemann and G. Kasneci, 'Learning model-agnostic counterfactual explanations for tabular data,' *Proceedings of The Web Conference 2020*, Apr. 2020. DOI: 10.1145/3366423.3380087. [Online]. Available: http://dx.doi.org/10.1145/3366423.3380087.

[5] F. Yang, S. S. Alva, J. Chen and X. Hu, 'Model-based counterfactual synthesizer for interpretation,' *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, Aug. 2021. DOI: 10.1145/3447548.3467333. [Online]. Available: http://dx.doi.org/10.1145/3447548.3467333.

[6] J. Moore, N. Hammerla and C. Watkins, *Explaining deep learning models with constrained adversarial examples*, 2019. arXiv: 1906.10671 [cs.LG].

[7] C. Molnar, *Interpretable Machine Learning, A Guide for Making Black Box Models Explainable*. 2019.

[8] C. Molnar, G. Casalicchio and B. Bischl, *Interpretable machine learning – a brief history, state-of-the-art and challenges*, 2020. arXiv: 2010.09337 [stat.ML].

[9] A.-H. Karimi, B. Schölkopf and I. Valera, 'Algorithmic recourse: From counterfactual explanations to interventions,' in *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, 2021, pp. 353–362.

[10] S. Wachter, B. Mittelstadt and C. Russell, *Counterfactual explanations without opening the black box: Automated decisions and the gdpr*, 2018. arXiv: 1711.00399 [cs.AI].

[11] S. Dandl, C. Molnar, M. Binder and B. Bischl, 'Multi-objective counterfactual explanations,' *Lecture Notes in Computer Science*, pp. 448–469, 2020, ISSN: 1611-3349. DOI: 10.1007/978-3-030-58112-1_31. [Online]. Available: http://dx.doi.org/10.1007/978-3-030-58112-1_31.

[12] J. L. C. Michael Downs, Y. Yacoby, F. Doshi-Velez and W. Pan, 'Cruds: Counterfactual recourse using disentangled subspaces,' *ICML Workshop on Human Interpretability in Machine Learning*, pp. 1–23, 2020. [Online]. Available: https://finale.seas.harvard.edu/files/finale/files/cruds-_counterfactual_recourse_using_disentangled_subspaces.pdf.

[13] L. Xu and K. Veeramachaneni, *Synthesizing tabular data using generative adversarial networks*, 2018. arXiv: 1811.11264 [cs.LG].

[14] A. Rustad, 'Model-based counterfacual synthesizer using tabular gan and a comparison to algorithmic based counterfactuals,' unpublished, 2021.

[15] F. Rosenblatt, 'The perceptron: A probabilistic model for information storage and organization in the brain.,' *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[16] C. Ma, S. Wojtowytsch, L. Wu *et al.*, 'Towards a mathematical understanding of neural network-based machine learning: What we know and what we don't,' *arXiv preprint arXiv:2009.10713*, 2020.

[17] V. Nair and G. E. Hinton, 'Rectified linear units improve restricted boltzmann machines,' in *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ser. ICML'10, Haifa, Israel: Omnipress, 2010, pp. 807–814, ISBN: 9781605589077.

[18] I. Goodfellow, Y. Bengio and A. Courville, *Deep learning (Adaptive Computation and Machine Learning series)*. MIT press, 2016.

[19] A. L. Maas, A. Y. Hannun, A. Y. Ng *et al.*, 'Rectifier nonlinearities improve neural network acoustic models,' in *Proc. icml*, Citeseer, vol. 30, 2013, p. 3.

[20] D.-A. Clevert, T. Unterthiner and S. Hochreiter, *Fast and accurate deep network learning by exponential linear units (elus)*, 2015. DOI: 10.48550/ARXIV.1511.07289. [Online]. Available: https://arxiv.org/abs/1511.07289.

[21] G. Klambauer, T. Unterthiner, A. Mayr and S. Hochreiter, 'Self-normalizing neural networks,' 2017. DOI: 10.48550/ARXIV.1706.02515. [Online]. Available: https://arxiv.org/abs/1706.02515.

[22] P. Ramachandran, B. Zoph and Q. V. Le, *Searching for activation functions*, 2017. DOI: 10.48550/ARXIV.1710.05941. [Online]. Available: https://arxiv.org/abs/1710.05941.

[23] D. Misra, *Mish: A self regularized non-monotonic activation function*, 2019. DOI: 10.48550/ARXIV.1908.08681. [Online]. Available: https://arxiv.org/abs/1908.08681.

[24] D. Hendrycks and K. Gimpel, *Gaussian error linear units (gelus)*, 2016. DOI: 10.48550/ARXIV.1606.08415. [Online]. Available: https://arxiv.org/abs/1606.08415.

[25] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever and R. Salakhutdinov, 'Dropout: A simple way to prevent neural networks from overfitting,' *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[26] D. Krueger, T. Maharaj, J. Kramár, M. Pezeshki, N. Ballas, N. R. Ke, A. Goyal, Y. Bengio, A. Courville and C. Pal, *Zoneout: Regularizing rnns by randomly preserving hidden activations*, 2016. DOI: 10.48550/ARXIV.1606.01305. [Online]. Available: https://arxiv.org/abs/1606.01305.

[27] S. Ruder, 'An overview of gradient descent optimization algorithms,' *arXiv preprint arXiv:1609.04747*, 2016.

[28] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2017. arXiv: 1412.6980 [cs.LG].

[29] D. E. Rumelhart, G. E. Hinton and R. J. Williams, 'Learning representations by back-propagating errors,' *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[30] J. Schmidhuber, 'Deep learning in neural networks: An overview,' *Neural Networks*, vol. 61, pp. 85–117, 2015, Published online 2014; based on TR arXiv:1404.7828 [cs.NE]. DOI: 10.1016/j.neunet.2014.09.003.

[31] S. Santurkar, D. Tsipras, A. Ilyas and A. Madry, *How does batch normalization help optimization?* 2018. DOI: 10.48550/ARXIV.1805.11604. [Online]. Available: https://arxiv.org/abs/1805.11604.

[32] S. Ioffe and C. Szegedy, *Batch normalization: Accelerating deep network training by reducing internal covariate shift*, 2015. DOI: 10.48550/ARXIV.1502.03167. [Online]. Available: https://arxiv.org/abs/1502.03167.

[33] J. L. Ba, J. R. Kiros and G. E. Hinton, *Layer normalization*, 2016. DOI: 10.48550/ARXIV.1607.06450. [Online]. Available: https://arxiv.org/abs/1607.06450.

[34] J. Xu, X. Sun, Z. Zhang, G. Zhao and J. Lin, *Understanding and improving layer normalization*, 2019. DOI: 10.48550/ARXIV.1911.07013. [Online]. Available: https://arxiv.org/abs/1911.07013.

[35] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville and Y. Bengio, *Generative adversarial networks*, 2014. arXiv: 1406.2661 [stat.ML].

[36] M. Arjovsky, S. Chintala and L. Bottou, *Wasserstein gan*, 2017. arXiv: 1701.07875 [stat.ML].

[37] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin and A. Courville, *Improved training of wasserstein gans*, 2017. arXiv: 1704.00028 [cs.LG].

[38] T. Chen and C. Guestrin, 'Xgboost,' *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 785–794, Aug. 2016. DOI: 10.1145/2939672.2939785. [Online]. Available: http://dx.doi.org/10.1145/2939672.2939785.

[39] L. Rokach and O. Maimon, 'Decision trees,' in Jan. 2005, vol. 6, pp. 165–192. DOI: 10.1007/0-387-25465-X_9.

[40] L. Breiman, 'Bagging predictors,' *Machine learning*, vol. 24, no. 2, pp. 123–140, 1996.

[41] L. Breiman, 'Random forests,' *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[42] Y. Freund and R. E. Schapire, 'A decision-theoretic generalization of on-line learning and an application to boosting,' *Journal of computer and system sciences*, vol. 55, no. 1, pp. 119–139, 1997.

[43] R. E. Schapire, 'A brief introduction to boosting,' in *Ijcai*, Citeseer, vol. 99, 1999, pp. 1401–1406.

[44] J. H. Friedman, 'Greedy function approximation: A gradient boosting machine,' *Annals of statistics*, pp. 1189–1232, 2001.

[45] J. H. Friedman, 'Stochastic gradient boosting,' *Computational statistics & data analysis*, vol. 38, no. 4, pp. 367–378, 2002.

[46] M. Hossin and M. N. Sulaiman, 'A review on evaluation metrics for data classification evaluations,' *International journal of data mining & knowledge management process*, vol. 5, no. 2, p. 1, 2015.

[47] J. Davis and M. Goadrich, 'The relationship between precision-recall and roc curves,' in *Proceedings of the 23rd International Conference on Machine Learning*, ser. ICML '06, Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, 2006, pp. 233–240, ISBN: 1595933832. DOI: 10.1145/1143844.1143874. [Online]. Available: https://doi.org/10.1145/1143844.1143874.

[48] K. E. Emam, *Accelerating ai with synthetic data*. [Online]. Available: https://www.nvidia.com/en-us/deep-learning-ai/resources/accelerating-ai-with-synthetic-data-ebook/.

[49] S. I. Nikolenko *et al.*, *Synthetic data for deep learning*. Springer, 2021.

[50] J. Drechsler and J. P. Reiter, 'An empirical evaluation of easily implemented, nonparametric methods for generating synthetic datasets,' *Computational Statistics & Data Analysis*, vol. 55, no. 12, pp. 3232–3243, 2011.

[51] A. Dandekar, R. A. Zen and S. Bressan, 'A comparative study of synthetic dataset generation techniques,' in *International Conference on Database and Expert Systems Applications*, Springer, 2018, pp. 387–395.

[52] M. Mirza and S. Osindero, *Conditional generative adversarial nets*, 2014. arXiv: 1411.1784 [cs.LG].

[53] A. Martın, A. Ashish, B. Paul, B. Eugene, C. Zhifeng, C. Craig, C. Greg S., D. Andy, D. Jeffrey, D. Matthieu, G. Sanjay, G. Ian, H. Andrew, I. Geoffrey, I. Michael, J. Yangqing, J. Rafal, K. Lukasz, K. Manjunath, L. Josh, M. Dandelion, M. Rajat, M. Sherry, M. Derek, O. Chris, S. Mike, S. Jonathon, S. Benoit, S. Ilya, T. Kunal, T. Paul, V. Vincent, V. Vijay, V. Fernanda, V. Oriol, W. Pete, W. Martin, W. Martin, Y. Yuan and Z. Xiaoqiang, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015. [Online]. Available: https://www.tensorflow.org/.

[54] E. Jang, S. Gu and B. Poole, *Categorical reparameterization with gumbel-softmax*, 2017. arXiv: 1611.01144 [stat.ML].

[55] Z. Lin, A. Khetan, G. Fanti and S. Oh, *Pacgan: The power of two samples in generative adversarial networks*. DOI: 10.48550/ARXIV.1712.04086. [Online]. Available: https://arxiv.org/abs/1712.04086.

[56] G. Van Rossum and F. L. Drake, *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009, ISBN: 1441412697.

[57] J. V. Dillon, I. Langmore, D. Tran, E. Brevdo, S. Vasudevan, D. Moore, B. Patton, A. Alemi, M. Hoffman and R. A. Saurous, *Tensorflow distributions*, 2017. arXiv: 1711.10604 [cs.LG].

[58] The pandas development team, *Pandas-dev/pandas: Pandas*, version latest, Feb. 2020. DOI: 10.5281/zenodo.3509134. [Online]. Available: https://doi.org/10.5281/zenodo.3509134.

[59] W. McKinney, 'Data Structures for Statistical Computing in Python,' in *Proceedings of the 9th Python in Science Conference*, S. van der Walt and J. Millman, Eds., 2010, pp. 56–61. DOI: 10.25080/Majora-92bf1922-00a.

[60] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot and E. Duchesnay, 'Scikit-learn: Machine learning in Python,' *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[61] C. M. Bishop and N. M. Nasrabadi, *Pattern recognition and machine learning*, 4. Springer, 2006, vol. 4.

[62] B. Schweizer and E. F. Wolff, 'On nonparametric measures of dependence for random variables,' *The annals of statistics*, vol. 9, no. 4, pp. 879–885, 1981.

[63] K. Armanious, C. Jiang, M. Fischer, T. Küstner, T. Hepp, K. Nikolaou, S. Gatidis and B. Yang, 'MedGAN: Medical image translation using GANs,' *Computerized Medical Imaging and Graphics*, vol. 79, p. 101 684, Jan. 2020. DOI: 10.1016/j.compmedimag.2019.101684. [Online]. Available: https://doi.org/10.1016%2Fj.compmedimag.2019.101684.

[64] N. Park, M. Mohammadi, K. Gorde, S. Jajodia, H. Park and Y. Kim, 'Data synthesis based on generative adversarial networks,' *Proceedings of the VLDB Endowment*, vol. 11, no. 10, pp. 1071–1083, Jun. 2018. DOI: 10.14778/3231751.3231757. [Online]. Available: https://doi.org/10.14778%2F3231751.3231757.

[65] A. Srivastava, L. Valkov, C. Russell, M. U. Gutmann and C. Sutton, *Veegan: Reducing mode collapse in gans using implicit variational learning*, 2017. DOI: 10.48550/ARXIV.1705.07761. [Online]. Available: https://arxiv.org/abs/1705.07761.

[66] J. Zhang, G. Cormode, C. M. Procopiuc, D. Srivastava and X. Xiao, 'Privbayes: Private data release via bayesian networks,' *ACM Transactions on Database Systems (TODS)*, vol. 42, no. 4, pp. 1–41, 2017.

[67] C. J. Quinn, T. P. Coleman and N. Kiyavash, 'Approximating discrete probability distributions with causal dependence trees,' in *2010 International Symposium On Information Theory Its Applications*, 2010, pp. 100–105. DOI: 10.1109/ISITA.2010.5649470.

[68] T. M. Beasley, S. Erickson and D. B. Allison, 'Rank-based inverse normal transformations are increasingly used, but are they merited?' *Behavior genetics*, vol. 39, no. 5, pp. 580–595, Sep. 2009.

[69] K. Bogner, F. Pappenberger and H. Cloke, 'The normal quantile transformation and its application in a flood forecasting system,' *Hydrology and Earth System Sciences*, vol. 16, no. 4, pp. 1085–1094, 2012.

[70] H. L. Harter, 'Expected values of normal order statistics,' *Biometrika*, vol. 48, no. 1/2, pp. 151–165, 1961, ISSN: 00063444. [Online]. Available: http://www.jstor.org/stable/2333139 (visited on 20/05/2022).

[71] G. Blom, 'Statistical estimates and transformed beta-variables,' Ph.D. dissertation, Almqvist & Wiksell, 1958.

[72] S. Verma, J. Dickerson and K. Hines, *Counterfactual explanations for machine learning: A review*, 2020. DOI: 10.48550/ARXIV.2010.10596. [Online]. Available: https://arxiv.org/abs/2010.10596.

[73] B. Kment, 'Counterfactuals and causal reasoning,' *Perspectives on Causation*, pp. 463–482, 2020.

[74] C. Fernández-Loría, F. Provost and X. Han, *Explaining data-driven decisions made by ai systems: The counterfactual approach*, 2020. DOI: 10.48550/ARXIV.2001.07417. [Online]. Available: https://arxiv.org/abs/2001.07417.

[75] S. Verma, J. Dickerson and K. Hines, *Counterfactual explanations for machine learning: Challenges revisited*, 2021. DOI: 10.48550/ARXIV.2106.07756. [Online]. Available: https://arxiv.org/abs/2106.07756.

[76] Y. Censor, 'Pareto optimality in multiobjective problems,' *Applied Mathematics and Optimization*, vol. 4, no. 1, pp. 41–59, 1977.

[77] A. Redelmeier, M. Jullum, K. Aas and A. Løland, *Mcce: Monte carlo sampling of realistic counterfactual explanations*, 2021. arXiv: 2111.09790 [stat.ML].

[78] D. Nemirovsky, N. Thiebaut, Y. Xu and A. Gupta, 'Countergan: Generating realistic counterfactuals with residual generative adversarial nets,' *arXiv preprint arXiv:2009.05199*, 2020.

[79] D. Dua and C. Graff, *UCI machine learning repository*, 2017. [Online]. Available: http://archive.ics.uci.edu/ml.

[80] K. Fernandes, P. Vinagre and P. Cortez, 'A proactive intelligent decision support system for predicting the popularity of online news,' in *Portuguese conference on artificial intelligence*, Springer, 2015, pp. 535–546.

[81] M. Själander, M. Jahre, G. Tufte and N. Reissmann, *EPIC: An energy-efficient, high-performance GPGPU computing research infrastructure*, 2019. arXiv: 1912.05848 [cs.DC].

[82] T. S. Sheikh, A. Khan, M. Fahim and M. Ahmad, 'Synthesizing data using variational autoencoders for handling class imbalanced deep learning,' in *International Conference on Analysis of Images, Social Networks and Texts*, Springer, 2019, pp. 270–281.

[83] M. N. Garofalakis, 'Differentially private data synthesis using variational autoencoders,' 2021.

[84] N. Cristianini and E. Ricci, 'Support vector machines,' in *Encyclopedia of Algorithms*, M.-Y. Kao, Ed. Boston, MA: Springer US, 2008, pp. 928–932, ISBN: 978-0-387-30162-4. DOI: 10.1007/978-0-387-30162-4_415. [Online]. Available: https://doi.org/10.1007/978-0-387-30162-4_415.

[85] B. Scholkopf, K.-K. Sung, C. Burges, F. Girosi, P. Niyogi, T. Poggio and V. Vapnik, 'Comparing support vector machines with gaussian kernels to radial basis function classifiers,' *IEEE Transactions on Signal Processing*, vol. 45, no. 11, pp. 2758–2765, 1997. DOI: 10.1109/78.650102.

[86] I. Tolstikhin, S. Gelly, O. Bousquet, C.-J. Simon-Gabriel and B. Schölkopf, *Adagan: Boosting generative models*, 2017. DOI: 10.48550/ARXIV.1701.02386. [Online]. Available: https://arxiv.org/abs/1701.02386.

[87] L. Deng, 'The mnist database of handwritten digit images for machine learning research,' *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[88] A. Nitanda and T. Suzuki, 'Gradient layer: Enhancing the convergence of adversarial training for generative models,' in *International Conference on Artificial Intelligence and Statistics*, PMLR, 2018, pp. 1008–1016.

Appendix A

Code Repository

For the interested reader the code used in this thesis can be found in the Github repository: `https://github.com/ArneRustad/Master-thesis-cf`.

Appendix B

Uniform order statistics

For n independent, identically distributed random variables $X_1, \ldots, X_n$ with probability distribution function f_X and cumulative distribution function F_X, the cumulative distribution of the kth order statistic, $X_{(k)}$, is

$$F_{X_{(k)}}(x) = \sum_{j=k}^{n} \binom{n}{k} [F_X(x)]^j \, [1 - F_X(x)]^{n-j} \, .$$

From this, the corresponding probability density function can be derived as

$$f_{X_{(k)}}(x) = \frac{n!}{(k-1)!(n-k)!} f_X(x) \, [F_X(x)]^{k-1} \, [1 - F_X(x)]^{n-k} \, .$$

Inserting for the Uniform$(0, 1)$ distribution we get

$$\begin{aligned} f_{X_{(k)}}(x) &= \frac{n!}{(k-1)!(n-k)!} \cdot 1 \cdot x^{k-1}(1-x)^{n-k} \\ &= \frac{\Gamma(n+1)}{\Gamma(k)\Gamma(n-k+1)} x^{k-1}(1-x)^{n-k} \\ &= \frac{1}{B(k, n-k+1)} x^{k-1}(1-x)^{n-k}, \quad \text{for } 0 \leq x \leq 1. \end{aligned}$$

We recognize this as the probability distribution function of a Beta$(k, n-k+1)$ distribution. Thus, the expected value of the kth order statistic for the uniform distribution given n samples is

$$\mathrm{E}[X_{(k)}] = \frac{k}{n+1}.$$

Appendix C

Additional Histograms from Comparison of Marginal Distributions on Adult Dataset

To not overwhelm the reader, we decided to move some of the plots referenced in the comparison in Section 7.3 to this appendix chapter.

Figure C.1: Histograms of generated samples from **ctabGAN-sd** plotted against histograms of observations from the Adult training dataset. One histogram for each column. The synthesized dataset is of the same size as the Adult train dataset.

Figure C.2: Histograms of generated samples from **ctabGAN-qt** plotted against histograms of observations from the Adult training dataset. One histogram for each column. The synthesized dataset is of the same size as the Adult train dataset.

Figure C.3: Histograms of generated samples from **CTGAN-pac10** plotted against histograms of observations from the Adult training dataset. One histogram for each column. The synthesized dataset is of the same size as the Adult train dataset.

Figure C.4: Histograms of generated samples from **CopulaGAN** plotted against histograms of observations from the Adult training dataset. One histogram for each column. The synthesized dataset is of the same size as the Adult train dataset.

Figure C.5: Histograms of generated samples from **TVAE-mod** plotted against histograms of observations from the Adult training dataset. One histogram for each column. The synthesized dataset is of the same size as the Adult train dataset.

Appendix D

Preprocessing Steps of Adult Dataset

After removing observations with missing values from the unedited Adult dataset, the numerical ranges or categories in addition to percentages are given in Table D.1. Each of the following preprocessing steps were performed on the Adult dataset discussed in Section 6.2.

1. The column "relationship" was deleted.
2. For the column "native_country" all countries except for the *US* was merged to one single category called *Non-US*.
3. For the column "marital_status" the categories *Married-AF-spouse*, *Married-civ-spouse*, *Married-spouse-absent* were all merged to a single category *Married*.
4. For the column "workclass" the categories *Local-gov*, *State-gov* and *Federal-gov* were merged to a single category *Government*.
5. For the column "education" the categories *10th*, *11th*, *12th*, *1st-4th*, *5th-6th*, *7th-8th*, *9th*, *Preschool* were all merged to the single category *<=12th*.
6. Change the name of the category *Amer-Indian-Eskimo* of the "race" column to *Native-American-Inuit*

Column	Values
age	[17, 90]
workclass	Federal-gov (3.1%), Local-gov (6.9%), Private (74%), Self-emp-inc (3.6%), Self-emp-not-inc (8.4%), State-gov (4.3%), Without-pay (0.046%)
fnlwgt	[13492, 1490400]
education	10th (2.7%), 11th (3.6%), 12th (1.3%), 1st-4th (0.49%), 5th-6th (0.99%), 7th-8th (1.8%), 9th (1.5%), Assoc-acdm (3.3%), Assoc-voc (4.3%), Bachelors (17%), Doctorate (1.2%), HS-grad (33%), Masters (5.6%), Preschool (0.16%), Prof-school (1.7%), Some-college (22%)
educational_num	[1, 16]
marital_status	Divorced (14%), Married-AF-spouse (0.071%), Married-civ-spouse (47%), Married-spouse-absent (1.2%), Never-married (32%), Separated (3.1%), Widowed (2.8%)
occupation	Adm-clerical (12%), Armed-Forces (0.031%), Craft-repair (13%), Exec-managerial (13%), Farming-fishing (3.3%), Handlers-cleaners (4.5%), Machine-op-inspct (6.6%), Other-service (11%), Priv-house-serv (0.51%), Prof-specialty (13%), Protective-serv (2.2%), Sales (12%), Tech-support (3.1%), Transport-moving (5.1%)
relationship	Husband (41%), Not-in-family (26%), Other-relative (3%), Own-child (15%), Unmarried (11%), Wife (4.6%)
race	Amer-Indian-Eskimo (0.96%), Asian-Pac-Islander (2.9%), Black (9.3%), Other (0.78%), White (86%)
gender	Female (32%), Male (68%)
capital_gain	[0, 99999]
capital_loss	[0, 4356]
hours_per_week	[1, 99]
native_country	Cambodia (0.057%), Canada (0.36%), China (0.25%), Columbia (0.18%), Cuba (0.29%), Dominican-Republic (0.21%), Ecuador (0.095%), El-Salvador (0.33%), England (0.26%), France (0.08%), Germany (0.43%), Greece (0.11%), Guatemala (0.19%), Haiti (0.15%), Holand-Netherlands (0.0022%), Honduras (0.042%), Hong (0.062%), Hungary (0.04%), India (0.33%), Iran (0.12%), Ireland (0.08%), Italy (0.22%), Jamaica (0.23%), Japan (0.2%), Laos (0.046%), Mexico (2%), Nicaragua (0.11%), Outlying-US(Guam-USVI-etc) (0.049%), Peru (0.1%), Philippines (0.63%), Poland (0.18%), Portugal (0.14%), Puerto-Rico (0.39%), Scotland (0.044%), South (0.22%), Taiwan (0.12%), Thailand (0.064%), Trinadad&Tobago (0.057%), United-States (91%), Vietnam (0.18%), Yugoslavia (0.051%)
income	<=50K (75%), >50K (25%)

Table D.1: Overview of the numerical range or different categories for each column in the Adult dataset introduced in Section 6.2.

Appendix E

Additional Information for Datasets

In Section 6.3 we briefly present three real world dataset. In this chapter in the appendix we present some tables with additional column information for each dataset.

E.1 Covertype Dataset

We include here two tables that provide additional information to the description of the Covertype dataset given in Section 6.3.1. Table E.1 gives a short description of each column, while Table E.2 gives a short overview of column categories or numerical ranges.

Table E.1: Description of each column in the Covertype dataset after the preprocessing described in Section 6.3.1. The information is taken from the Covertype dataset page in the UCI Machine Learning Repository[79].

Column	Data Type	Description
Elevation	Quantitative	Elevation in meters
Aspect	Quantitative	Aspect in degrees azimuth
Slope	Quantitative	Slope in degrees
Horizontal_Distance_To_Hydrology	Quantitative	Horizontal distance to nearest surface water features
Vertical_Distance_To_Hydrology	Quantitative	Vertical distance to nearest surface water features
Horizontal_Distance_To_Roadways	Quantitative	Horizontal distance to nearest roadway
Hillshade_9am	Quantitative	Hillshade index at 9am (summer solstice)
Hillshade_Noon	Quantitative	Hillshade index at noon (summer soltice)
Hillshade_3pm	Quantitative	Hillshade index at 3pm (summer solstice)
Horizontal_Distance_To_Fire_Points	Quantitative	Horizontal distance to nearest wildfire ignition points
Wilderness_Area	Qualitative	Wilderness area designation
Soil_Type	Qualitative	Designated soil type
Cover_Type	Qualitative	Designated forest cover type

Table E.2: Overview of each column in the Covertype dataset. For numerical columns the value range is reported, while for discrete columns the categories along with percentages are reported. The percentages for all categories of a single discrete column sum to 1.

Column	Values (numerical range or categories)
Aspect	$[0, 360]$
Cover_Type	CoverType1 (40.0%), CoverType2 (50.0%), CoverType3 (6.0%), CoverType4 (0.5%), CoverType5 (2.0%), CoverType6 (3.0%), CoverType7 (4.0%)
Elevation	$[1859, 3858]$
Hillshade_3pm	$[0, 254]$
Hillshade_9am	$[0, 254]$
Hillshade_Noon	$[0, 254]$
Horizontal_Distance_To_Fire_Points	$[0, 7173]$
Horizontal_Distance_To_Hydrology	$[0, 1397]$
Horizontal_Distance_To_Roadways	$[0, 7117]$
Slope	$[0, 66]$
Soil_Type	Soil_Type1 (0.5%), Soil_Type10 (6.0%), Soil_Type11 (2.0%), Soil_Type12 (5.0%), Soil_Type13 (3.0%), Soil_Type14 (0.1%), Soil_Type15 (0.0005%), Soil_Type16 (0.5%), Soil_Type17 (0.6%), Soil_Type18 (0.3%), Soil_Type19 (0.7%), Soil_Type2 (1.0%), Soil_Type20 (2.0%), Soil_Type21 (0.1%), Soil_Type22 (6.0%), Soil_Type23 (10.0%), Soil_Type24 (4.0%), Soil_Type25 (0.08%), Soil_Type26 (0.4%), Soil_Type27 (0.2%), Soil_Type28 (0.2%), Soil_Type29 (20.0%), Soil_Type3 (0.8%), Soil_Type30 (5.0%), Soil_Type31 (4.0%), Soil_Type32 (9.0%), Soil_Type33 (8.0%), Soil_Type34 (0.3%), Soil_Type35 (0.3%), Soil_Type36 (0.02%), Soil_Type37 (0.05%), Soil_Type38 (3.0%), Soil_Type39 (2.0%), Soil_Type4 (2.0%), Soil_Type40 (2.0%), Soil_Type5 (0.3%), Soil_Type6 (1.0%), Soil_Type7 (0.02%), Soil_Type8 (0.03%), Soil_Type9 (0.2%)
Vertical_Distance_To_Hydrology	$[-173, 601]$
Wilderness_Area	Wilderness_Area1 (40%), Wilderness_Area2 (5%), Wilderness_Area3 (40%), Wilderness_Area4 (6%)

E.2 Credit Card Fraud Dataset

We include here a table that provides additional information to the description of the Credit Card Fraud dataset given in Section 6.3.1. Table E.2 gives a short overview of categories or numerical range for each column in the dataset.

Table E.3: Overview of each column in the Creditcard dataset. For numerical columns the value range is reported, while for discrete columns the categories along with percentages are reported. The percentages for all categories of a single discrete column sum to 1.

Column	Values (numerical range or categories)
Amount	[0.0, 25691.16]
Class	Class0 (99.83%), Class1 (0.17%)
Time	[0.0, 172792.0]
V1	[-56.41, 2.45]
V10	[-24.59, 23.75]
V11	[-4.8, 12.02]
V12	[-18.68, 7.85]
V13	[-5.79, 7.13]
V14	[-19.21, 10.53]
V15	[-4.5, 8.88]
V16	[-14.13, 17.32]
V17	[-25.16, 9.25]
V18	[-9.5, 5.04]
V19	[-7.21, 5.59]
V2	[-72.72, 22.06]
V20	[-54.5, 39.42]
V21	[-34.83, 27.2]
V22	[-10.93, 10.5]
V23	[-44.81, 22.53]
V24	[-2.84, 4.58]
V25	[-10.3, 7.52]
V26	[-2.6, 3.52]
V27	[-22.57, 31.61]
V28	[-15.43, 33.85]
V3	[-48.33, 9.38]
V4	[-5.68, 16.88]
V5	[-113.74, 34.8]
V6	[-26.16, 73.3]
V7	[-43.56, 120.59]
V8	[-73.22, 20.01]
V9	[-13.43, 15.59]

E.3 Online News Popularity Dataset

We include here a table that provides additional information to the description of the Online News Popularity dataset given in Section 6.3.3. Table E.4 gives a short overview of categories or numerical range for each column in the dataset.

Table E.4: Overview of each column in the Online News Popularity dataset. For numerical columns the value range is reported, while for discrete columns the categories along with percentages are reported. The percentages for all categories of a single discrete column sum to 1.

Column	Values (numerical range or categories)
abs_title_sentiment_polarity	[0.0, 1.0]
abs_title_subjectivity	[0.0, 0.5]
average_token_length	[0.0, 8.04]
avg_negative_polarity	[-1.0, 0.0]
avg_positive_polarity	[0.0, 1.0]
data_channel	bus (20%), entertainment (20%), lifestyle (20%), socmed (6%), tech (20%), world (20%)
global_rate_negative_words	[0.0, 0.18]
global_rate_positive_words	[0.0, 0.16]
global_sentiment_polarity	[-0.39, 0.73]
global_subjectivity	[0.0, 1.0]
is_weekend	no (90%), yes (10%)
kw_avg_avg	[0.0, 43567.66]
kw_avg_max	[0.0, 843300.0]
kw_avg_min	[-1.0, 42827.86]
kw_max_avg	[0.0, 298400.0]
kw_max_max	[0.0, 843300.0]
kw_max_min	[0.0, 298400.0]
kw_min_avg	[-1.0, 3613.04]
kw_min_max	[0.0, 843300.0]
kw_min_min	[-1.0, 377.0]
LDA_00	[0.0, 0.93]
LDA_01	[0.0, 0.93]
LDA_02	[0.0, 0.92]
LDA_03	[0.0, 0.93]
LDA_04	[0.0, 0.93]
max_negative_polarity	[-1.0, 0.0]
max_positive_polarity	[0.0, 1.0]
min_negative_polarity	[-1.0, 0.0]
min_positive_polarity	[0.0, 1.0]
n_non_stop_unique_tokens	[0.0, 650.0]
n_non_stop_words	[0.0, 1042.0]
n_tokens_content	[0.0, 8474.0]
n_tokens_title	[2.0, 23.0]
n_unique_tokens	[0.0, 701.0]
num_hrefs	[0.0, 304.0]
num_imgs	[0.0, 128.0]
num_keywords	[1.0, 10.0]
num_self_hrefs	[0.0, 116.0]
num_videos	[0.0, 91.0]
rate_negative_words	[0.0, 1.0]
rate_positive_words	[0.0, 1.0]
self_reference_avg_sharess	[0.0, 843300.0]
self_reference_max_shares	[0.0, 843300.0]
self_reference_min_shares	[0.0, 843300.0]
shares	[1, 843300]
timedelta	[8.0, 731.0]
title_sentiment_polarity	[-1.0, 1.0]
title_subjectivity	[0.0, 1.0]
weekday	Friday (10%), Monday (20%), Saturday (6%), Sunday (7%), Thursday (20%), Tuesday (20%), Wednesday (20%)

Arne Rustad

tabGAN: A Framework for Utilizing Tabular GAN for Data Synthesizing and Generation of Counterfactual

NTNU
Norwegian University of
Science and Technology