A synthetic data generating approach and synthetic data mechanisms for regularization

Jin Xu

Master of Science
Artificial Intelligence
School of Informatics
University of Edinburgh
2017

Abstract

This project explores how to improve predictive performance of machine learning system using synthetic data when training data is insufficient. We investigate different ways to generate synthetic data, and how to train methods on them. In particular, we propose an approach to generate synthetic data by only modeling the conditional of one input feature given all the others. We show that it is possible to sample from the model totally upon these conditionals.

By modeling conditionals separately, we obtain a flexible framework, that many methods could be brought in as conditional distribution models. We try GB (Gradient Boosting), RF (Random Forest), neural network (and also a parameters sharing architecture) and SVM (Support Vector Machine) as conditional distribution models for categorical features. Afterward, we extend our exploration to real-valued cases by introducing MDN (Mixture Density Network) and GAN (Generative Adversarial Network) as conditional distribution models for real-valued features.

Furthermore, as a main application of our approach, we apply synthetic data regularization to state-of-the-art methods in various classification and regression tasks. Performance improvements are observed on all datasets, even though the original methods are already high-performance ones. We show that our approach is very successful on categorical datasets, and clearly outperform joint distribution models like RBM (Restricted Boltzmann Machine) in terms of regularization. The extension to real-valued datasets is not as successful as expected. Currently, we still lose to GMM (Gaussian Mixture Model) on real-valued datasets.

Finally, we explore and discuss the effects of possible hyper-parameters and model options for our approach, along with some empirical techniques that could be useful in practice. Lots of experiments are conducted to verify our hypotheses, and a list of recommendations are provided in the end.

In addition, we also show that our synthetic data generating approach can also be useful in knowledge distilling, and give an example of compressing the knowledge from an ensemble into a single model.

Acknowledgements

First of all, I would like to express my sincere gratitude to my supervisor Dr. Iain Murray for his expertise, patience, and encouragement. His guidance helped me in all the time of this project. Without his insightful comments and detailed feedback, this work is not possible. The discussions with Iain greatly widen my horizon in various fields, and I feel so fortunate and privileged to have been supervised by him.

I would also like to thank all my lecturers and tutors, for teaching me so much and helping me with any questions I have. It has been such a fruitful and pleasant year for me, and I really appreciate all your hard work.

Thanks to all my friends I met here. It is a great pleasure to know all of you, and I wish you all have a brilliant future.

Last but not least, an immense gratitude to my parents, for being supportive throughout my life. They always give me the freedom to do what I want to do, and always firmly stand behind me. Their unconditional love gives me the strength to go through any hard time.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Jin Xu*)

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In a typical machine learning problem, we train our model on the training data, trying to capture relationships between variables. Then the model is used to give predictions on new unseen data. Sometimes, the amount of training data is very limited and it may be expensive or hard to collect more data. Our model could just memorize all the training data including noise, instead of identifying the meaningful trends in it. In that case, the model cannot have good predictive performance on new data. The phenomenon is often called overfitting (see 2.1.3 for details), and it is a big obstacle to improve the predictive performance of a machine learning system.

In order to overcome the overfitting problem, we often use techniques to control the complexity of our model. With limited expressive power, our model is forced to capture significant relationships between variables, rather than random noise. In section 2.6, we will discuss various techniques that control the model complexity by modifying the training procedure, we often call them regularization. As an effective and flexible regularization technique, feeding synthetic data together with training data to our machine learning system could potentially contribute considerable performance improvement. However, the performance heavily depends on how we generate these synthetic data.

Existing approaches for synthetic data generation mainly fall into two categories. The most straightforward one is to model the joint distribution $P(\boldsymbol{x})$ over all variables $\boldsymbol{x}$ of the data, and then sample directly from the model. The other way is more heuristic, often involving perturbations or transformations on the training data. For example, adding noise to the training data[35], applying transformations to the data[25] (mainly for image data or sequence data), or swapping attributes between neighbouring samples like MUNGE[6].

Our approach is landing on the middle ground. We adopt a model called the dependency network [19] (see section 3.1), which is actually a collection of conditional distribution models $P(x_m \mid \boldsymbol{x}_{\setminus m})$ (the conditional of one feature given all the others). So if the dataset has M attributes, we will have a collection of M conditional distribution models. we do not require these conditional distribution models to be consistent with each other. By relaxing the consistency constraint, theoretically, any machine learning method that is able to model $P(x_m \mid \boldsymbol{x}_{\setminus m})$ can be introduced into our framework. To generate synthetic data, we could perturb the training data by resampling attributes from corresponding conditionals one-by-one and record changes along the way. The sampling procedure is actually called Gibbs sampling (see 2.2). It has been proven that, if we run the procedure for enough steps, the original training data will be forgotten and an equilibrium distribution could be reached.

Our approach is interesting to explore because it potentially has the following advantages:

- Modeling conditional distributions $P(x_m \mid \boldsymbol{x}_{\setminus m})$ is considered to be easier than modeling the joint distribution $P(\boldsymbol{x})$. So it is reasonable to expect our model outperforming some joint distribution models.

- Many existing state-of-the-art machine learning methods can be used as one-dimensional conditional distribution models, and most implementations directly support doing that. So this work could greatly expand our model choices.

- If we generate synthetic data by applying perturbations or transformations, the synthetic data will be inevitably similar to the training data. The synthetic data only carry limited information except for some prior knowledge. In our approach, attributes are perturbed by Gibbs sampling. As we discussed above, if we wait for enough time, the original training data will be forgotten, and the synthetic data could be quite informative.

Synthetic data can also be useful in other ways. For example, sometimes, people want to compress the knowledge of a large expensive model into a small, cheap model, which is more efficient in terms of computing time and storage space. The task is often called knowledge distilling (also known as model compression, or teacher-student model) [6][21]. When we conduct knowledge distilling, we often need additional samples as ”teaching examples” in addition to the training data, and here is where synthetic data come into play. In this project, we will use a simple example to demonstrate that our synthetic data generating approach can also perform fairly well in knowledge distilling.

1.2 Objective

This project aims to develop a synthetic data generating approach based on the dependency network model, and compare it to other existing approaches. We intend to explore all kinds of candidates for conditional distribution models, and compare them with each other. Furthermore, we want to focus on the usage of synthetic data in regularization when limited training data are given, and verify whether performance improvement can be observed in various datasets. As a new approach, there will be many tunable parameters or options, we want to explore and understand the effects of them through experiments and all kinds of measures.

1.3 Contributions

The contributions of this work can be summarized as follows:

- Based on the dependency network model, a synthetic data generating approach is developed. By giving up the consistency constraints of conditionals, we introduce a flexible framework that can bring in many state-of-the-art methods that we used to believe are only suitable for classification or regression tasks.
- We explore the possibility of using random forest, gradient boosting, neural network and support vector machine as the conditional distribution model for categorical features, and the possibility of using mixture density network and generative adversarial network as the conditional distribution model for real-valued features. We evaluate their performance by various measures on datasets.
- By applying synthetic data regularization to other methods, which can be seen as an application of our data generating approach, we achieve performance improvement on all datasets with limited training data (considerable improvement on categorical datasets, and slight improvement on real-valued datasets). Even though the original methods we are comparing to are already state-of-the-art methods. Experiments are conducted on various datasets, including different tasks and feature types.

- We introduce lots of empirical techniques for synthetic data regularization that can potentially be useful under certain circumstances. Lots of experiments are conducted to verify our hypotheses. Moreover, a useful empirical guideline is provided.

- We show that our approach can also be used in knowledge distilling, and back up this argument by compressing an ensemble of models into a single model using synthetic data generated by our approach.

1.4 Overview

In chapter 2, we give a background review of existing synthetic data generating approaches, including joint distribution models, and other more pragmatic approaches. We will also give a more detailed explanation of regularization and why synthetic data can be used for that purpose. Our approach will be presented in chapter 3, including all kinds of model choices for conditional distributions, the sampling method, and some modeling and sampling options associated with our approach. A theoretical comparison to other approaches is at the end of that chapter. The following chapters are all empirical results. In chapter 4, we show that our synthetic data regularization method can greatly improve performance on categorical datasets. We also extend our conclusions to real-valued cases. Different conditional distribution models for categorical features and real-valued features are compared to their counterparts based on experiment results. We also compare our dependency network model to other generative models in the end. The chapter 5 is mainly about empirical techniques that could be useful under certain circumstances. We describe and discuss experiments that we used to verify our hypotheses. A handy empirical guideline is presented in the last section. Chapter 6 is used to discuss some interesting topics that are not closely related to the central claims, but still contains interesting findings. Finally, In chapter 7, we draw overall conclusions. Directions for future work are also proposed.

Chapter 2

Background

The chapter reviews background knowledge to help with reading the later chapters. Most contents are definitions or standard results. Those who are familiar with machine learning field can safely skip materials here. At first, we introduce some basic concepts in machine learning. Then we present a family of sampling methods called Markov chain Monte Carlo, which is especially suitable for sampling from a distribution over high-dimensional space. After that, we introduce two generative models which will be compared to our approach through experiments. Then, an overview of synthetic data generating approaches is provided. In the end, we talk about a technique called regularization, which can be used to overcome the so-called overfitting problem (see 2.1.3), and also why we can use synthetic data in regularization.

2.1 Preliminary

2.1.1 Classification task and regression task

Classification and regression are two typical tasks in machine learning. Generally, classification identifies group membership of data points, whereas regression estimates relationships between variables. (It is often about predicting one dependent variable given several input features.) Both of them can be described as predicting label y given input features $\boldsymbol{x}$, or modeling $P(y \mid \boldsymbol{x})$ if in a probabilistic viewpoint. Although in classification, y is a categorical variable indicating which class it belongs, whereas, in regression, y is often a continuous variable.

The main differences between classification and regression tasks appear when we try to define the error function, which measures the discrepancy between our predictions and the ground truth. Also, the error function is often the optimization objective for training. In a C class classification task, our model predicts that, given input features $\boldsymbol{x}$, $P(c \mid \boldsymbol{x})$ is the probability that the sample belongs to class c. Then we can define the error function as

$$err(\boldsymbol{x}, y) = \sum_{c=1}^{C} -\log P(y = c \mid \boldsymbol{x}), \tag{2.1}$$

which is often called **cross-entropy error**. In a regression task, the label y is a continuous variable, and a commonly used error function is **mean squared error**. When the prediction is $f(\boldsymbol{x})$, the error function is defined as

$$err(\boldsymbol{x}, y) = (f(\boldsymbol{x}) - y)^2 \tag{2.2}$$

It is worthwhile to mention that, in a probabilistic aspect, the two error function above can be unified. You may notice that the cross-entropy error is simply the NLL (Negative Log Likelihood) of the label under the predicted distribution. If we convert our prediction $f(\boldsymbol{x})$ to a probabilistic one $\mathcal{N}(f(\boldsymbol{x}), \varepsilon^2)$ (assuming Gaussian noise), then the NLL (Negative Log Likelihood) of label is

$$-\log \mathcal{N}(y; f(\boldsymbol{x}), \varepsilon^2) \propto (f(\boldsymbol{x}) - y)^2, \tag{2.3}$$

which is proportional to the original error function.

2.1.2 Generative model and discriminative model

Roughly speaking, given input variables $\boldsymbol{x}$ and output label y, a generative model is a model of $P(\boldsymbol{x}, y)$, and a discriminative model is model of $P(y \mid \boldsymbol{x})$. The definitions are over-simplified, because not all discriminative models have probabilistic meanings.

If we only care about the prediction accuracy, discriminative models are sometimes regarded as a superior choice, because conditional distribution is easier to model. However, a generative model does not necessarily lose to a discriminative model. As a rule of thumb, generative models will probably perform better than discriminative models when training data is extremely insufficient.

2.1.3 Overfitting and underfitting

In machine learning tasks, the model is fit to the training data. We say our model is **overfitting**, when the model describes random noise in the training data instead of the underlying relationship. Overfitting often happens when the model has excessive complexity. A typical sign of overfitting is that the model has a very low error on the training set, but a much higher error on a new dataset. An intuitive understanding of this phenomenon is that, the model performs very well on the training data simply because it has memorized all the training data, instead of actually learning the underlying relationship between variables.

Underfitting is the opposite of overfitting, it means the model does not have enough complexity to capture the underlying relationship.

Both overfitting and underfitting lead to poor generalization performance. It is worthwhile to mention that, there is no clear boundary between overfitting and underfitting, and there does not exist a perfect complexity that can make sure the model capture all the trends but do not model the random noise. In fact, to some extent, most models are overfitting and underfitting at the same time.

2.1.4 Cross validation

A parametric model (which can be described by a finite number of parameters) often has hyper-parameters that can control the complexity of the model. For instance, the degree in polynomial regression and the number of components in a Gaussian mixture model (see 2.4) are all hyper-parameters. We cannot choose hyper-parameters by minimizing error on the training set because we will end up infinitely increasing the complexity to fit all the training data. A simple model is often embedded in a complex model, so increasing complexity can always lead to lower error. But in that case, our model is simply overfitting the training data and cannot generalize well.

Usually, we pick our hyper-parameters on the validation dataset, which is different from the training set and the test set. The procedure is to train the model on the training data with different combinations of hyper-parameters. Then the error is computed on the validation set, to see if our model can generalize well. In the end, the combination of hyper-parameters which leads to the best validation error is adopted. We call the procedure **cross validation**.

Using cross validation means we need to set aside a validation set to validate models. When we do not have sufficient data, this is a huge waste. Therefore, sometimes we use a cross validation method called **K-fold cross validation**, which does not need a separate validation set. The training set is randomly split into K parts with equal sizes. Each part will be used as the validation set once, and the training set consists of the rest $K-1$ parts. In the end, K validation errors are obtained, and we report the final K-fold validation error as the mean of these K validation errors.

2.2 Markov chain Monte Carlo and Gibbs sampling

MCMC (Markov Chain Monte Carlo) is a family of sampling methods, which can give correlated samples from the target distribution $P(\boldsymbol{x})$. It constructs a random walk to explore $P(\boldsymbol{x})$. MCMC is widely used to sample from distribution over high-dimensional variables. Other sampling methods such as rejection sampling and importance sampling scale badly with dimensionality[29].

To sample from the distribution $P(\boldsymbol{x})$, we construct a transition kernel $T(\boldsymbol{x}' \leftarrow \boldsymbol{x})$ for a Markov chain, to make sure the the equilibrium distribution of this Markov chain happens to be $P(\boldsymbol{x})$. In that case, regardless of the initialization, if we run the Markov chain for enough time, the states of this Markov chain can be regarded as samples from the equilibrium distribution. Recall that the equilibrium distribution is the same as the desired distribution $P(\boldsymbol{x})$, these samples are exactly what we want. We often call the reach of equilibrium distribution **mixing** and the period before mixing **burn in period**.

In this project, a popular MCMC method called Gibbs sampling will be used as the sampling method for our model. So we provide more details about it below:

To introduce this MCMC method, we only need to specify the transition kernel. Because a Markov chain is fully defined by the initial state and the transition kernel. Assuming $\boldsymbol{x}$ is a M dimensional random variable and $\boldsymbol{x} = (x_1, x_2, ..., x_M)$, we could compute conditional distributions $P(x_m \mid \boldsymbol{x}_{\setminus m})$ from the joint distribution $P(\boldsymbol{x})$. we could construct a transition kernel for each m:

$$T_m(\boldsymbol{x}' \leftarrow \boldsymbol{x}) = P(x'_m \mid \boldsymbol{x}_{\setminus m}), \tag{2.4}$$

where $\boldsymbol{x}' = (x'_m, \boldsymbol{x}_{\setminus m})$. Now let's decide a sampling order $o_1, o_2, ..., o_M$, which is a permutation of $1, 2, ..., M$. The transition kernel for Gibbs sampling can be defined as

$$T = T_{o_m}, \tag{2.5}$$

where m is the step.

2.3 Restricted Boltzmann machine

The RBM (Restricted Boltzmann Machine) is a generative model that can model the distribution over a set of binary inputs. It is a very good choice to model joint distribution if all the input features are binary, because of its ability to capture complex relationship between variables and the efficient training method called contrastive divergence [8]. (It is possible to extend the model to real-valued features [20], but we won't discuss it here.)

2.3.1 Model structure

The RBM has binary visible units $\boldsymbol{v} = (v_i)_i$ and hidden units $\boldsymbol{h} = (h_j)_j$, and a matrix of weights $W = (w_{ij})$ with each element w_{ij} corresponding to a connection between v_i and h_j. There will also be an offset a_i for each visible unit v_i and an offset b_j for hidden unit h_j. We define the energy function E of the RBM as

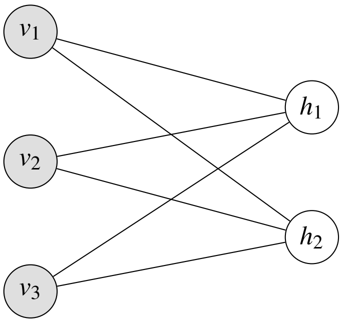
$$E(\boldsymbol{v},\boldsymbol{h}) = \sum_i \sum_j v_i w_{ij} h_j + \sum_i a_i v_i + \sum_j b_j h_j. \tag{2.6}$$

The energy function can be used to determine a distribution $P(\boldsymbol{v},\boldsymbol{h})$ over all visible and hidden units

$$P(\boldsymbol{v},\boldsymbol{h}) = \frac{1}{Z}\exp(E(\boldsymbol{x},\boldsymbol{h})), \tag{2.7}$$

where $Z = \int \exp(E(\boldsymbol{v},\boldsymbol{h}))d\boldsymbol{v}d\boldsymbol{h}$ is a partition function. Therefore, the RBM is a generative model parameterized by $\boldsymbol{a} = (a_i)_i, \boldsymbol{b} = (b_j)_j$ and W. Usually $\boldsymbol{v}$ corresponds to our observations , whereas $\boldsymbol{h}$ are hidden variables used to explain the observations.

Figure 2.1: Structure of the RBM



The structure of RBM is a bipartite graph. A symmetric connection exists between each hidden unit and each visible unit. There are no intra-layer connections within the hidden layer or the visible layer.

If we call all visible units the visible layer and all hidden units the hidden layer, RBM is a network which has no intra-layer connections. Because of this special structure, visible variables are mutually independent given all hidden variables. Similarly, hidden variables are mutually independent given all visible variables. Formally,

$$P(\boldsymbol{v} \mid \boldsymbol{h}) = \prod_i P(v_i \mid \boldsymbol{h}) \tag{2.8}$$

$$P(\boldsymbol{h} \mid \boldsymbol{v}) = \prod_j P(h_j \mid \boldsymbol{v}). \tag{2.9}$$

2.3.2 Sampling from RBM

We use Gibbs sampling to sample from RBM, and the special structure of RBM also leads to a more efficient sampling procedure. From equation 2.8, 2.9, it's not hard to conclude

$$P(v_i \mid \boldsymbol{v}_{\setminus i}, \boldsymbol{h}) = P(v_i \mid \boldsymbol{h}) \tag{2.10}$$

and

$$P(h_j \mid \boldsymbol{h}_{\setminus j}, \boldsymbol{v}) = P(h_j \mid \boldsymbol{v}). \tag{2.11}$$

Therefore, we can resample all visible variables or all hidden variables at the same time. One round of Gibbs sampling is simply

$$\boldsymbol{h}_t \sim P(\boldsymbol{h} \mid \boldsymbol{v}_t) \tag{2.12}$$

$$\boldsymbol{v}_{t+1} \sim P(\boldsymbol{v} \mid \boldsymbol{h}_t) \tag{2.13}$$

We often call it **block Gibbs sampling**, because we resample a block of variables at the same time.

2.3.3 Training algorithm: contrastive divergence

To be concise, we simplify our energy function as

$$E(\boldsymbol{v}, \boldsymbol{h}) = \boldsymbol{v}^T W \boldsymbol{h}. \tag{2.14}$$

The offset terms can be absorbed by adding new variables $v_0 = 1$ and $h_0 = 1$. So

$$E(\boldsymbol{v}, \boldsymbol{h}) = \sum_i \sum_j v_i w_{ij} h_j + \sum_i v_i a_i h_0 + \sum_j v_0 b_j h_j. \tag{2.15}$$

From equation 2.16, it is easy to have

$$\frac{\partial E}{\partial w_{ij}} = v_i h_j. \tag{2.16}$$

To train a RBM, our objective is to minimize the discrepancy between $P(\boldsymbol{v})$ and $P_{data}(\boldsymbol{v})$, here $P(\boldsymbol{v})$ is determined by RBM. Therefore, we want to minimize

$$J = < E >_\infty - < E >_0, \tag{2.17}$$

where $< \cdot >_k$ means the expectation with respect to distribution over visible variables when these variables are drawn from the kth block Gibbs sampling round. In practice, we won't run block Gibbs sampling for infinite rounds, so instead we minimize

$$J_k =< E >_k - < E >_0, \tag{2.18}$$

The gradient of J with respect to w_{ij} will be

$$\frac{\partial J}{\partial w_{ij}} =< v_i h_j >_k - < v_i h_j >_0 . \tag{2.19}$$

$< \cdot >_0$ can be estimated by sampling from the training data, whereas $< \cdot >_k$ can be estimated by running Gibbs sampling for k rounds starting from the training data. When $k = K$, we call the algorithm CD-K (CD stands for constrastive divergence). In practice, CD-1 works surprisingly well[20].

2.4 Gaussian mixture model

The GMM (Gaussian Mixture Model) is a generative model over real-valued variables. Regardless of its simple structure, it performs very well on some tasks. In our project, the GMM appears in three places:

1. As a joint distribution model, GMM is used to generate synthetic data and compared to our approach on real-valued datasets.
2. GMM is can be used as a component of mixture density network (see section 3.5).
3. GMM is used to initialize the mixture density network.

2.4.1 Gaussian mixture distribution

The Gaussian mixture distribution has the following density function:

$$P(x) = \sum_{c=1}^{C} \omega_c \mathcal{N}(x; \boldsymbol{\mu}_c, \Sigma_c), \tag{2.20}$$

where $\sum_{c=1}^{C} \omega_c = 1, \omega_c \geq 0$, and $\mathcal{N}$ is the density function for Gaussian distribution. It is easy to notice that the density function is simply a weighted sum of Gaussian density functions. We call each Gaussian distribution a component, and the component weights define a categorical distribution $Cat(\{\omega_c\}_c)$. A Gaussian mixture model describes data using a Gaussian mixture distribution.

2.4.2 Sampling from GMM

Sampling from a GMM takes 2 steps. Firstly, we sample from the categorical distribution $Cat(\{\omega_c\}_c)$ to determine which component the sample comes from. When the component is determined, the next step is simply a standard procedure to sample from a Gaussian distribution. Formally, it is

$$c \sim Cat(\{\omega_c\}_c) \tag{2.21}$$

$$\boldsymbol{x} \sim \mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}_c, \Sigma_c) \tag{2.22}$$

2.4.3 Training GMM: expectation-maximization algorithm

The EM (Expectation-Maximization) [13] algorithm is an iterative method, which iteratively runs the E step and the M step. The E step compute the expectation of all unobserved latent variables, given the observations and the current parameters. The M step updates parameters to maximize the likelihood, assuming the latent variables are not hidden and given by the last E step.

In the context of GMM, assuming $\{\boldsymbol{x}^{(n)}\}_{n=1,2,...,N}$ is the dataset, the latent variable is $\boldsymbol{z}^{(n)} = (z_1^{(n)}, z_2^{(n)}, ..., z_C^{(n)})$ where $z_c^{(n)}$ indicates whether the sample $\boldsymbol{x}^{(n)}$ comes from component c. So the training of GMM is just iteratively running the following two steps:

1. E step: fill in hidden variables $z_c^{(n)}$ with expectations, given the current parameters:

$$z_c^{(n)} = \frac{\omega_c \mathcal{N}(\boldsymbol{x}^{(n)}; \boldsymbol{\mu}_c, \Sigma_c)}{\sum\limits_{i=1}^{C} \omega_i \mathcal{N}(\boldsymbol{x}^{(n)}; \boldsymbol{\mu}_i, \Sigma_i)} \tag{2.23}$$

2. M step: Update all parameters $\{\omega_c, \boldsymbol{\mu}_c, \Sigma_c\}_{c=1,2,...C}$ to maximize the negative log likelihood:

$$-\sum_{n=1}^{N} \log \big[\sum_{c=1}^{C} \omega_c \mathcal{N}(\boldsymbol{x}^{(n)}; \boldsymbol{\mu}_c, \Sigma_c) \big] \tag{2.24}$$

So:

$$\omega_c = \sum_{n=1}^{N} z_c^{(n)} / N \tag{2.25}$$

$$\boldsymbol{\mu}_c = \sum_{n=1}^{N} z_c^{(n)} \boldsymbol{x}^{(n)} \Big/ \sum_{n=1}^{N} z_c^{(n)} \tag{2.26}$$

$$\Sigma_c = \sum_{n=1}^{N} z_c^{(n)} \boldsymbol{x}^{(n)} \boldsymbol{x}^{(n)T} \Big/ \sum_{n=1}^{N} z_c^{(n)} - \boldsymbol{\mu}_c \boldsymbol{\mu}_c^T \tag{2.27}$$

2.5 Synthetic data generating approaches

In this section, we review approaches for generating synthetic data. Roughly speaking, there are two mainstreams to obtain synthetic data. One is to construct a generative model by learning from the training data, and design a sampling procedure to get samples from the model. On the other hand, we could generate synthetic data simply by modifying training data, for example applying transformations based on prior knowledge, adding noise to the data, or perturbing attributes according to some rules.

2.5.1 Approaches based on generative models

Previously, we talk about RBM and GMM, which are all high-performance generative models which model the joint distributions over all variables. We describe the model structure, sampling methods together with training algorithms. These are all we need to design a synthetic data generating approach based on a generative model. Firstly, we train the generative model on the training data using the training algorithm. To generate synthetic data, we sample from the model using the corresponding sampling method.

There are of course other generative models. Theoretically, any generative model can be used as a synthetic data generating approach, if an efficient sampling method can be found. In this project, we will compare our approach to RBM on categorical datasets, and to GMM on real-valued datasets.

2.5.2 Modifying training data

Rather than modeling the training data using a generative model, we could apply transformation or perturbation to the training data. To deal with image data or sequence data, we are able to find invariant transformations [25][12]. For example, for an image, we could apply rotation, transition, up-down or left-right flip and so on. The transformations are based on the prior knowledge that our predictions should not change because of these transformations. However, for a general machine learning task, it is hard to find such invariant transformations. Alternatively, people could add random noise to the features, or set a random subset of features to 0 (for example denoising auto-encoder[39]).

Different from the above, a pragmatic approach to generate synthetic data called MUNGE is proposed in [6], which performs quite well in knowledge distilling tasks(see

6.2). Firstly, given the training data, we calculate pairwise distances. Here Euclidean distance is used for continuous variables, whereas hamming distance is used for categorical variables. Secondly, for each data point $\boldsymbol{x}$, we find its nearest neighbour $\boldsymbol{x}'$. After that, we swap attributes between neighbours $\boldsymbol{x}$ and $\boldsymbol{x}'$. For each attribute a, if a is categorical, we swap $\boldsymbol{x}[a]$ and $\boldsymbol{x}'[a]$ with probability p (here $\boldsymbol{x}[a]$ is the attribute a for $\boldsymbol{x}$). If a is continuous, with probability p, we resample $\boldsymbol{x}[a]$ and $\boldsymbol{x}'[a]$ from the following distribution:

$$\boldsymbol{x}[a] \sim \mathcal{N}(\boldsymbol{x}[a], \sigma^2) \tag{2.28}$$

$$\boldsymbol{x}'[a] \sim \mathcal{N}(\boldsymbol{x}'[a], \sigma^2), \tag{2.29}$$

where

$$\sigma = \frac{\text{distance}(\boldsymbol{x}[a], \boldsymbol{x}'[a])}{s} \tag{2.30}$$

p and s are considered as hyper-parameters of this method.

2.5.3 Discussion

In this part, we discuss the advantages, disadvantages, and limitations of all the synthetic data generating approaches mentioned above. The discussion is a useful guide of whether to use a particular approach under a certain circumstance, and some of the disadvantages and limitations become motivations of our approach, which will be presented in later chapters.

- Learning the joint distribution over a high-dimensional space is a hard task. Although methods like RBM and GMM are already high-performance, for some tasks we will still end up with a poor generative model. Sampling from such models will give us untrustworthy synthetic data.

- Generating synthetic data by modifying training data is easy to understand and implement. However, the synthetic data only carry our prior knowledge, rather than the relationships between variables. So compared to a generative model, these synthetic data give us very limited information.

- Invariant transformations are only possible for special kinds of data, so we cannot apply them to a general machine learning problem.

- Although modifying training data is easy to implement, it does not mean it is efficient. For example, in MUNGE we need to calculate pairwise distances, therefore, the computational cost is $O(MN^2)$, where M is the number of attributes, N

is the number of training samples. Because the cost is super-linear with respect to training set size, we cannot use it for large datasets.

2.6 Regularization

Regularization is a technique which can be used to overcome the overfitting (section 2.1.3) problem by controlling the complexity of the model. As mentioned before, parametric models often have hyper-parameters that can control the complexity of the model. However, hyper-parameters usually control the complexity by adjusting the structure of the model (for example the number of units or layers in a neural network). Regularization, on the other hand, often controls the complexity by modifying the training procedure. For example, adding a regularization term to the objective function, or terminating training procedure based on some evaluation (see early-stopping below).

2.6.1 L1 and L2 regularization

The most commonly used regularization technique is to add a loss penalty to the objective function. Let's say we have a model parameterized by θ. To train the model, we define a loss function $\mathcal{L}(\theta)$ on the parameters. Our training objective is to minimize the $\mathcal{L}(\theta)$. As a regularization, we could add a regularization term to the objective function, so the new function becomes:

$$\mathcal{L}'(\theta) = \mathcal{L}(\theta) + \lambda\Omega(\theta), \tag{2.31}$$

where $\Omega(\theta)$ is a penalty on the parameters θ, and λ controls the strength of this penalty.

We often want to control the magnitude of parameters, thus controls the complexity of the model. So if $\theta = (\theta_1, \theta_2, ..., \theta_K)$ we could let

$$\Omega(\theta) = \sum_{k=1}^{K} \theta_k^2. \tag{2.32}$$

This is called **L2 regularization**. We could also let

$$\Omega(\theta) = \sum_{k=1}^{K} |\theta_k|, \tag{2.33}$$

which is called **L1 regularization**.

L1 and L2 regularization are very efficient to compute and easy to use. They are widely used in all kinds of models.

2.6.2 Early-stopping

If you use cross validation (see section 2.1.4) to estimate hyper-parameters, usually the training error monotonically goes down during the training procedure, but after a certain point, the validation error stops going down and starts increasing. A possible explanation is that, at the beginning, our model updates parameters to capture the relationships between variables. However, as the training proceeds, the model continues to increase its complexity simply trying to model the noise in the training data. In that case, the model becomes overfitting, and cannot generalize well to the validation set. To prevent this phenomenon and control the model complexity, we could stop our training at the turning point of the validation error. We often call this technique **early-stopping**. Early-stopping is a very easy technique and can actually reduce training time because we won't run additional steps to model the noise.

2.6.3 Tangent propagation

In section 2.5.2, we talk about applying invariant transformations to training data, thus generate new synthetic data. Normally, people feed these synthetic data together with real training data into our model for the final task. (see the next section). **Tangent propagation** [36] is such an idea that, instead of really applying these transformations, we could directly encode the associated prior knowledge into our model.

To be specific, if we know an invariant transformation $T(\boldsymbol{x},\alpha)$, where $\boldsymbol{x}$ is the feature vector, and α is the transformation parameter. An example of α is the degree of rotation. Let's assume that the transformation is continuous with respect to the parameter, so

$$||T(\boldsymbol{x},\alpha)-\boldsymbol{x}|| \to 0 \tag{2.34}$$

when $\alpha \to 0$.

Assuming our model is a function of $\boldsymbol{x}$, and we denote it as $f(\boldsymbol{x})$. Because of invariance, we have

$$\frac{\partial f(T(\boldsymbol{x},\alpha))}{\partial \alpha}|_{\alpha=0} = 0 \tag{2.35}$$

If our training set is $D = \{\boldsymbol{x}_n\}_{n=1,2,\ldots,N}$, then we can encode our prior knowledge about this invariant transformation as a penalty term

$$\Omega(\theta) = \sum_{n=1}^{N} \frac{\partial f(T(\boldsymbol{x},\alpha))}{\partial \alpha}|_{\alpha=0} \tag{2.36}$$

Tangent propagation is often more efficient than directly feeding synthetic data. When feeding K times synthetic data, the computing time grows at least linearly with

K. But with tangent propagation, we only need to deal with an additional term in the objective function.

2.7 Feeding synthetic data as regularization

Feeding synthetic data together with training data into our model for training can be considered as another regularization technique. More training data is always beneficial, if these training data come from the real underlying distribution. If our synthetic data generating approach is good, it would be hard for us to discriminate these synthetic data from training data. Thus, feeding synthetic data would be like expanding the training set, even though the additional data are actually synthetic.

We talk about many approaches to generate synthetic data in this chapter, and we will introduce our synthetic data generating approach in the next chapter. Potentially, they can all be used for regularization.

However, there will always be a discrepancy between synthetic data and real data. Clearly, we do not want synthetic data to be overwhelming. But usually, in order to make the synthetic data regularization work, we need much more synthetic data than the original training data. To overcome this problem, we use sample weight in the objective function. If we have a training set $\{(\boldsymbol{x}_n, y_n)\}_{n=1,2,\ldots,N}$, we could generate K time synthetic data $\{(\boldsymbol{x}'_n, y'_n)\}_{n=1,2,\ldots,KN}$. Then the objective function to minimize would be:

$$J(\theta) = \sum_{n=1}^{N} err(\boldsymbol{x}^{(n)}, y^{(n)}; \theta) + \frac{\lambda}{K} \sum_{n=1}^{KN} err(\boldsymbol{x}^{(n)}, y^{(n)}; \theta), \tag{2.37}$$

where err is the error function, and θ stands for parameters of our model. If we do not use sample weight, $\lambda = K$ and synthetic data will be overwhelming. But if we set λ to 1, the training data and synthetic data will have equal total weights, thus we do not need to worry about using too much synthetic data.

λ is a tunable hyper-parameters. In this project, we always set λ to 1 for the reported results, because we find that $\lambda = 1$ is a good option for most problems. We believe that cross validate λ could lead to a higher performance, however, this issue is not explored and could be studied in future work.

Chapter 3

A Synthetic Data Generating Approach

This chapter describes our synthetic data generating approach in detail. We specify our model structure, training algorithms and the sampling method in the first half. Then we introduce a few options for sampling, which may influence the distribution of our synthetic data. These options can be considered as hyper-parameters of our approach, and could be useful under some circumstances. In the end, we introduce a measure of synthetic data quality called DIS, which is mainly used for debugging and reasoning in our project.

3.1 Dependency network

Dependency network [19] is a graphical model that can represent relationships between variables. Based on this graphical model, we introduce a new framework to build a generative model, which is quite different from the generative models we discussed in the last chapter. Under this framework, our model is a collection of models which are able to model the conditional $P(x_m \mid \boldsymbol{x}_{\setminus m})$. The training procedure depends on which conditional distribution model we are actually using, but the sampling method is unified. In this part, we will introduce the structure of dependency network and the sampling method. In the later sections, we will concentrate on various conditional distribution models, including models for categorical features and models for real-valued features.

3.1.1 Definition

A consistent dependency network is a pair $(\mathcal{G}, \mathcal{P})$, where $\mathcal{G}$ is a cyclic directed graph, and $\mathcal{P}$ is a set of conditional distributions. Each node corresponds to a random variable, and the edge goes from a parent node to a child node, thereby defining a direct dependent relationship between variables.

Considering a set of variables $\boldsymbol{x} = (x_1,\ x_2,\ ...,\ x_M)$, $(\mathcal{G}, \mathcal{P})$ is a dependency network for $\boldsymbol{x}$ when

$$P(x_m \mid \boldsymbol{x}_{\setminus m}) = P(x_m \mid Pa(x_m)). \tag{3.1}$$

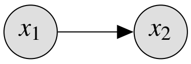
Here $m = 1, 2, ..., M$, and $Pa(x_m)$ denotes parent nodes of node x_m in graph $\mathcal{G}$.

We call $P(x_m \mid Pa(x_m))$ local probability distribution[19].

3.1.2 Inconsistency

Given a dependency network $(\mathcal{G}, \mathcal{P})$ over $\boldsymbol{x} = (x_1,\ x_2,\ ...,\ x_M)$, it is not always possible to find a joint distribution P over $\boldsymbol{x}$ that all the conditional distributions in $\mathcal{P}$ can be deduced. By relaxing the consistent condition, we obtain a flexible model that is sometimes very useful. We can call it inconsistent dependency network. From now on, when we say dependency network, we do not assume the consistent condition.

Figure 3.1: An example of inconsistent dependency network



According to the graph, we have local probability distributions $P(x_2 \mid x_2)$ and $P(x_2)$ (x_1 has no parents in the graph). However, according to Bayes' rule, if our local probability distributions are consistent, and $P(x_1 \mid x_2) = P(x_1)$, we can always have $P(x_2 \mid x_1) = P(x_2)$. Obviously, it cannot be guaranteed simply based on the dependency network model. Figure credits: IRP [42]

We give a simple example of inconsistent dependency network in figure 3.1. In the context of this project, we try to model $P(x_m \mid \boldsymbol{x}_{\setminus m})$ separately. Because our models are just approximating the ground truth, there is no way to guarantee these conditional distributions are consistent. However, consider the fact that these local probability distributions are all learned from the same dataset, these conditionals may only have small deviations from the truth. We call this state it near consistency[19] and in practice, it may not cause any problem.

3.1.3 Sampling

We have many choices for conditional distribution models, and each choice actually leads to a different generative model. However, under the framework of the dependency network, the sampling method is unified.

As discussed in section 2.2, the transition kernel of Gibbs sampling can be determined totally upon conditional distributions $P(x_m \mid \boldsymbol{x}_{\backslash m})$. Because a dependency network model for $\boldsymbol{x} = (x_1,\ x_2,\ ...,\ x_M)$ could be roughly seen as a collection of local probability distributions $P(x_m \mid Pa(x_m))$, where $m = 1,2,...,M$, Gibbs sampling is a natural choice for the whole family of generative models which are based on dependency network.

Technically, if our dependency network is inconsistent, what we are actually running is just pseudo Gibbs sampling, as there may not exist a global joint distribution at all. So our concern is, even though the deviation is small for a particular local probability distribution, will it be amplified along with the sampling procedure? This issue is examined in [19] from a theoretical aspect, and we find it not a problem in our applications. In the following part, we will still call our sampling Gibbs sampling, in spite of the inconsistency of local distributions.

3.1.4 Pseudo likelihood

It is possible to train all the conditional distribution models separately, but we could also define a global loss function and train these models together.

We have mentioned in section 2.1.1 that many error functions can be unified by NLL (Negative Log Likelihood) in a probabilistic aspect.

Let's say our dataset is $D = \{\boldsymbol{x}^{(n)}\}_{n=1,2,...,N}$, where $\boldsymbol{x}^{(n)} = (x_1^{(n)}, x_2^{(n)}, ..., x_M^{(n)})$. The NLL loss function for each conditional distribution model is

$$\mathcal{L}_m(\theta) = -\sum_{n=1}^{N} \log P(x_m^{(n)} \mid \boldsymbol{x}_{\backslash m}^{(n)}; \theta). \tag{3.2}$$

Instead, we could define a global pseudo likelihood

$$PL(\theta) = \prod_{n=1}^{N} \prod_{m=1}^{M} P(x_m^{(n)} \mid \boldsymbol{x}_{\backslash m}^{(n)}; \theta). \tag{3.3}$$

Then a global loss function can be defined on top of this pseudo likelihood:

$$\mathcal{L}(\theta) = -\log PL(\theta) = -\sum_{n=1}^{N}\sum_{m=1}^{M} \log P(x_m^{(n)} \mid \boldsymbol{x}_{\setminus m}^{(n)};\theta) \tag{3.4}$$

3.2 Neural network as a model of conditionals

Starting from this section (until section 3.6), we introduce various choices for conditional distribution models. In this part, we turn the neural network model (introduced below) into a conditional distribution model for categorical features. We first talk about a standard feed forward architecture, and then introduce a novel architecture, which shares parameters among individual neural networks.

3.2.1 Neural network

A multilayer feed forward neural network, also known as a multilayer perceptron (MLP) stacks multiple artificial neural network layers, and each layer is a combination of a linear transformation and a non-linear element-wise activation function.

Formally, given input feature vector $\boldsymbol{x}$, An artificial neural network layer consists of a linear transformation

$$\boldsymbol{z} = W\boldsymbol{x} + \boldsymbol{b}, \tag{3.5}$$

and a non-linear element-wise activation function

$$\boldsymbol{a} = \sigma(\boldsymbol{z}). \tag{3.6}$$

The non-linearity activation function σ can take many forms such as

$$\sigma(z) = \frac{1}{1+\exp(-z)} \qquad \text{(sigmoid function)} \tag{3.7}$$

$$\sigma(z) = \frac{1-\exp(-2z)}{1+\exp(2z)} \qquad \text{(hyperbolic tangent function)} \tag{3.8}$$

$$\sigma(z) = max(0,z) \qquad \text{(rectified linear unit)} \tag{3.9}$$

$$\tag{3.10}$$

In this project, we find out that the hyperbolic tangent function (Tanh) and the rectified linear unit (ReLU) usually performs better than sigmoid function in our tasks. The choice between the former two usually does not matter. In practice, we recommend ReLU because it avoids the so-called saturation problem (the gradient is close to 0 when the element of $\boldsymbol{z}$ has big magnitude) [31].

For a neural network with L layers, the architecture can be formalized as

$$\boldsymbol{a}^0 = \boldsymbol{x} \tag{3.11}$$

$$\boldsymbol{a}^l = \sigma(W_l \boldsymbol{a}^{l-1} + \boldsymbol{b}_l), \tag{3.12}$$

where $l = 1, 2, ...L$.

In order to model the conditional of a categorical feature, the output $\boldsymbol{a}^{(l)} = (a_1^l, a_2^l, ..., a_C^l)$ must define a categorical distribution, which means we need to have

$$a_c^l \geq 0 \tag{3.13}$$

$$\sum_{c=1}^{C} a_c^l = 1 \tag{3.14}$$

To ensure this property of outputs, we substitute the non-linear activation function with the softmax function $\boldsymbol{a}^{(l)} = softmax(\boldsymbol{z}^{(l)})$, where

$$a_c^l = \frac{\exp z_c^l}{\sum\limits_{c=1}^{C} \exp z_c^l}. \tag{3.15}$$

Because of softmax function, we guarantee the property in equations 3.13 and 3.14.

3.2.2 Parameters sharing

When we use the neural network to model $P(x_m \mid \boldsymbol{x}_{\setminus m})$, we need M neural networks, one for each conditional. However, it is possible to share parameters among these models, and use just one single neural network to model all the conditionals.

The architecture is illustrated in figure 3.2. For each conditional $P(x_m \mid \boldsymbol{x}_{\setminus m})$, we build a binary mask $\boldsymbol{m}_m$ with length M, with the m-th element being 0 and all the rest being 1. The masked inputs $\bar{\boldsymbol{x}}$ is

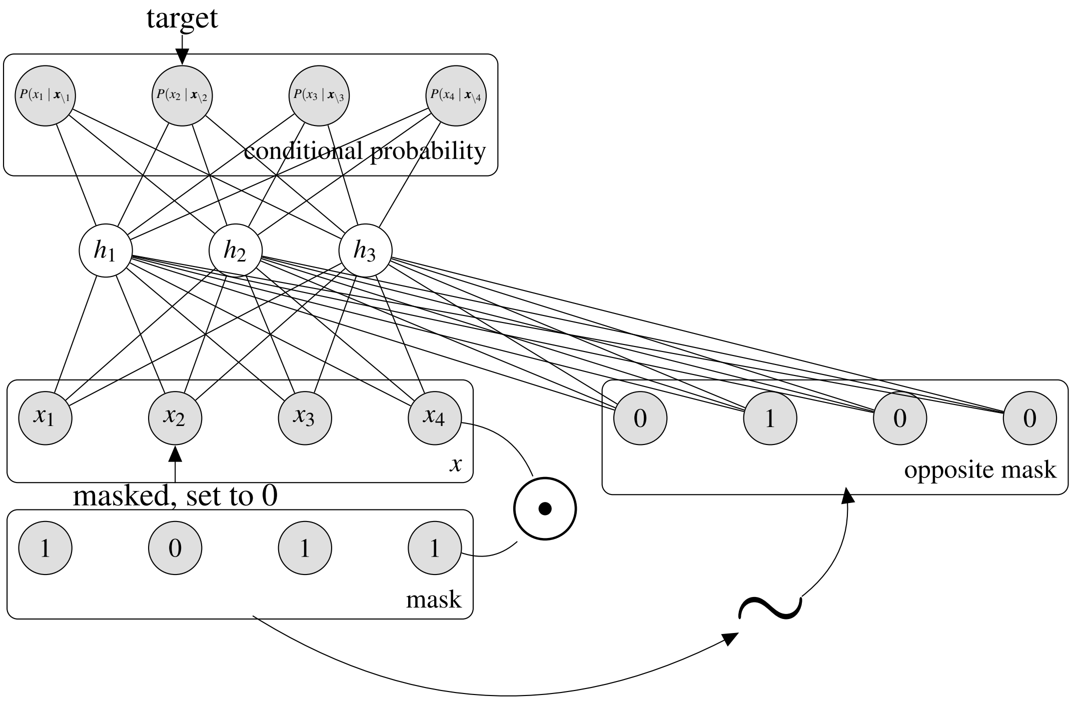
$$\bar{\boldsymbol{x}} = \boldsymbol{x} \odot \boldsymbol{m}_m, \tag{3.16}$$

where $\odot$ denotes element-wise multiplication. Except for the output layer, all the parameters are shared by M conditional distribution models. The global loss function is given by pseudo likelihood (see equation 3.4).

The parameters sharing architecture borrows ideas from NADE (Neural Autoregressive Density Estimator) [38], which is a state-of-the-art density estimation method. Generally, NADE uses neural network outputs $(a_1, a_2, ..., a_M)$ to model

$$P(x_{o_m} \mid \boldsymbol{x}_{o_{<m}}), m = 1, 2, ..., M, \tag{3.17}$$

Figure 3.2: Neural network with weights sharing


target
$P(x_1 \mid \boldsymbol{x}_{\setminus 1})$
$P(x_2 \mid \boldsymbol{x}_{\setminus 2})$
$P(x_3 \mid \boldsymbol{x}_{\setminus 3})$
$P(x_4 \mid \boldsymbol{x}_{\setminus 4})$
conditional probability
h_1
h_2
h_3
x_1
x_2
x_3
x_4
x
0
1
0
0
opposite mask
masked, set to 0
$\odot$
1
0
1
1
mask
$\sim$

This figure is taken from IRP [42].

where order o is a permutation of $1, 2, ..., M$, x_{o_m} is the m-th variable according to the order o, and $\boldsymbol{x}_{o_{<m}}$ means all the variables before x_{o_m} in order o. As you can see, the model depends on the order o, and there are $M!$ different orders. So an order-agnostic version or NADE shares parameters among models using different orders.

However, it has been pointed out in [38] that the strength of parameters sharing is sometimes too strong, and could lead to poor performance. Parameters sharing significantly reduces the number of parameters, thus can be roughly seen as a very strong regularization. To relax it a little bit, we also adopt the technique proposed in [38]. Previously, we use a binary mask $\boldsymbol{m}_m$ to mask the inputs. Now to indicate which variable is masked by the binary mask, we include opposite of the binary mask $\sim \boldsymbol{m}_m$ as additional inputs. As illustrated in 3.2, now $(\bar{\boldsymbol{x}}, \sim \boldsymbol{m}_m)$ is actually our neural network inputs. Since only the m-th element in $\sim \boldsymbol{m}_m$ is non-zero, only weights associated with the m-th element have effects. The net effect is that, an additional bias term is added to the original structure. However, because different m defines different masks, these additional bias terms are not shared by these M models. Now each model has its own bias term, thus relax the constraint of parameters sharing.

3.3 Calibrated classifiers

Theoretically, it is possible to turn arbitrary classification methods into conditional distribution models for categorical features through probability calibration. In this project, we adopt Platt scaling (or Platt calibration)[34], and transform SVM (Support Vector Machine)[18] and GB (Gradient Boosting)[14][15] into conditional distribution models. An alternative approach is to fit an isotonic regression model to the ill-calibrated model, and it has been proven to work even better than Platt scaling when we have enough training data[32]. However, since we focus on the situation when training data is insufficient, the isotonic regression is not suitable.

To understand experiments in this project, we only need to know that we transform SVM and GB into conditional distribution models through Platt scaling, even though the original methods do not carry explicit probability meanings.

To be more specific, consider a binary classification task, if our classifier can output a score $f(\boldsymbol{x})$ for inputs $\boldsymbol{x}$, (The function f is often called decision function) and the binary label is predicted by $sign(f(\boldsymbol{x}))$, we could map the score $f(\boldsymbol{x})$ into an estimated probability.

$$P(y=1 \mid \boldsymbol{x}) \approx P_{A,B}(f) = \frac{1}{1+\exp(Af(\boldsymbol{x})+B)}[27] \tag{3.18}$$

where A,B are estimated by solving a maximum likelihood problem.

3.4 Random forest

A decision tree is a tree-like model that each node corresponds to a test on one feature. Based on the outcome of this test, the tree grows out branches. In the end, each sample resides in one leaf. The decision tree can easily assign a class probability to a sample belongs a certain leaf. The probability is computed as the fraction of training samples of a class in that leaf.

Random forest [5] can be roughly seen as an ensemble of decision trees. So the class probability is just the mean predicted class probability of all the decision trees. Therefore, we do not need calibration methods to turn the random forest into a conditional distribution model. Random forest is already a well-calibrated model, and can be directly used for our purpose.

3.5 Mixture density network

Previously, we talk about many conditional distribution models for categorical features. In this section, we move on to talk about a conditional distribution model for real-valued features. The model is called MDN (Mixture Density Network) [3]. Just as its name suggests, it is a combination of mixture density model and neural network.

3.5.1 Model structure

The mixture density model extends the idea of GMM (Gaussian Mixture Model) (see section 2.4). In GMM, the component distributions are all Gaussians. But in fact, the components can be arbitrary distributions over the domain of interest. We restrict the mixture density model to GMM here, because that's enough to understand the experiments in this projects. An extension of model structure using arbitrary mixture density models would be straightforward, given materials in this part.

Recall that a one-dimensional Gaussian mixture distribution has the following density function

$$P(x) = \sum_{c=1}^{C} \omega_c \mathcal{N}(x; \mu_c, \sigma_c^2), \tag{3.19}$$

where $\sum\limits_{c=1}^{C} \omega_c = 1, \omega_c \geq 0$, and $\mathcal{N}$ is the density function for Gaussian distribution.

Now we use this model to approximate $P(x_m \mid \boldsymbol{x}_{\setminus m})$. In order to do that, a feed forward neural network (see section 3.2.1) is used to represent a mapping from $\boldsymbol{x}_{\setminus m})$ to all the parameters of GMM. Because GMM is parameterized by $\{\omega_c, \mu_c, \sigma_c\}_{c=1,2,...,C}$, the output vector has a dimension of $3C$. You may notice that these parameters are constrained, so reparameterize techniques must be applied. Assuming $\boldsymbol{z} = (z_1, z_2, ..., z_{3C})$ is the final outputs (before any non-linearity), we denote

$$\boldsymbol{z}_\omega = (z_1, z_2, ..., z_C) \tag{3.20}$$

$$\boldsymbol{z}_\mu = (z_{C+1}, z_{C+2}, ..., z_{2C}) \tag{3.21}$$

$$\boldsymbol{z}_\sigma = (z_{2C+1}, z_{2C+2}, ..., z_{3C}). \tag{3.22}$$

Now we map these outputs to the parameters we want:

$$(\omega_1, \omega_2, ..., \omega_M) = softmax(\boldsymbol{z}_\omega) \tag{3.23}$$

$$(\mu_1, \mu_2, ..., \mu_M) = \boldsymbol{z}_\mu \tag{3.24}$$

$$(\sigma_1, \sigma_2, ..., \sigma_M) = \exp(\boldsymbol{z}_\sigma) \tag{3.25}$$

The definition of a softmax function is already given in 3.2.1.

Through reparameterization, training is just an unconstrained optimization problem, and can be solved by standard packages.

3.5.2 Initialization of MDN

It is worthwhile to mention that, sometimes, people use GMM to initialize the bias terms of the output layer in MDN [30]. To be specific, if we are using MDN to model $P(x_m \mid \boldsymbol{x}_{\setminus m})$, we could use GMM to model the marginal $P(x_m)$ at first. The number of components in MDN should be equal to the number of components in GMM. The bias terms of the output layer should be set in such a way that, if all the weights are 0, the MDN recovers GMM, which is actually a model of the marginal.

In this project, we tried this initialization scheme but cannot improve performance when we apply our approach for regularization. So we simply initialize all bias terms to 0.

3.6 Generative adversarial network as a model of conditionals

GAN (Generative Adversarial Network) [17] is a newly proposed generative model. So far, GAN is mainly applied to image or sequence data, and applications in applied machine learning problems are rarely seen. In this section, we first introduce the structure of GAN and how to sample from it. Then, we modify the structure of conditional GAN [28], making it suitable for our problem.

3.6.1 Adversarial network

To model a distribution $P(\boldsymbol{x})$ over $\boldsymbol{x}$, we use a multilayer feed forward neural network $G(\boldsymbol{z};\theta_g)$ as a generator. We define a prior $P_z(\boldsymbol{z})$ over noise variables $\boldsymbol{z}$ (for example a Gaussian distribution). The neural network G is simply a mapping from noise space to data space. If $G(\boldsymbol{z};\theta_g) \sim P_g$, we want P_g to learn the data distribution $P(\boldsymbol{x})$. So we define a second neural network $D(\boldsymbol{x};\theta_d)$ to discriminate $G(\boldsymbol{z};\theta_g)$ from $\boldsymbol{x}$. The discriminator D outputs a scalar, assigning probability of how likely the input sample comes from P_{data} rather than P_g. Simultaneously, the generator G is trained to maximize the error of the discriminator (to fool the discriminator), whereas, the discriminator D is trained to minimize it. Formally, G and D is playing a two-player minimax game, which can be formulalized as:

$$\min_G \max_D V(D,G) = <\log D(\boldsymbol{x})>_{\boldsymbol{x}\sim P_{data}(\boldsymbol{x})} + <\log(1-D(G(\boldsymbol{z})))>_{\boldsymbol{z}\sim P_{\boldsymbol{z}}(\boldsymbol{z})}, \quad (3.26)$$

where $<\cdot>$ is a notation for expectation, and V is called value function in game theory.

3.6.2 Sampling from GAN

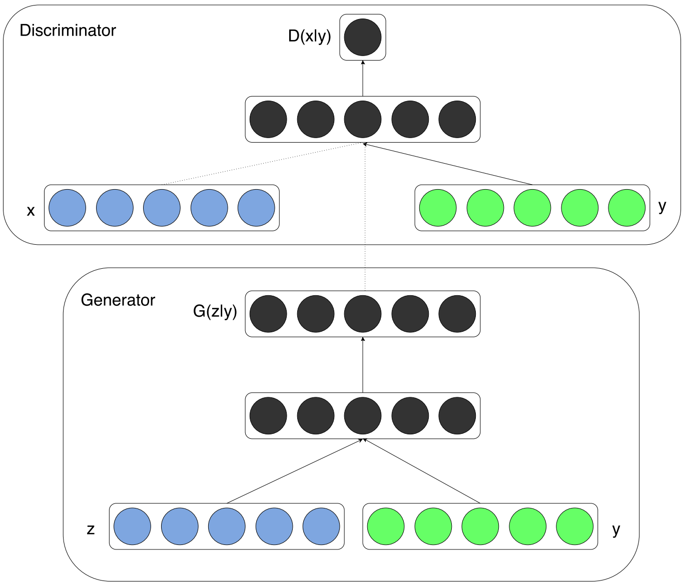
Sampling from GAN is extremely easy. We only need to sample prior noise z from $P_{\boldsymbol{z}}(\boldsymbol{z})$. The prior distribution is chosen by us, and is usually Gaussian distribution or uniform distribution. So sampling the prior noise is straightforward. Then computing $G(\boldsymbol{z})$ is just a forward pass through the neural network, and $G(\boldsymbol{z})$ is the sample we want.

3.6.3 An architecture to model conditionals

We construct an architecture to model conditionals by feeding the variables we conditioning on as additional inputs. As shown in figure 3.3, to model $P(\boldsymbol{x} \mid \boldsymbol{y})$ with GAN, the prior noise $\boldsymbol{z}$ and the given variables $\boldsymbol{y}$ are concatenated as inputs in generator. In discriminator, we discriminate $(G(\boldsymbol{x} \mid \boldsymbol{y}), \boldsymbol{y})$ from $(\boldsymbol{x}, \boldsymbol{y})$. The architecture is called conditional GAN [28].

However, in our problem, we want to model $P(x_m \mid \boldsymbol{x}_{\setminus m})$. Since $\boldsymbol{x}_{\setminus m}$ are all given, synthetic data only have one feature that is different from real data. It could cause difficulty for the discriminator, because synthetic data and real data are so similar except for one feature, and the discriminator has no idea which feature it is. To make training easier, we add the opposite binary mask $\sim \boldsymbol{m}_m = (\mathbb{1}(m=1), \mathbb{1}(m=2), ..., \mathbb{1}(m=M))$ (binary mask has been introduced in 3.2.2) as additional inputs to the discriminator. So now we present $(G(\boldsymbol{x} \mid \boldsymbol{y}), \boldsymbol{y}, \sim \boldsymbol{m}_m)$ to the discriminator. This opposite binary mask indicates which feature is perturbed, thus the discriminator would find it easier to discriminate generated samples.

Figure 3.3: Conditional GAN



The goal is to model $P(x \mid y)$, and z denotes prior noise. The prior noise z and the given variables y are concatenated as inputs to the generator. x and y are concatenated as features for discrimination. Figure credits to [28].

3.7 An intermediate summary

Based on a graphical model called the dependency network, we introduce a novel approach to generate synthetic data. Under the framework of the dependency network, we discuss various choices for conditional distribution models. Below is a list of them:

- Conditional distribution models for categorical features: neural network, neural network with parameters sharing, gradient boosting), SVM, random forest.
- Conditional distribution models for real-valued features: mixture density network, generative adversarial network.

We also provide fundamental steps of our approach in algorithm 1

Algorithm 1: The synthetic data generating approach

```
Input  : dataset $D = \{\boldsymbol{x}^{(n)} = (x_1^{(n)}, ..., x_M^{(n)})\}_{n=1,...,N}$, sampling round $R$,
         sampling order $O$ (permutation of $1, 2, ..., M$), conditional distribution
         models $\{\mathcal{M}_m\}_{m=1,2,...,M}$
Output: synthetic dataset $S$
for $m \leftarrow 1$ to $M$ do
    $X = \{\boldsymbol{x}_{\backslash m}^{(n)}\}_{n=1,...,N}$;
    $Y = \{x_m^{(n)}\}_{n=1,...,N}$;
    dataset $C_m = (X, Y)$;
    train $\mathcal{M}_m$ on $C_m$ to model $P(x_m \mid \boldsymbol{x}_{\backslash m})$;
end
for $n \leftarrow 1$ to $N$ do
    $\boldsymbol{x} = (x_1, ..., x_M) \leftarrow \boldsymbol{x}^{(n)}$;
    for $r \leftarrow 1$ to $R$ do
        for $m \leftarrow 1$ to $M$ do
            $m = O_m$;
            $x_m \leftarrow$ Sample from $\mathcal{M}_m(x_m \mid \boldsymbol{x}_{\backslash m})$;
        end
    end
    add $\boldsymbol{x}$ to $S$;
end
return $S$;
```

3.8 Randomized sampling order

We run Gibbs sampling (section 2.2) to generate samples from our model, setting training data as initial samples. The sampling procedure can be seen as perturbing features one by one. In practice, the sampling order can be an arbitrary permutation of attributes, so an interesting question would be, how does sampling order influences the generated samples, thus influences the regularization performance when using these synthetic data for regularization?

In this project, we do not try to explore whether there exists a best sampling order for a particular problem, but use a random order. There are two options in practice. Firstly, we can decide a sampling order at the beginning of each run, and then stick to that order when we run Gibbs sampling. Secondly, we could have different sampling order for each initial sample. We call the first one a fixed order, and the second one a randomized order. The choice may cause some differences and have effects on the generated synthetic data.

Intuitively, when we adopt a fixed order, the order we are using will determine the distribution of synthetic data. Different orders will lead to slightly different distributions, especially when the sampling procedure hasn't reached the equilibrium distribution. When a randomized order is adopted, it is like mixing different sampling orders in one run. So potentially, a randomized order will make our results less dependent on the order we choose. However, in practice, we could run many times using a fixed order, and combine all the synthetic data together.

3.9 Introducing sampling temperature

3.9.1 Motivation

When we train a generative model, the error function is often NLL (Negative Log Likelihood) (discussed in 2.1.1) of the training data under our model. It is also the case when we train our conditional distribution model. However, there is one thing about NLL that we need to pay attention to. If a sample is very unlikely to appear but actually exist in the training set, the error will be very big and we are heavily punished for that. Let's take a look at the definition of NLL again:

$$NLL = -\sum_{n=1}^{N} \log P(\boldsymbol{x_n}, y_n). \tag{3.27}$$

If $P(\boldsymbol{x_n}, y_n)$ is extremely small, $-\log P(\boldsymbol{x_n}, y_n)$ would be very big. So if NLL is our training objective, our model will try everything to avoid that situation. The consequence is, our model will spread the probability mass just to cover the outliers.

However, we do not want to sample from these locations where only some outliers reside. So we want our model to be more concentrated on the region with high probability mass. Therefore, we introduce the concept of temperature below, which is in accord with this intuition.

3.9.2 Formal Definition

If $P(x)$ is density function for the distribution we draw samples from, we can define a new distribution:

$$P_T(x) = \frac{1}{Z} P(x)^{\frac{1}{T}}, \tag{3.28}$$

where $Z = \int P(x)^{\frac{1}{T}} dx$ is the normalization constant.

When $T = 1$, we recover the original distribution $P(x)$. When $T \to \infty$, we recover a uniform distribution over positive density region. When $T \to 0$, x is always the maximum likelihood solution:

$$x = \arg\max_x P(x) \tag{3.29}$$

When we run Gibbs sampling in our problem, we sample from $P(x_m \mid \boldsymbol{x}_{\setminus m})$ one by one following a particular order. So when we say we run Gibbs sampling at temperature T, at each step, we are actually sampling from:

$$P_T(x_m \mid \boldsymbol{x}_{\setminus m}) = \frac{1}{Z_m} P(x_m \mid \boldsymbol{x}_{\setminus m})^{\frac{1}{T}}, \tag{3.30}$$

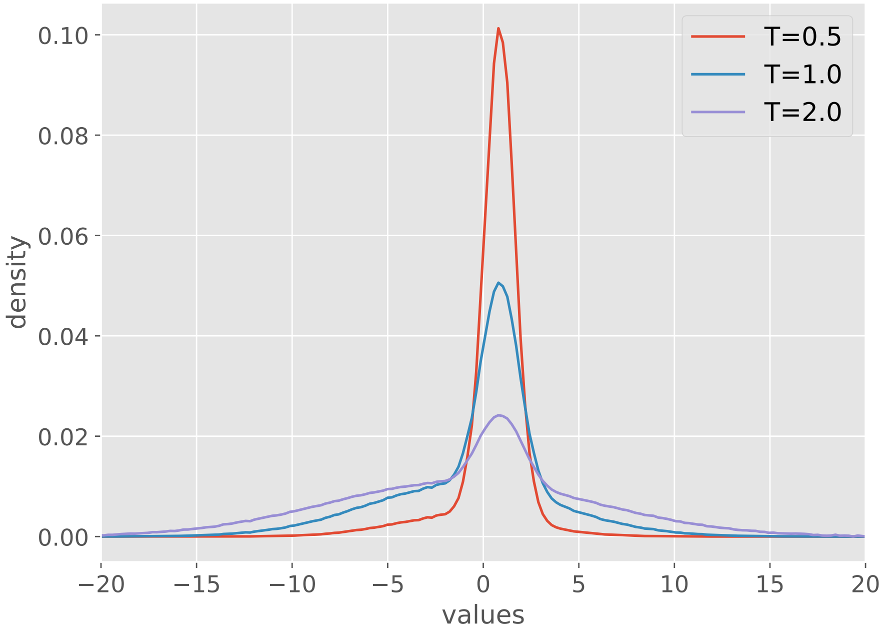
where $Z = \int P(x_m \mid \boldsymbol{x}_{\setminus m})^{\frac{1}{T}} dx$.

We provide an illustration in figure 3.4. When $T = 1$, we have a broad distribution, which covers a wide area far away from the central region. Then we decrease the temperature to $T = 0.5$, now you can see more probability mass is put on the central region. We also provide the distribution when $T = 2$ as comparison, which goes the opposite way.

3.9.3 A heuristic definition for Gaussian mixture model

In order to sample from $P_T(x)$, we need to compute the normalization constant. It is easy to compute when the distribution is over a discrete variable, the computation is

Figure 3.4: Effects of temperature



When the temperature decreases, we favour the region with high probability mass more, and seldom draw samples from the region with low density.

simply a summation. Unfortunately, for many distributions over a real variable, the integral is hard to compute. In this project, we will have to deal with Gaussian mixture distribution, which has the following density function:

$$P(x) = \sum_{k=1}^{K} \omega_k \mathcal{N}(x; \mu_k, \sigma_k^2), \tag{3.31}$$

where K is the number of components. Then the normalization constant for temperature T

$$Z = \int [\sum_{k=1}^{K} \omega_k \mathcal{N}(x; \mu_k, \sigma_k^2)]^{1/T} \tag{3.32}$$

is hard to compute. So instead, we give a heuristic definition of $P_T(x)$ as:

$$P_T(x) = \frac{1}{Z} \sum_{k=1}^{K} \omega_k^T \mathcal{N}(x; \mu_k, \frac{\sigma_k^2}{T}). \tag{3.33}$$

The normalization constant Z is now is to obtain because:

$$Z = \sum_{k=1}^{K} \omega_k^T. \tag{3.34}$$

There are of course other ways to define it, and this definition here is just trying to accord with our intuition.

3.10 DIS: A measure of synthetic data quality using a discriminator

If we have a synthetic dataset gen and a training set $train$, we could create a binary classification task discriminating fake samples from real samples. The original features and labels will all be treated as input features in this task. The labels are 0 for samples in gen, and 1 for samples in $train$. Intuitively, the best accuracy a discriminator can achieve tell us how good our synthetic data are. Of course, the performance is not deterministic, and the results depend on the classifier we are using.

Based on this idea, we would like to define a measure of synthetic data quality. The goal is to invent a tool for debugging and reasoning, rather than a formal theoretical measure. It is more like an illustration that can help us build our intuition and criticize our model.

3.10.1 Definition

We define

$$DIS_{\text{classifier cls, random seed rs}}(gen, train) \tag{3.35}$$

as the output of the following procedure:

1. If $train$ and gen have different sizes, resample without replacement from the larger one to make sure that the two datasets have the same size.

2. Transform the original labels into additional input features, and assign new labels to indicate which dataset the samples come from.

3. Mix samples from the two datasets, and split them into the training set T and the validation set V ($50\% - 50\%$ split).

4. Train cls on T, and tune hyper-parameters on V. Report the best accuracy on V.

5. All random choices are determined by random seed rs.

3.10.2 Notes

About this measure, there are a few things we need to point out:

- Strictly, it is not a measure. The result depends on the classification method, hyper-parameters for the method, and random seed.

- If the synthetic data is perfect, we should get an accuracy that is close to 50%.
- Because when usually train our generative model on the training data, we will also want to see $DIS_{\text{cls, rs}}(gen, valid)$, where *valid* is the validation set of the original problem.

3.10.3 Why we need this?

If we want to evaluate our generative model, a straightforward solution is to calculate the likelihood of training data under the model. If we have a generative model $P(\boldsymbol{x}, y)$, and a training set $\{\boldsymbol{x_n}, y_n\}_{n=1,2,\ldots,N}$, we want to know how good our model is. We normally calculate the NLL (Negative Log Likelihood):

$$NLL = -\sum_{n=1}^{N} \log P(\boldsymbol{x_n}, y_n). \tag{3.36}$$

Unfortunately, this approach does not apply to our dependency network model. Since a collection of conditional distributions is all we have, it is not possible to calculate NLL. Alternatively, we can calculate the pseudo likelihood 3.1.4.

So the question is, why we don't directly evaluate the generative model? First of all, we generate data by perturbation. An evaluation of our generative model cannot tell us how good the synthetic data are after the first, or second round of perturbation. Secondly, it is not always possible to calculate *NLL*. In our case, only the pseudo likelihood can be provided. In the case of MUNGE 2.5.2, nothing similar can be defined. Thirdly, the DIS is an intuitive measure. It is sometimes more useful for debugging and reasoning.

If we only care about synthetic data regularization, can we just feed synthetic data into the final predictive model, and observe the final performance? Surely it is a good idea. But sometimes, we want to make the data generating procedure right before involving the final task. It is a more modularized workflow. Such a workflow can avoid building a big system, but do not know which part is wrong when debugging.

Chapter 4

Synthetic Data for Regularization

A synthetic data generating approach has been described in the previous chapter. In practice, these synthetic data can be useful in many ways. This chapter presents empirical results about using the synthetic data for regularization.

4.1 An introduction of default experiments settings

We will start with introducing a few default experiments settings, to ensure the reproducibility of our experiments. Many of our experiments share the same settings, so it would be verbose to talk about them in every part. Unless stated explicitly, experiments are conducted in a way that follows the default settings here.

4.1.1 Datasets

Table 4.1: A summary of datasets

Datasets	**AD**[1]	**RCV1**	**OTTO**	**RNA**	**COVTYPE**	**HOUSES**
Feature type	categorical	categorical	integer	real	mixed[1]	real
Task Type	classification	classification	classification	classification	classification	regression
Number of classes	2	2	9	2	2	-
Number of attr.	50	150	93	8	12	8
Train set size	1000	500	10000	2000	5000	1000
Valid set size	1000	500	10000	2000	5000	1000
Test set size	35000	38000	20000	39535	80000	1760

[1] The perfect Bayesian classifier using the underlying generative model gives 99.57% on the AD dataset.

[2] It contains 2 categorical features and 10 real value features

At the beginning, we give a summary of datasets in table 4.1,and the arguments presented in the following parts will be demonstrated on these datasets. It is our intention to make sure the task is not too easy, so that it is possible to observe a significant improvement. We also prefer datasets with sufficient data, so that a big test set can be used to provide a reliable comparison.

All in all, we have 3 categorical datasets (the OTTO dataset is not a categorical dataset, but in practice, we treat it as one) and 3 datasets with real-valued features. 4 of them are binary classification tasks, and the other two are a multi-class classification task and a regression task.

The artificial dataset (AD) is generated as follows: A Bayesian network with 50 variables is randomly generated. For each new variable, we randomly pick 1 to 5 parents (uniform) from the previously generated variables. All the variables are binary, and all the conditional probabilities in the factorization take random values between 0.1 and 0.9 (uniform). This dataset has two classes with the same factorization, but different conditional distributions.

RCV1, RNA, and COVTYPE datasets are all from UCI machine learning repository[26]. However, clean and pre-processed versions from the LIBSVM dataset [9] are used in our experiments. The HOUSES dataset is also taken from the LIBSVM dataset, and original dataset appears in [33].

The OTTO dataset is taken from the Kaggle competition *Otto Group Product Classification Challenge* [1]. The feature values are all counts of products, with most of them close to 0. In practice, when we build the generative models, we treat these features as categorical features with 5 classes $0, 1, 2, 3, \geq 4$, so that is why we present the results in the parts for categorical datasets.

It should be clarified that, all datasets are shuffled and split again. So the original sample order and dataset split do not maintain.

You may notice that the dataset split is different from usual: The training set and validation set is much smaller than the test set. That is because, our aim is not to provide the best possible solution on these datasets. Instead, we want to explore how to use synthetic data as regularization when data is relatively insufficient. So a small training set is deliberately used. Another consideration is that, the training set should be small enough so that significant improvement is possible. The test set is much bigger than the training set and validation set, in order to reduce the sampling error and provide a reliable comparison. Because we use a small training set, our conclusions

[1] https://www.kaggle.com/c/otto-group-product-classification-challenge

should not be extended to problems with large training sets.

Lastly, all the performance measures are calculated on the test sets, so that it is convenient and sensible to compare them with each other. The test sets are used only for reporting purposes, not for the tuning of models. Moreover, in case we find some interesting phenomena on the test set and need to do more experiments or modify our previous models, we also keep reserved datasets which have never been touched before and do not belong to all the three datasets.

4.1.2 Pre-processing

In our experiments, we try not to introduce tricky pre-processing techniques, otherwise we may find it hard to tell where our improvements come from. However, there are a few techniques which we find very necessary, and they are also recommended in applications of our methods.

Firstly, for a categorical feature x_m with c_m classes, we replace it with c_m binary features $(\mathbb{1}(x_m == 1), \mathbb{1}(x_m == 2), ..., \mathbb{1}(x_m == c_m))$, where $\mathbb{1}$ is an indication function. This is often called **one-hot encoding**. In our experiments, it is applied to all categorical features except binary features. It is a reasonable representation for all categorical features if we do not want to introduce untrue value ordering assumption to the feature. It has already been widely accepted that this is almost always beneficial.

Secondly, there are some numerical features in our datasets that are actually integers or are calculated from integers. We may have some problem when we use a real-valued distribution to model it. For example, if a Gaussian mixture model is used to model the conditionals, we may end up using one component with a narrow bandwidth to model a single integer value. That is because, under a real-valued conditional distribution model, the probability of taking a certain value is exactly 0. To avoid this problem, uniform noise $\sigma \sim U(-d2, d2)$ is added to the features, where d is the minimum gap between neighbouring feature values (in the case of integer features, $d = 1$).

Lastly, conditionals of real-valued features are generally hard to model, and we find out that a statistical procedure called PCA (Principle Component Analysis) [41] can make the job much easier. Generally, PCA applies an orthogonal transformation A to the inputs $\boldsymbol{x}$ and convert it into uncorrected variables (which are called principle components). PCA is often used to reduce dimensionality, because we can keep a subset of principle components. However, in our experiments, we only use it as a linear transformation and keep all the variables. That is to say, in our experiments, the dimension of data does not change after PCA.

It is worthwhile to clarify that, one-hot encoding and PCA are beneficial not only to the training of the dependency network, but also to the final task. In the following parts, all the benchmark results can benefit from all these pre-processing, and are compared to our methods on top of these techniques. Therefore, there will be no concern about whether our improvements come from these pre-processing techniques.

4.1.3 Metrics

In this part, we will introduce and clarify the definitions of several metrics. They are heavily used in our experiments and will be referred throughout the following chapters.

First of all, we would like to introduce the performance measure for classification tasks. We will simply report the accuracy because our datasets are well-balanced. In practice, we also calculated precision, recall and F-score on all tasks. It turned out that each of these measures gave similar conclusions in our experiments. So for simplicity, we omit results of other measures. The accuracy Acc is defined as:

$$Acc = \frac{T}{N}, \tag{4.1}$$

where T is the number of true predictions and N is the number of all test samples.

Secondly, R^2 (coefficient of determination) will be used as performance measure for regression tasks. Assuming we have N test samples with ground truth $\{f_n\}_{n=1,2,...,N}$, and the predictions corresponding to f_n is y_n. The residual for the nth sample will be $e_n = y_n - f_n$. On account of the above, we define total sum of square SS_{tot} as:

$$SS_{tot} = \sum_{n=1}^{N} (y_n - \frac{1}{N} \sum_{n=1}^{N} y_n)^2, \tag{4.2}$$

and define residual sum of square SS_{res} as:

$$SS_{res} = \sum_{n=1}^{N} e_n^2. \tag{4.3}$$

So

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}} \tag{4.4}$$

. If the predictions are perfect, $R^2 = 1$, and if we predict the expected output value regardless of input features, we will get $R^2 = 0$.

In order to measure the quality of synthetic data, we want to know how likely these data will be generated by the actual data distribution. NLL (Negative Log Likelihood) is such a measure:

$$NLL = \frac{1}{N}\sum_{n=1}^{N} -\log P(\boldsymbol{x_n}), \tag{4.5}$$

where $\{\boldsymbol{x_n}\}_{n=1,2,\ldots N}$ are the data points (jointly with labels), and $p(\cdot)$ is the density function. It is possible to calculate NLL on the artificial dataset because the exact underlying generative model is known.

4.1.4 Default settings for training neural network

In this project, many models use the neural network model as components, such as GAN, MDN and so on. We would like to clarify some experiments settings for neural network training. However, they are not directly related to our conclusions, so we won't go into details here. It is possible that changing some options could lead to better performance. But we did not explore them in this project.

- We adopted Xavier initialization [16] for all the parameters in neural networks except for biases (biases are all initialized to 0).
- We used the Adam optimizer [24] with initial learning rate 0.001 to update parameters.
- We did not use other techniques such as batch normalization [22] or dropout [37].

4.1.5 Randomized parameter optimization

We often need cross validation to estimate hyper-parameters, and grid search is a widely used method for this task. However, if there are many hyper-parameters, the number of combinations will grow exponentially, which makes grid search an expensive method. So instead of searching the whole grid, we conduct a randomized search over hyper-parameters [2]. Firstly, we pose distributions over each hyper-parameters (could just be a list of parameter values). Then during the cross validation, hyper-parameters are sampled from the distribution independently. Consider the fact that many hyper-parameters do not have significant effects on the performance, it is possible to achieve good estimation with much less computing time. Another advantage is that, the computing budget is tunable and can be easily configured.

We have two levels of cross validation: we need to pick hyper-parameters for the conditional distribution models, and we also need to pick hyper-parameters for the final estimators. It would be wrong if our generative models peek at the validation set in some way. Because in that case, even though an improvement is observed, it may just due to the fact that our generative models have access the validation set through cross validation. So in our experiments, K-fold cross validation is used for generative models within training data, and validation set is only used for the final tasks.

4.1.6 A convention of feeding synthetic data

In this thesis, when we say feeding synthetic data together with training data into an estimator, specifically we mean the following:

- We run Gibbs sampling for one round, and get samples at the end of the first round. In experiments, we find out that for most datasets, the sampler already shows a sign of reaching the equilibrium distribution after one round. So it makes no sense to run the sampling longer. On the other hand, we cannot improve our performance by running for less than one round.
- We run the procedure above for K times, and combine all the samples into one synthetic dataset. Now the amount of synthetic data is K times as many as the training data.
- We use sample weights to balance the synthetic data and training data, so that they have the equal total weights ($\lambda = 1$ in section 2.7). In experiments, we find out that, this is very often a good choice. Surely, sample weights for synthetic data can be tuned by cross validation.

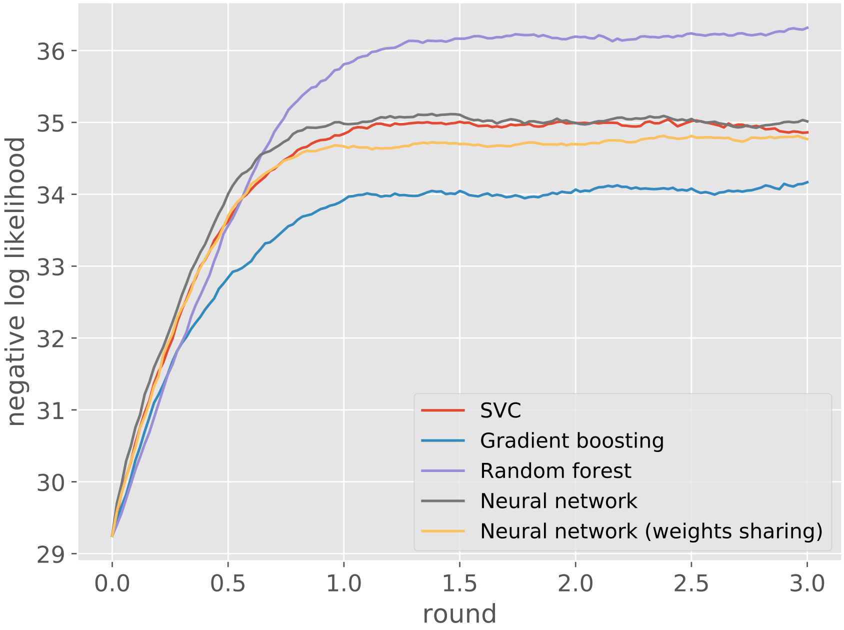
We do not conclude that all these conventions are the best for all problems. But as justified above, these choices are very often good default choices, and are recommended if you do not want to tune them.

4.1.7 Implementation

This project involves lots of models and methods. We develop a package on top of open source packages including *Tensorflow* [1], *Sklearn* [7], *XGBoost* [10], *Keras*[11], *Numpy* [40], *SciPy* [23]. This package is available at `https://github.com/Aaron-Jin-Xu/msc_package`.

4.2 A comparison of conditional distribution models for categorical features

Figure 4.1: NLL of synthetic data against sampling round on the AD dataset when using different conditional distribution models



(a) The fraction round is proportional to the Gibbs sampling steps, thus the number of steps in each round is equal to the number of attributes.

(b) Since a single MCMC run can be too noisy to identify the trend, the curves are actually averages of multiple runs.

(c) The starting point of each round is the NLL of training data, which estimates the entropy of the underlying distribution.

The performance of our synthetic data generating approach heavily depends on the choice of appropriate conditional distribution models. After all, our dependency network model can be roughly seen as a collection of these conditional distribution models. In the previous chapter, we discussed all kinds of model choices for categorical features, but we need to test which one works best in practice.

To give a preliminary comparison of these categorical conditional distribution models, we use dependency networks with each of these choices to model the AD dataset. Synthetic data can be generated from all these models through Gibbs sampling. Because we know the underlying model of the AD dataset, NLL (Negative Log Likelihood) can be calculated. For every sampling step, we calculate the NLL (given by equation 3.27) of the current samples, and plot the NLL against sampling steps in figure 4.1.

Our sampling method perturbs attributes one-by-one by resampling from the conditionals. Intuitively, our training data are corrupted little by little along with the sampling procedure. If NLL is a measure of synthetic data quality, it should increase along with the sampling steps. On the other hand, as discussed in section 2.2, Gibbs sampling will eventually reach an equilibrium distribution, and forget about the initial state (which is training data). Therefore, at same point, NLL should stop increasing. When we plot NLL against sampling steps, the convergence of this curve indicates reaching of the equilibrium distribution (we call mixing, see section 2.2). The equilibrium distribution is actually the distribution of our generative model, so the final NLL roughly indicates how good our model is. Compare the curves in figure 4.1, we have the following observations:

First of all, on the AD dataset, gradient boosting clearly outperform all other models because of fast mixing and the best NLL it can achieve in the end. The performance of SVM is close to neural network in this experiment, and random forest seems to be the worst model in terms of the NLL measure. In addition, an interesting observation is that, the neural network model with parameters sharing actually outperform the one without weight sharing, even though the former one has much fewer parameters.

Moreover, one this particular dataset, after one round of sampling, all these curves show a sign of mixing no matter which model we are using. That is to say, the distribution of samples at the end of the first round is already close to the equilibrium distribution.

However, the NLL measure only indicates how likely these synthetic data are generated by the true underlying distribution. Although it is our belief that data with high NLL often means it will benefit the final task more, this is not guaranteed. A good contrary example may be, if our generative model simply remembers all the training data, and only generate data that is close to the training data, then for sure it will achieve high NLL. But in that case, the synthetic data only gives us limited additional information, and may not be very useful in the final task.

We use synthetic data from the first round, and feed them together with the real training data into the final estimator. As shown in table 5.4, the synthetic data generated with different conditional distribution models all give an improvement of accuracy.

Gradient boosting still outperforms other models, and neural network with parameters sharing is still better than the one without parameters sharing. However, previously, we regard random forest as the worst conditional distribution in terms of NLL on the AD dataset, but now, it is slightly better than all the neural network models. This is a solid evidence that sometimes high likelihood does not mean it will work better as regularization.

Table 4.2: Accuracy on the AD dataset when feeding synthetic data together with training data, different conditional distribution models are used and compared to the accuracy without synthetic data

Conditional models	GB+Syn Accuracy
-	94.13% [1]
SVM	96.61%
Neural network	95.56%
Neural network (weights sharing)	95.95%
Random forest	96.17%
Gradient boosting	97.34%

[1] No synthetic is used here. This line is included for comparison.

4.3 Synthetic data regularization on categorical datasets

In this part, we give a study of applying synthetic data regularization to categorical datasets, and see if our approach can improve performance compared to state-of-the-art methods. In the previous section, we regard gradient boosting as a very good conditional distribution model, so from now on, gradient boosting will be the default conditional distribution model for categorical features unless otherwise stated.

The synthetic data regularization has only been applied together with gradient boosting (which is also used here as the classification method to report final performance). Logistic regression is meant to be a baseline model, and is often seriously underfitting. SVM cannot scale to large datasets, so it will be very time-consuming to use synthetic data. Regarding random forest, the implementation from *sklearn* seems

Table 4.3: Applying synthetic data regularization to categorical datasets

Dataset	**AD**	**RCV1**	**OTTO**[1]
Logistic Regression	91.08%	76.48%	74.68%
SVM	94.23%	84.15%	75.10%
RF (Random Forest)	93.05%	85.80%	78.18%
GB (Gradient Boosting)	94.13%	85.69%	78.58%
GB (train x 2)[2]	95.43%	87.20%	80.15%
GB+Syn(B)[3]	97.34%	86.61%	79.49%
GB+Syn(J) [4]	96.49%	86.62%	51.66%

[1] The OTTO dataset is presented in the categorical part because we treat counts as categorical features to construct the generative model.

[2] Double the training set size using reserved data, it is a good comparison to the results when feeding synthetic data.

[3] Model each class separately and construct the generative model using Bayes' rule.

[4] Model the data jointly with the labels.

to ignore the sample weights, thus the total weights of synthetic data would be much larger than the total weights of training data. Luckily, gradient boosting is almost always among the best models and the *XGBoost* implementation is highly efficient. However, we will talk about a technique in section 5.4 which could make it possible to apply the synthetic data regularization to other estimators like random forest and SVM.

The most important observation in table 4.3 is that, on all datasets, our synthetic data regularization contributes considerable improvement of accuracy, even though it is compared to many state-of-the-art methods. Actually, it is quite impressive that the accuracy using our synthetic data is close or even better than doubling the training set size.

However, we need to emphasize again that, since we always use datasets with very limited training data, the conclusions above should not be extended to tasks with sufficient training data. In fact, we find out that, if we have a large training data set, synthetic data regularization did not contribute significant improvement. This fact accords with our understanding that regularization is important when data is insufficient.

In practice, for a classification task, there are two options to build a generative model. The first one is to model each class separately, and acquire the whole generative model using Bayes' rule. The class probability can simply be estimated by counting training data labels. The other option is to treat the class label as another attribute, and model the data using just one dependency network. In table 4.3, we try both options on all categorical datasets. It turns out that, the option choice should be task-dependent. Generally, they all work well on AD and RCV1, but the second way seems to work poorly on the OTTO dataset, which has 9 classes and each class has quite different distribution of inputs. Intuitively, modeling the data jointly with labels is like sharing parameters between different classes. If the distributions of inputs from all these classes are similar, we can benefit from this option because now one class can borrow useful information from other classes. On the other hand, if this is not the case, just like the OTTO dataset, forcing different classes to share parameters may end up with a very bad generative model.

4.4 Mixture density network and generative adversarial network

As discussed above, the synthetic data regularization significantly improves accuracy on categorical datasets. But this is not enough, because there are lots of real-life applications in which real-valued attributes are provided. So we would like to extend our conclusion to real-valued datasets.

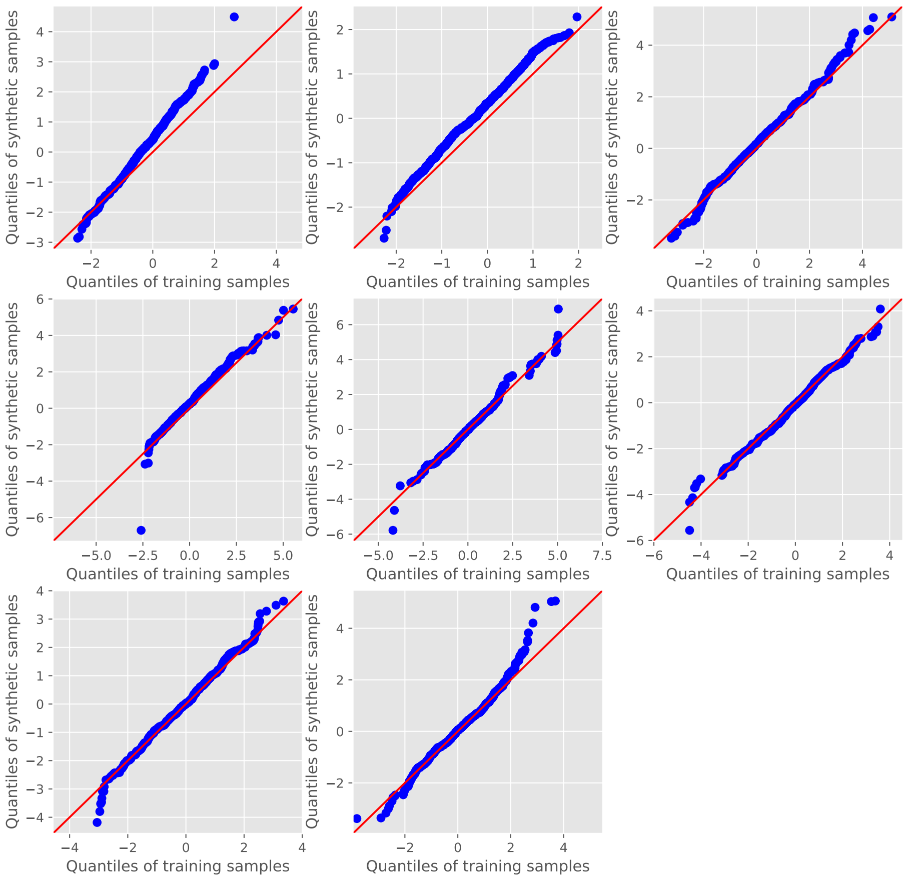
We discussed two models for modeling conditional distributions of real-valued features in the previous chapter. One is MDN (Mixture Density Network), the other is GAN (Generative Adversarial Network).

If the synthetic data distribution and the training data distribution fit well, their one-dimensional marginals should fit well. Although the reverse is not true, it is still an intuitive way of examining how good our models are. This comparison can be achieved by QQ (Quantile-Quantile) plots.

We present the 8 QQ plots corresponding to 8 attributes of the HOUSES dataset below. The plots are provided for both MDN and GAN.

From only the information provided by QQ plots, MDN is a better model, as all

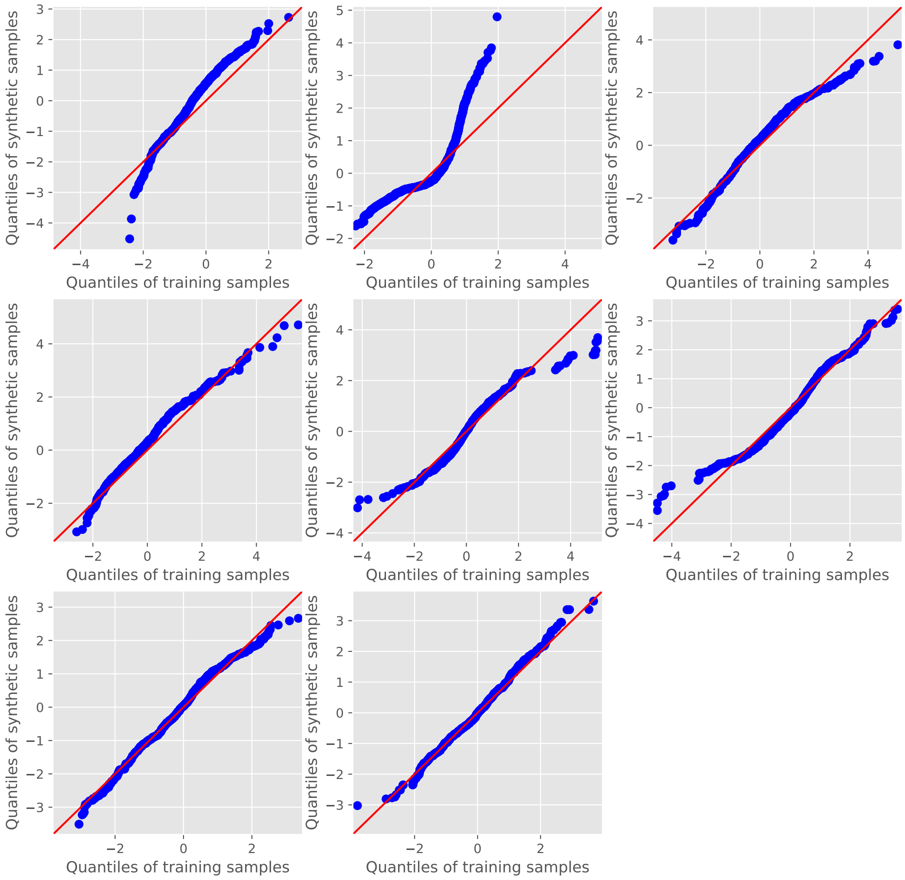
Figure 4.2: QQ plots of marginals (a comparison of training data and synthetic data generated by MDN)



the marginals fit very well. While on the other hand, GAN has marginals that deviate far from the actual distribution, for example the 2nd feature (second column of the first row in figure 4.3). To investigate the problem, we also provide the histogram for the 2nd feature in figure 4.4. The blue area is the histogram for synthetic data and the red one is the histogram for training data. It seems that the problem is, GAN only captures on peak of the training data, and fails to cover another.

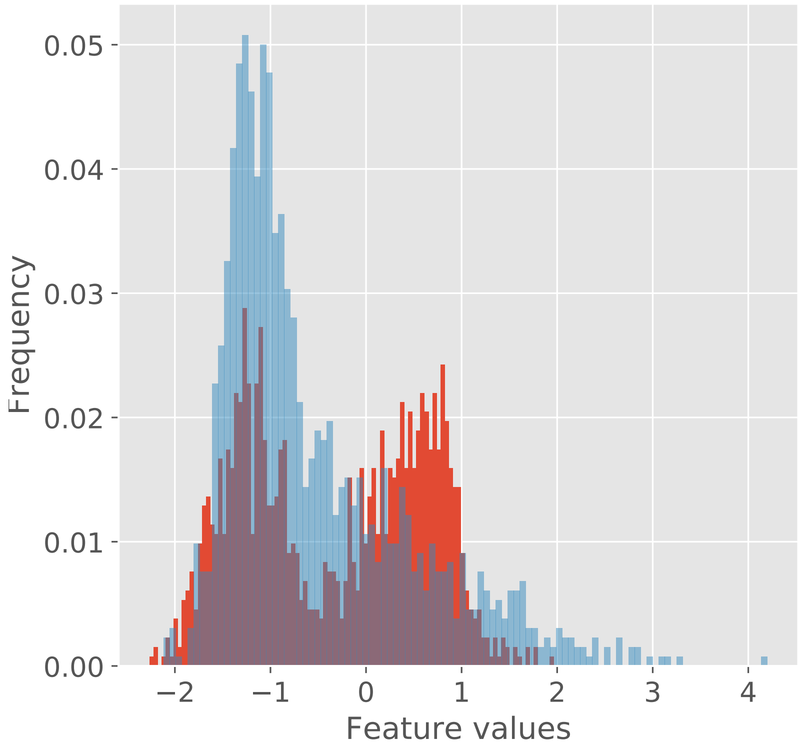
For the following parts, we consider MDN as the default model for real-valued features and give up GAN for now. It is not because we think it underperforms MDN. We have no clear evidence to say so. GAN is a very new model, and has been applied and studied mainly for image and sequence data. Therefore, it is trick to train a GAN

Figure 4.3: QQ plots of marginals (a comparison of training data and synthetic data generated by GAN)



for our problem, and many heuristic techniques must be carefully applied in order to make it work. So let's not draw any conclusions about GAN here before it is thoroughly studied. We save it for future work, and hopefully it can be a good candidate for real-valued conditional distribution models.

Figure 4.4: Histogram of the marginal for 2nd feature of HOUSES



(a) The histogram is a normalized version.

(b) The blue area corresponds to synthetic data generated by GAN, and the red area corresponds to training data.

4.5 An extension to real-valued datasets

In this section, we use MDN (Mixture Density Network) as conditional distribution models for real-valued attributes, and see if the previous conclusions on categorical datasets still apply.

As shown in table 4.4, synthetic data regularization improves accuracy of gradient boosting, on all the datasets with real-valued features. (On the HOUSES dataset, our model with synthetic data loses to the SVM model because SVM performs particularly well on this task, but SVM is not suitable for synthetic data regularization because it cannot scale well).

However, the improvement is not as significant as those on the categorical datasets. We give two possible explanations for this phenomenon. On the one hand, how much we can improve is limited by the asymptotic performance that would be obtained by an infinite training set. For example, on the RNA dataset, a big improvement was observed even though the training set size was doubled. On the other hand, as stated earlier, the performance heavily depends on which conditional distribution model we are using. In the categorical cases, gradient boosting has been proven to be a very good conditional distribution model. For real-valued cases, only MDN is tested. It is possible that we can find a better conditional distribution model for real-valued features, and eventually get considerable performance improvement.

Table 4.4: Applying synthetic data regularization to datasets with real value features

Dataset	**RNA**	**COVTYPE**	**HOUSES**
Logistic Regression	93.67%	75.40%	0.5898
SVM	93.96%	79.64%	0.6740
RF (Random Forest)	93.75%	79.77%	0.6277
GB (Gradient Boosting)	94.23%	79.30%	0.6298
GB (train x 2) [2]	94.69%	81.82%	0.6678[1]
GB+Syn(B)[3]	94.48%	80.13%	-
GB+Syn(J)[4]	94.55%	79.31%	0.6588

[1] When we use a doubled training set to train the SVM, we got a R^2 score of 0.6917.

[2] Double the training set size using reserved data, it is an good comparison to the results when feeding synthetic data.

[3] Model each class separately and construct the generative model using Bayes' rule.

[4] Model the data jointly with the labels.

4.6 DIS measure on all datasets

As introduced in section 3.10, DIS an intuitive measure that tells us how accurate a discriminator can discriminate synthetic data from real data. It is a useful tool for debugging and reasoning. We use it a lot to check our implementation, and would like to recommend it if you want to develop new methods for synthetic data generation.

In this section, we calculate DIS on all datasets and describe some findings.

Table 4.5: DIS on all the datasets

Dataset	AD	RCV1	RNA	OTTO	COVTYPE	HOUSES
DIS (Gen, Train)	51%	53%	53.8%	62.4%	55.12%	61.10%
DIS (Gen, Valid)	55.7%	59.4%	55.65%	62.24%	56.18%	60.20%

We use gradient boosting as the discriminator, and calculate the measure on all datasets. Both training set and validation set are used as the true data.

As you can see in table 4.5, on most datasets, the measure is quite close to 50%, which is an indication that our approach is doing very well on synthetic data generation. We can also see that, the DIS using the validation set is very often higher than the counterpart using the training set, which tells us our model is slightly over-fitting. It is reasonable, as all parameterized models will be over-fitting to some extent.

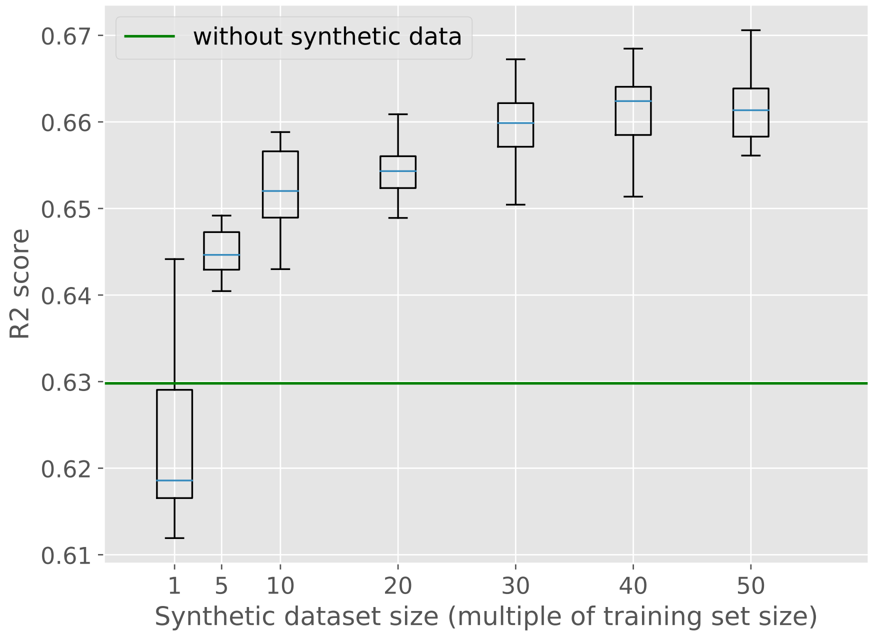
4.7 How much synthetic data should we use?

When we apply synthetic data regularization, we wonder how much synthetic data we should feed into the final estimator. In this part, we use a simple experiment to answer this question. We apply synthetic data regularization to gradient boosting. We tune the amount of synthetic data from 1 to 50 times as many as the training data. The way of feeding these synthetic data sticks to the statements in 4.1.6. 10 independent experiments are conducted for each synthetic dataset size. The results are shown in figure 4.5 in a form of box plot.

From the figure, we have two main take-away points:

- As long as our computing resources permit, the more synthetic data the better.
- If very few synthetic data are used, the variance of performance may be big, and we also have the danger of underperforming the counterpart without synthetic data.

Figure 4.5: Performance against synthetic data size



(a) For each synthetic dataset size, 10 independent experiments are conducted. The 10 R^2 scores are then used to construct the box plot.

(b) The green horizontal line indicates where we are if no synthetic data regularization is applied. As you can see, if only one round of synthetic data is used in this example, we have the danger of getting even worse results.

4.8 An empirical comparison to other approaches

All generative models can provide synthetic data for regularization. Previously we claim that conditional distributions are much easier to model than a joint distribution. It would be interesting to see, in terms of regularization, is the dependency network a better generative model compared to some joint distribution models?

In this section, we make use of synthetic data generated from joint distribution models by feeding them into the final estimator together with training data, just like what we did with the dependency network model. The performance of the final estimator with synthetic data regularization is compared to the results in previous experiments. For categorical datasets, we use a RBM (Restricted Boltzmann Machine) model to model the joint distribution, which is considered as a state-of-the-art method. For real-valued cases, GMM (Gaussian Mixture Model) is adopted, which is a benchmark model with high-performance.

Table 4.6: A comparison of dependency network and joint distribution models (RBM and GMM)

Dataset	**AD**	**RCV1**	**RNA**	**HOUSES**
Feature type	categorical	categorical	real	real
Joint distribution model	RBM	RBM	GMM	GMM
GB (Gradient boosting)	94.13%	85.69%	94.23%	0.6298
GB+Syn (Joint)[1]	95.55%	86.34%	**94.72%**	**0.6704**
GB+Syn (DN)[2]	**97.34%**	**86.62%**	94.55%	0.6588

[1] Synthetic data is generated from the corresponding joint distribution model.

[2] Synthetic data is generated from the dependency network. (The default conditional distribution model is gradient boosting for categorical features, and MDN for real value features)

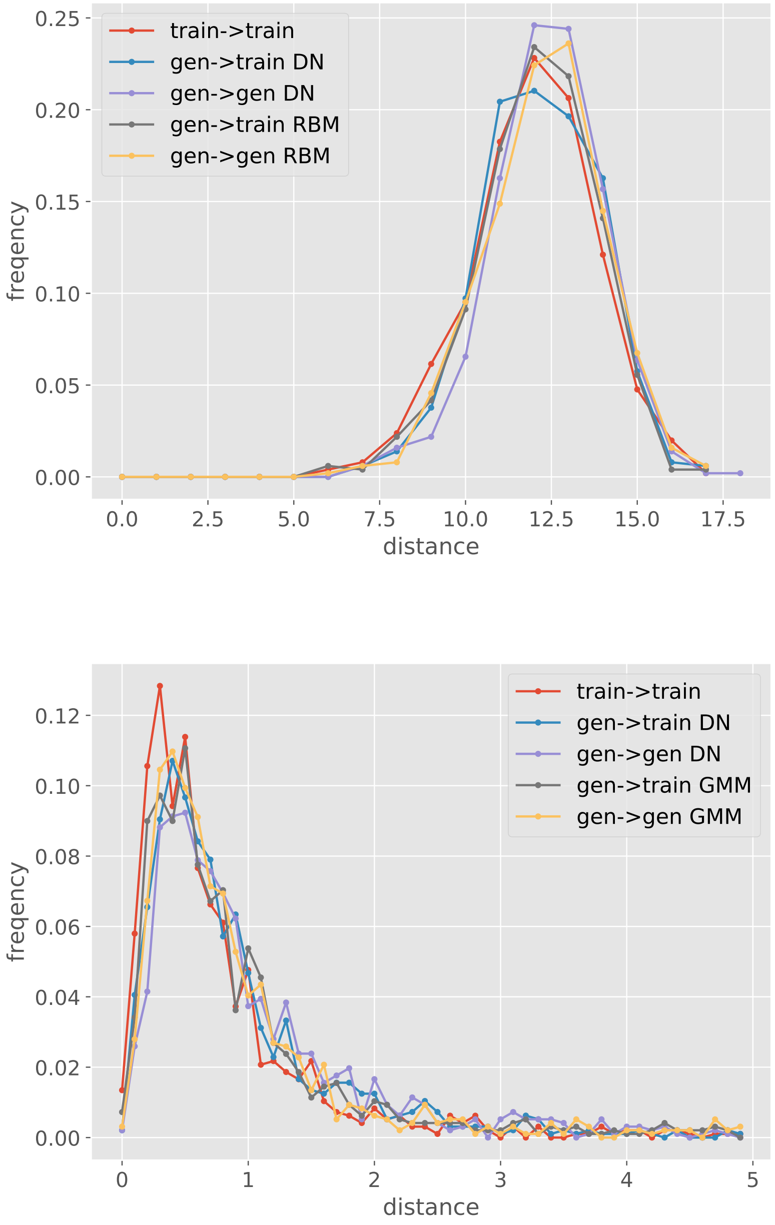
As shown in table 4.6, the dependency network model outperforms RBM on both categorical datasets, but loses to GMM on both real-valued datasets. The fact corresponds to our previous conclusion that we have very good categorical conditional distribution models, but need to find better real-valued conditional distribution models. The experiments are not conducted on the other two datasets because we don't want to introduce complexity for dealing with mixed attribute types and counts data.

We also calculate the NLL (Negative Log Likelihood) of synthetic data, which generated by RBM, on the AD dataset. The NLL of the first round data is 35.54 (here one round means a block Gibbs sampling step of sampling hidden variables given the inputs, and then another step of input variables given hidden variables generated by the previous step.). If you compare it to the results in table 4.7, it is better than the model random forest conditional distribution model, but worse than all the others. As indicated by NLL, the dependency network with high-performance conditional distribution models is actually a better generative model than RBM on this dataset.

Table 4.7: NLL of the first round synthetic data on AD dataset

Conditional models	**NLL**
SVM	34.84
Neural network	34.98
Neural network (parameters sharing)	34.66
Random forest	35.81
Gradient boosting	33.92

Figure 4.6: Distance between generated data and training data



(a) The upper figure shows results on the AD dataset, and hamming distance is used. The lower figure shows results on the HOUSES dataset, and Euclidean distance is used.

(b) DN stands for Dependency Network, RBM stands for Restricted Boltzmann Machine, and GMM stands for Gaussian Mixture Model.

(c) When we say gen→gen, the generated data on the two sides are actually from different sampling runs.

4.8.1 How “close” are synthetic data and training data?

Given a synthetic data generating approach, we want to know whether the generated synthetic data are very similar to the original training data. Because if that is the case, our model may be heavily overfitting to the training data. Or in our approach, maybe it is simply because we did not run Gibbs sampling for enough steps.

In order to measure how “close” two datasets are, we need to first measure the distance between data points, and then the distance between a dataset and a data point. For categorical features, hamming distance is adopted as point-to-point distance, and for all numerical features, Euclidean distance is adopted point-to-point distance. Established on the definition of point-to-point distance, the distance between a datatset and a point would be the minimum distance between this point and all the points in that dataset (except that point). In fact, if that data point is taken from another dataset B (let’s call the original dataset A), we would be able to calculate the point to dataset distance for each point in B. In that case, we obtain a distribution of distances, which gives a good illustration of how “close” these two datasets are.

The distance distribution is interesting to see because if we have a generative model that is actually over-fitting the training data, then the distance from generated data to the training data will probably be smaller than usual. So when we plot this distribution, we will observe a left shifting of density to 0 compared to the distance from training set to training set itself.

In figure 4.6, we can see that all our generative models, no matter the dependency network, or the joint distribution models, do not show a sign of seriously over-fitting. The generated samples are quite new, when compare to the training set. On the other hand, the distance distribution gen$\rightarrow$gen is also close the train$\rightarrow$train. As stated under the figure, gen$\rightarrow$gen measures how close the samples from two different runs are. So the results tell us that we can repeatedly sample from our models, and obtain synthetic data that are quite different from each other.

Chapter 5

Empirical Techniques

The last chapter has shown that synthetic data generated by our approach can be used as regularization in various tasks. The chapter below will move on to describe some empirical techniques that could be useful in some circumstance when we apply the synthetic data regularization. Some of these techniques do not show clear evidence that they will improve performance in practice, but still, interesting phenomena have been observed. We present experiments results here, and have a general empirical guideline at the end of this chapter.

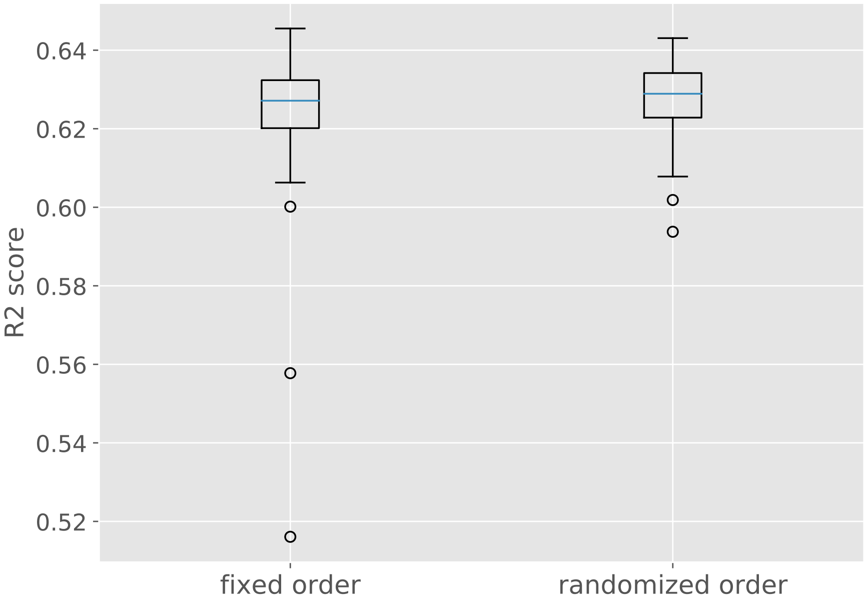
5.1 Randomized sampling order

As introduced in section 3.8, when we run sampling, a fixed order could be determined at the beginning of each run, or we could have different sampling orders for each individual initial samples (we call randomized sampling order in this thesis).

In this section, we would like to explore differences between the two. The conjecture is, on average, there should be no performance difference between the two. However, if a randomized sampling order is adopted, we do not rely on a particular order that is generated at the beginning. Thus we could avoid some unexpected bad results due to a bad choice of sampling order, which totally relies on random number generators.

In order to test our hypothesis, we design an experiment in such a way that only one round of synthetic data is generated, and is fed into the final estimator together with training data. We run the experiments for 100 times for both the case of a fixed order and the case of a randomized order. To clarify the definition of a fixed order, the order is fixed means all the samples share the same order in each run. But the orders in 100 experiments are different from each other as they are randomly generated at the beginning of each run. Moreover, it is also worthwhile to point out that the generative model is the same for all the experiments, because it is the randomness of sampling order that we care about here.

Figure 5.1: R^2 on the HOUSES dataset (randomized order versus fixed order)



(a) One round of synthetic data is generated and fed into our estimator together with training data (here the synthetic dataset has the same size of training set, please notice that before we normally use more synthetic data than training data by running sampling many times.). We run the experiment for 100 times, and record the R^2 score for each of them. The left box plot shows the situation when a fixed order is determined at the beginning of each run, while the right box plot shows the situation when a randomized sampling order is adopted.

(b) We denote Q_1, Q_2, Q_3 (from small to big) as quartile values. The box extends from Q_1 to Q_3, with a line Q_2. The whiskers extend from $Q_1 - IQR * 1.5$ to $Q_3 + IQR * 1.5$, where $IQR = Q_3 - Q_1$. Outliers are those data points which past the end of the whiskers.

Now we have 100 R^2 scores for both sampling order options, and we show box plots in figure 5.1 for comparison. From the figure, the medians of the two are very close. Overall, we cannot draw conclusions about which one is better even on this particular dataset. However, when a fixed order is adopted, we can observe a few outliers showing extremely bad performance. This does not happen for the randomized order. To sum up, at least on this particular HOUSES dataset, the results accord with our conjecture.

We also try to use the DIS measure to tell if the randomized order has advantages. The synthetic data distributions will be slightly different from each other when sampling orders are different. Intuitively when a randomized order is adopted, it is like mixing all the sampling orders in one run. So the hypothesis is, the synthetic data will be harder to discriminate when the randomized sampling order is adopted.

Table 5.1: A comparison of fixed order and randomized order in terms of DIS

Dataset	RNA	COVTYPE	HOUSES
DIS (Gen, Train), fixed order	53.8%	55.12%	61.10%
DIS (Gen, Valid), fixed order	55.65%	56.18%	60.20%
DIS (Gen, Train), randomized order	52.40%	55.5%	58.30%
DIS (Gen, Valid), randomized order	54.2%	57%	58.1%

However, we cannot draw conclusions confidently according to the results in table 4.5. On the RNA and HOUSES datasets, the synthetic datasets are indeed closer to the training set and validation set, when using a randomized order. But this is not true on the COVTYPE dataset, actually, the DIS measure is even slightly higher.

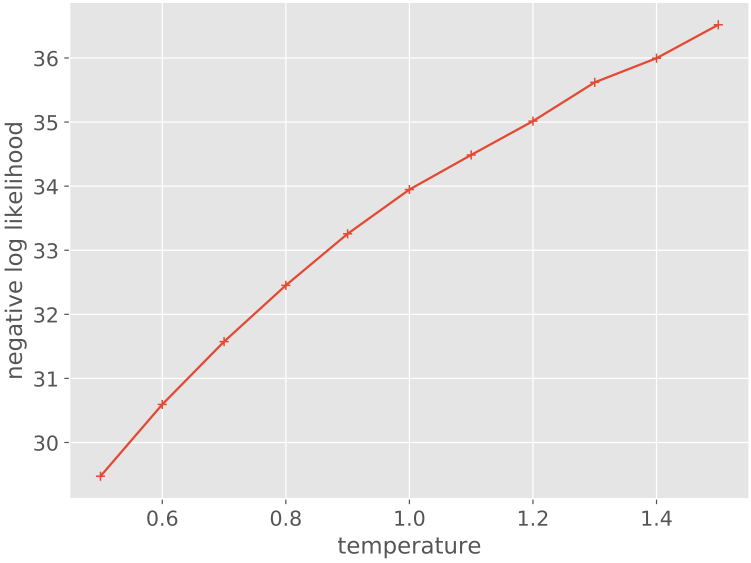
5.2 A study of sampling temperature

As discussed in section 3.9, when NLL is used as the error function, we could end up with a broad model that spreads out probability mass just to cover some outliers. To avoid that, the notion of temperature could be introduced when sampling from the model. The definition of sampling temperature is already provided in section 3.9.

To demonstrate the effect of temperature on the generated samples, we plot the NLL against temperature in figure 5.2. All the data are taken at the end of the first sampling round.

The figure clearly shows that smaller temperature leads to higher NLL. The observation is in accord with our intuition. When the temperature is small, we favour the area with high probability density more than usual, so that more samples are generated from these high density area. Thus we push up the NLL.

Figure 5.2: NLL of synthetic data against temperature on the AD dataset

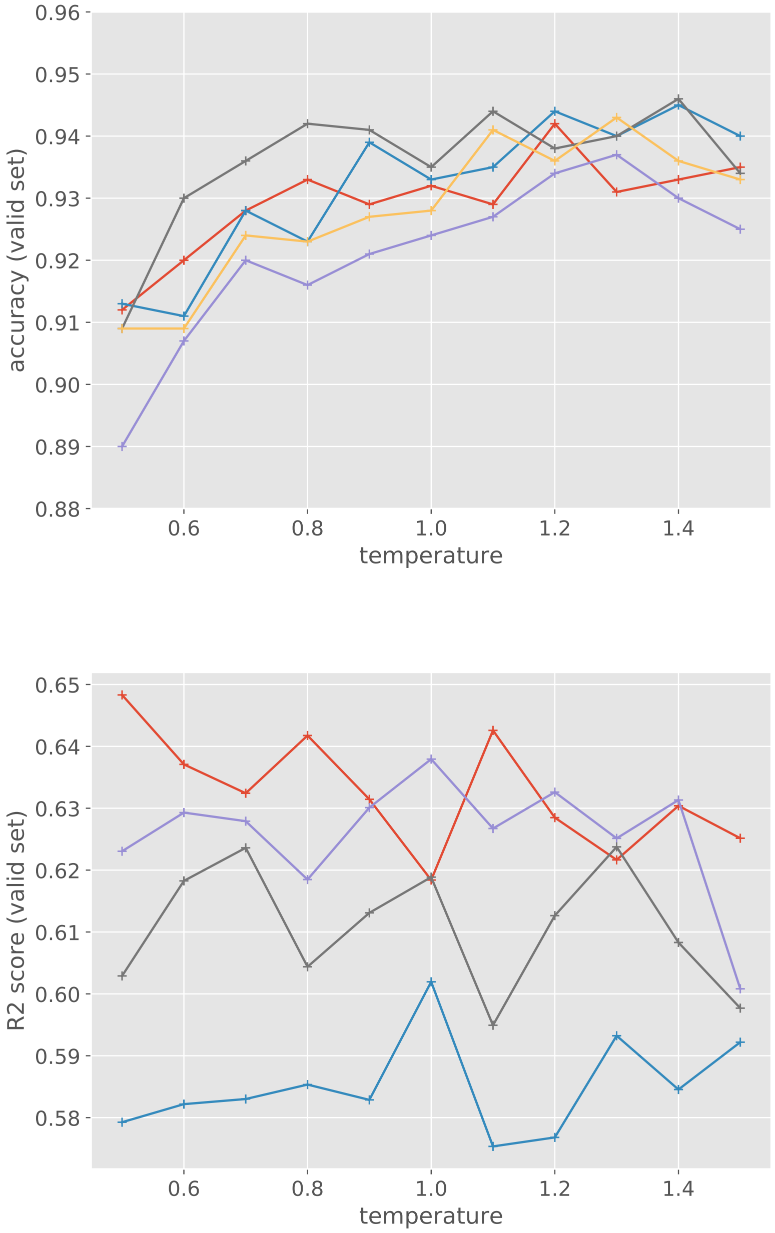


Again, we should clarify that high NLL does not necessarily mean better regularization performance. A natural question we want to ask is, should we introduce the temperature as another hyper-parameter for our methods? Using synthetic data generated under different temperature, and feeding them into the estimators, we report performance of the final task in figure 5.3. We deliberately make our training dataset smaller, by only use 20% of training data. Also, because the results can depend on which training set we are using, we run independent experiments using all the 5 training subsets. In figure 5.3, different curves correspond to different runs with different training subsets. On the AD dataset, in all runs, a temperature of 1.2 or 1.3 is always better than a temperature of 1.0. Therefore, we have the reason to believe that a temperature of 1 is not the best setting, and tuning the temperature as a hyper-parameter can improve the regularization performance in that task. However, this is not the case when it comes to the HOUSES dataset. There is no obvious trend in all the curves.

It is worthwhile to point out again that, the definition of temperature in these two examples are different. Because adding temperature to the mixture of Gaussian will make computation intractable, we actually use a heuristic definition (see section 3.9.3) which may not be right. It is still unclear if there exists a better computational tractable approximation to the exponent of mixtures.

Is setting a higher temperature on the AD dataset indeed a better idea? The answer may be negative. Previously, we conduct all the experiments on the validation set. Now let's try temperature $1.0, 1.1, 1.2, 1.3$ on the test set, and report the accuracy using synthetic data regularization in figure 5.4. Because the test set is much bigger, the sampling error is significantly reduced. Now you can see that the curves become quite flat, and we cannot say there exists a better temperature than 1.

Figure 5.3: Performance (accuracy and R^2) when feeding synthetic data together with training data against sampling temperature on the AD dataest and HOUSES dataset

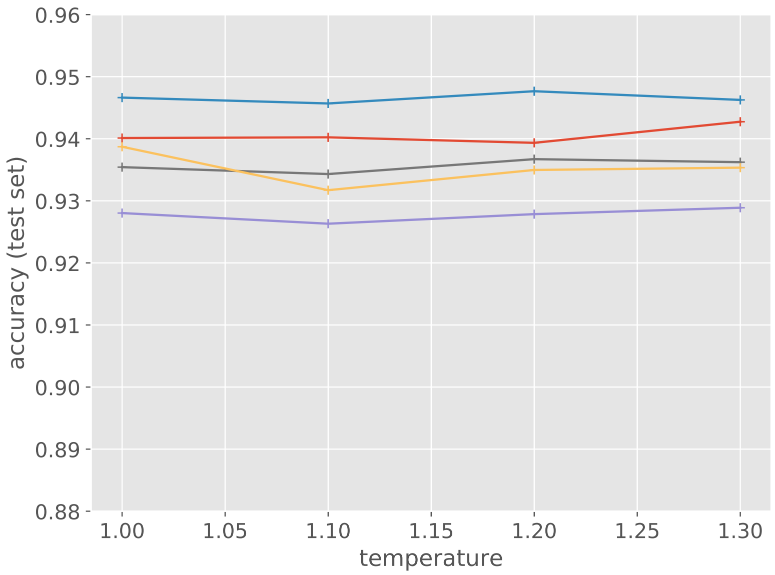


(a) Each curve corresponds to a different training set, and for each training set, different temperatures are tested.

(b) The training set size is only 20% of the original one, in order to imitate the situation when the generative model is untrustworthy.

(c) All the results in this figure is on validation sets.

Figure 5.4: R^2 performance on the AD test dataset



5.3 Pseudo importance sampling using a discriminator

Obviously, no matter how good our model is, there will always be discrepancy between the true underlying distribution $P_{true}(\boldsymbol{x}, y)$ and the synthetic data generating distribution $P_{gen}(\boldsymbol{x}, y)$. Sometimes, the discrepancy is big if the data is particularly hard to model. Theoretically, we are sampling from the synthetic data distribution, and try to estimate expectations on the actual data distribution. So if we know exactly about the two distribution, importance sampling could be used. Unfortunately, there is no way to know the distributions. But it is possible for us to do something similar: We could construct a binary classification task to tell whether the samples are from the actual distribution $P_{true}(\boldsymbol{x}, y)$ or from the synthetic data distribution $P_{gen}(\boldsymbol{x}, y)$. The discriminator assigns probability $p(\boldsymbol{x}, y)$ to each synthetic data point, estimating how likely this is a real sample instead of a synthetic one. If we fully trust this discriminator, then we have

$$p(\boldsymbol{x}, y) = \frac{P_{true}(\boldsymbol{x}, y)}{P_{true}(\boldsymbol{x}, y) + P_{gen}(\boldsymbol{x}, y)}. \tag{5.1}$$

So if we want to estimate

$$\mathbb{E}_{(\boldsymbol{x}, y) \sim P_{true}(\boldsymbol{x}, y)} \left[f(x, y) \right], \tag{5.2}$$

but only have samples drawn from $P_{gen}(\boldsymbol{x}, y)$, instead we could estimate

$$\mathbb{E}_{(\boldsymbol{x}, y) \sim P_{gen}(\boldsymbol{x}, y)} \left[\frac{p(\boldsymbol{x}, y)}{1 - p(\boldsymbol{x}, y)} f(x, y) \right]. \tag{5.3}$$

In the application of regularization, $p(\boldsymbol{x}, y)$ is assigned to each synthetic data point by the discriminator, and we re-weight sample weights in the final estimator by multiplying a coefficient of $\frac{p(\boldsymbol{x}, y)}{1 - p(\boldsymbol{x}, y)}$. Theoretically, this technique gives us the chance to put less weights on those synthetic samples we do not trust, and increase the weights on more promising samples.

Table 5.2: Re-weight samples with pseudo importance sampling

Dataset	OTTO	HOUSES
GB + Syn	79.49%	0.6588
GB + Syn + Imp[1]	79.19%	0.6614

[1] The synthetic data is re-weighted by pseudo importance sampling.

Intuitively, this technique may be useful when our generative model is not good enough. So we refer to table 4.5 and pick the OTTO and HOUSES datasets which have the largest DIS accuracy. In table 5.2, pseudo importance sampling is applied to re-weight the sample weights on both datasets. On the HOUSES dataset, there is a slight improvement, but it is not true on the OTTO dataset. Therefore, we cannot justify our hypothesis through this experiment, and further investigations are needed.

5.4 Creating diversified ensemble using synthetic data

Previously, the amount of synthetic data is K times as many as the training data. When applying synthetic data regularization, all the synthetic data are fed into the final estimator together with training data. To avoid synthetic data becoming overwhelming, sample weights are used to balance the two, so that training data and synthetic data have equal total weights.

Table 5.3: Creating an ensemble using synthetic data

DataSet	AD	RCV1	RNA	OTTO	COVERTYPE	HOUSES
SVM+Bagging	93.18%	85.16%	94.22%	75.54%	79.36%	**0.6734**
RF+Bagging	93.02%	87.12%	93.61%	77.29%	80.22%	0.6309
GB+Bagging	93.64%	86.26%	94.37%	78.55%	79.55%	0.6598
GB+Syn[1]	**97.34%**	86.61%	94.48%	79.49%	80.13%	0.6588
SVM+Syn Ensemble	95.34%	85.45%	94.43%	75.72%	79.53%	0.6598
RF+Syn Ensemble	93.74%	**87.33%**	94.05%	77.83%	**80.69%**	0.6415
GB+Syn Ensemble	95.25%	86.92%	**94.58%**	**79.71%**	80.52%	0.6658

[1] Results in this row are taken from table 4.3 and table 4.4, where all synthetic data is fed into one estimator.

In this section, we adopt a different way of using synthetic data. Instead of feeding all the synthetic data into just one estimator, K estimators are created and form an ensemble when predicting. The same amount of synthetic data as training data is used in each estimator. In table 5.3, we compare performance using this kind of ensemble to the previous versions. In addition, since we now benefit from ensemble methods, it would be unfair to compare our results to the performance of a single estimator. So we also create ensembles without synthetic data using bagging[4] and compare them to our methods. On 4 of all the 6 datasets, ensembles using synthetic data achieve the best performance so far. However, it is not our intention to creating a whole ensemble just to achieve a slight improvement. This technique is attractive because of the following reasons:

- There are methods or implementations that cannot deal with sample weights well, for example, the *sklearn* implementation of random forest. In that case, synthetic data will be overwhelming if they are fed into one estimator. We bypass this issue by creating the ensemble, so that synthetic data regularization can be applied to all the methods.

- There are methods that cannot scale well, for example SVM. So creating an ensemble will be much faster than feeding all the synthetic data into just one estimator.

- The algorithm is now more suitable for parallel computing by nature.

5.5 Auxiliary features

When we feed synthetic data together with training data into one estimator, the estimator has no access to the information of whether a particular sample is synthetic. What if we add an auxiliary feature to indicate whether this sample is actually generated, instead of coming from the training set? It is potentially a good idea because now the estimator is given the opportunity to decide whether to trust these synthetic data.

To be more formal, assuming the original dataset is $D = \{(\boldsymbol{x_n}, y_n)\}_{n=1,2,\dots,N}$, and $I(\boldsymbol{x}, y)$ is an indicator of whether this sample is a synthetic one. Using $I(\boldsymbol{x}, y)$ as an auxiliary feature, now the dataset becomes $D = \{(\bar{\boldsymbol{x_n}}, y_n)\}_{n=1,2,\dots,N}$, where $\bar{\boldsymbol{x}} = [\boldsymbol{x}, I(\boldsymbol{x}, y)]$. During test time, $I(\boldsymbol{x}, y)$ will always be set to 0.

We introduce this auxiliary feature on the AD dataset and HOUSES dataset. At least on these two particular task, no clear improvements can be observed.

Table 5.4: Use auxiliary feature to indicate whether the sample is a synthetic one

Dataset	AD	HOUSES
GB	94.13%	0.6298
GB+Syn[1]	**97.34%**	**0.6588**
GB+Syn+Aux[1]	97.05%	0.6555

[1] Results in this row are taken from table 4.3 and table 4.4.

[2] One auxiliary feature is used to indicate whether the sample is a synthetic one. The feature will be set to 0 during test time.

5.6 Discretizing real value features

As discussed in the last chapter, our synthetic data regularization performs much better on the categorical dataset than on the real value dataset. A natural idea would be, can we just convert these real value features into discrete ones and model the conditional distributions just like those for the categorical features? A typical way of conversion is to discretize the values by splitting at quantiles, thus get ordinal features instead of real values. We further apply one-hot encoding to these ordinal features and turn them into categorical features. By discretizing, we lose information, but as we have much better conditional distribution models for categorical features, it may be worthwhile to try.

We test this technique on all our real value datasets, and show results in table 5.5. Here deciles calculated from training data are used as split points. Categorical features are only used for our generative models, and we will convert them back to ordinal values when training the final estimator. That is because the relative order of these values is important for these tasks. Unfortunately, this technique did not improve performance. The results turn out to be worse. However, we argue that this is not because the information we lose by discretization, at least not the main reason. If discretization is what should be blamed, then the second row in table 5.5 should be much worse than the first row, and no significant improvement can be observed on the third row. Clearly this is not the case. The results actually tell us, if we have more data, even though the feature values are discretized, we can still achieve high performance.

Table 5.5: Discretizing real value features[2]

Dataset	RNA	COVTYPE	HOUSES[3]
GB	94.23%	79.30%	0.6298
GB + Discretized[1]	93.68%	79.36%	0.6126
GB + Discretized (train x 2)[4]	93.83%	81.35%	0.6456
GB + Discretized + Syn	93.56%	78.62%	0.6110

[1] Each real value features are divided by deciles calculated from training data.

[2] When we train the generative model, discretized features are processed by one-hot encoding. However, when predicting, these features are converted back to ordinal features.

[3] The HOUSES dataset is a regression task, so the targets are not discretized.

[4] Just as those in the previous tables, train x 2 means we double the training set size using reserved data.

A possible explanation we can provide is, although we have good categorical conditional distribution models, what we actually deal with are ordinal features. We give up the information of relative order when applying one-hot encoding to them. So probably we can achieve better performance using this technique, if a good conditional distribution model for ordinal features can be found.

5.7 Empirical guidelines

Many empirical techniques and procedures are introduced throughout the thesis, and some of them are worthwhile to try only under certain circumstances. In this part, we give a concise list of these techniques and summarize when they should be tried:

- Adding uniform noises to feature values (see section 4.1.2): It is recommended when feature values are integers or indirectly calculated from integers, to avoid putting too much probability mass on a single value.

- PCA (Principle Component Analysis) (see section 4.1.2): We recommend PCA for all the dataset with real value features, as it will normally make the modeling of conditionals much easier. When the attributes are of mixed types, PCA can be applied to the subset of attributes taking real values.

- Randomized sampling order (see section 5.1): It is recommended when we want to avoid extremely bad performance because of a poor sampling order choice. This technique is not important when synthetic data are taken from different runs and then combined together.

- Pseudo importance sampling (see section 5.3): There is no clear evidence that it leads to performance improvement in our experiments. However, it is still worth trying if we are not confident about our generative model, and want to adjust weights on synthetic data.

- Creating an ensemble using synthetic data (see section 5.4): It is recommended when inference speed is not an important issue. It should be adopted when our final estimator cannot scale well or cannot deal with sample weights appropriately.

- Using auxiliary features (see section 5.5): It shows no improvement in our experiments, further investigations may be needed.

- Real value discretization (see section 5.6): We recommend to try it only when a good conditional distribution model for ordinal features can be found.

Chapter 6

Further Investigations

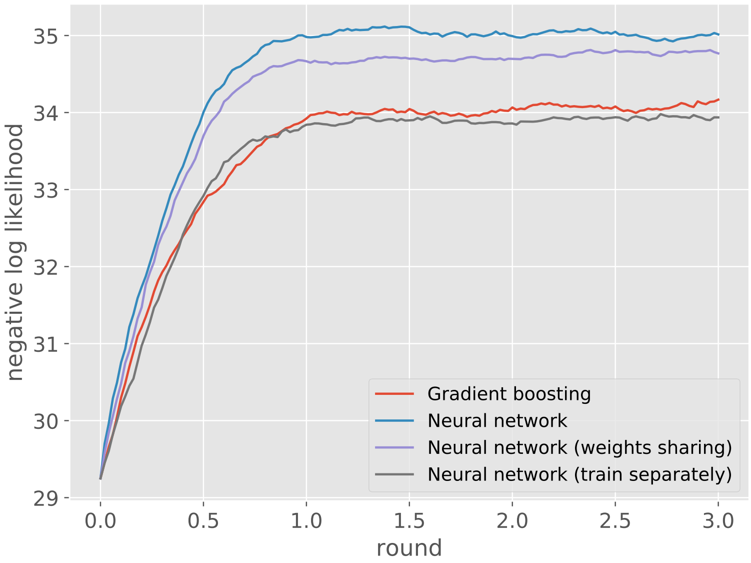
This chapter presents some results and discussions that do not concern the central arguments of this thesis, but are still very worthwhile to investigate. Only a few experiments are conducted, but the discoveries turn out to be quite interesting, and may lead to thorough investigations in the future.

6.1 An additional note on parameters sharing neural network architecture

In section 4.2, we compare various conditional distribution models for categorical features. There is an interesting phenomenon that neural network model with parameters sharing actually outperforms the model without parameters sharing, no matter in terms of negative log likelihood on the AD dataset, or in terms of final accuracy when synthetic data regularization is applied. As discussed in section 3.2.2, parameters sharing saves us lots of storage space and computing time. Consider all of these, at least on this particular dataset, parameters sharing architecture is a better option. There are also other models based on neural networks involves an architecture of parameters sharing, such as NADE [38]. So it is worthwhile to devote more efforts to it.

In this part, we investigate this phenomenon further. Before, even though parameters are not shared, we still use a global loss function based on pseudo likelihood (see 3.4), and all the neural network conditional distribution models share the same hyperparameters (number of layers, number of hidden units and so on.). All these constraints could be relaxed. In fact, we are going to make the training of all conditional distribution models totally independent. That is to say, each neural network has its own loss function, pick its own hyper-parameters through cross validation, and apply early-stopping separately. We plot the corresponding curve of NLL against sampling round just like those in figure 4.1, and compare it to other neural network architecture and gradient boosting in figure 6.1. As you can see, now our neural network model become the model with lowest final NLL, even slightly lower than gradient boosting, which is previously considered as the best model for categorical features.

Figure 6.1: Negative log likelihood against sampling round on the AD dataset (A more flexible neural network model is explored)



However, low NLL does not mean good classification performance when we use the model to generate synthetic data and train on them. We use the model described above in synthetic data regularization, and achieves an accuracy of 95.45%, which is worse than the original neural network model without parameters sharing (95.56%).

6.2 Other usages of synthetic data: knowledge distilling

Although we mainly focus on regularization so far when demonstrating the application of our synthetic data generating approach. There is no particular reason why we should limit the usage of the generated synthetic data.

In this section, we demonstrate an example of using generated synthetic data for knowledge distilling [6][21] (or namely model compression, teacher-student model) on the HOUSES dataset.

At first, we would like to provide a brief introduction of knowledge distilling:

To be concise, we simplify a general machine learning problem as an approximation to an underlying function $f(\boldsymbol{x})$ using a parameterized function, where $\boldsymbol{x}$ is the inputs. If we have a complex high-performance model $f_A(\boldsymbol{x};\theta_A)$, and a simple model $f_B(\boldsymbol{x};\theta_B)$. If$f_B$ is flexible enough, we could let f_B learn from f_A so that f_B can possibly achieve performance close to f_A. Since f_B is a simple model, the computational cost for inference is low and the storage space for parameters is small. By compressing the knowledge of f_A into f_B, we acquire a much simpler model which outperforms in terms of computing time and storage, but still achieve similar performance for prediction.

As examples, we can distill the knowledge from a very deep neural network into a shallow neural network, or from an ensemble of estimators into a single estimator. Some online applications need fast inference, and only limited storage space can be provided in devices like mobile phones. That is why knowledge distilling is often a very useful method.

For a general machine learning problem, we need synthetic data in order to let f_B learn from f_A. That is because our optimization objective is, assuming the mean square error is used:

$$\min \mathbb{E}_{\boldsymbol{x}\sim P(\boldsymbol{x})}\left[(f_B(\boldsymbol{x};\theta_B) - f_A(\boldsymbol{x};\theta_A)^2\right], \tag{6.1}$$

where $P(\boldsymbol{x})$ is the distribution of inputs, which is normally unknown.

So instead, we should get enough samples from $P(\boldsymbol{x})$, and use them as “teaching examples”. For a general task, using only training data as examples is far from enough, and this is where a synthetic data generating approach comes into play.

In [6], a pragmatic approach called MUNGE is used to generate synthetic data. However, this approach cannot scale well, because we need to re-calculate distance between data points at every update. So the computing time will grow quadratically along with the training set size.

In this part, we use the HOUSES dataset as an example, and compress an ensemble of SVM (which is the best model we have on this dataset) into a single estimator like linear regression, random forest or gradient boosting.

As shown in table 6.1, the performance of a single random forest model and a gradient boosting model is largely improved after distilling and has a performance that is close to the teacher model. Unsurprisingly, we cannot compress our model into a

Table 6.1: Knowledge distilling using synthetic data on the HOUSES dataset

Estimator	**Linear regression**	**Random forest**	**Gradient boosting**
R^2	0.5898	0.6277	0.6298
R^2 after distilling[1]	0.5680	0.6588	0.6682
R^2, SVM+Bagging		0.6727	

[1] Linear regression, random forest and gradient boosting are student models, while SVM+Bagging is the teacher model. Synthetic data generated by our approach is used for knowledge distilling, instead of MUNGE in [6].

linear regression, as it is seriously under-fitting.

Chapter 7

Conclusions and Future Work

In this thesis, we develop a synthetic data generating approach based on the dependency network model. We study all kinds of choices for conditional distribution models, and explore various parameters and options for both modeling and sampling. A broad study of empirical techniques is also conducted. In this chapter, we summarize the main points of this thesis and highlight some directions for future work.

7.1 Conclusion

First of all, although the conditional distribution models in a dependency network could be inconsistent with each other, our approach works well in practice. we obtain synthetic data which reflect the true underlying distribution well through sampling from the model, and our model does not show a sign of over-fitting to the training data. At least on the tested datasets, the sampler seems to reach an equilibrium distribution quickly.

Secondly, regarding the conditional distribution models for categorical features, GB (Gradient Boosting) outperforms other models, no matter in terms of the likelihood of generated data, or in terms of synthetic data regularization performance, or in terms of computational efficiency. (We later show in 6.1 that neural network could be comparable to GB if they are trained separately, but computational cost is still much bigger.) In the case of real value features, MDN (Mixture Density Network) works fairly well, and shows a more stable performance than GAN (Generative Adversarial Network). However, we believe that GAN could perform better if reasonable architectures and appropriate training tricks could be thoroughly studied.

Thirdly, we tested our hypotheses on various datasets with limited training data.

With GB as the conditional distribution model, our synthetic data regularization works very well on categorical datasets. If training data is insufficient, our approach can contribute significant performance improvement. On the other hand, conditionals of real value features seem to be much harder to model. With MDN as the conditional distribution model, normally we can only observe a small improvement on the performance.

In addition, we give an empirical comparison between dependency network model and joint distribution models like RBM and GMM. On categorical datasets, our model clearly outperforms RBM as far as regularization is concerned. However, on real value datasets, our model still loses to GMM.

What's more, many empirical techniques are proposed throughout chapter 4 and chapter 5, a detailed summary of conclusions can be found in 5.7.

Finally, knowledge distilling can be conducted successfully with our synthetic data generating approach. If the computational cost of training the conditional distribution model grows linearly with the training set size N (which is normally the case), the overall computational cost of our approach is also linear with N. However, the cost of MUNGE is $O(N^2M)$ (as discussed in 2.5.3). In that sense, our approach is superior MUNGE when the training set is big.

7.2 Future work

Turning now to future work, our findings and analysis suggest the following directions that would be interesting and potentially rewarding to explore in the future:

- The performance of our approach heavily depends on the choice of conditional distribution models. While the results on categorical datasets are satisfactory, we still need to find a better model as conditional distribution models for real value features. We suggest more research could be conducted to improve GAN. Specifically, we would like to design a good neural network architecture for GAN, when using it as a conditional model in a general machine learning problem. We also need to explore the best practices in order to acquire stable performance.

- If a high-performance conditional distribution model for real value features is hard, then another direction would be discretizing the values by quantiles. However, we have already shown that, in order to make it work, we may need a better model to deal with ordinal features.

- For synthetic data regularization, we believe that giving our system a mechanism to discard or put less weights on untrustworthy synthetic samples is a promising direction to further enhance performance. Roughly speaking, our exploration on pseudo importance sampling and auxiliary features are examples of such efforts.

- The number of conditional distribution models grows linearly with the number of attributes, so for a problem with a large number of features, our approach will be unacceptably slow. So more efforts should be devoted to studying how to overcome this difficulty. Sharing parameters between models could be a starting point.

- Because we can define an arbitrary loss function for *XGBoost* [10] (an efficient implementation of gradient boosting), we could use it to estimate parameters of a GMM. So in fact, *XGBoost* can be used to model real-valued conditional distributions. We tried this idea, but the performance is bad. It's still unclear whether this is due to the *XGBoost* implementation, or gradient boosting is not suitable for this task. More work could be devoted, because potentially we could find a good conditional distribution model for real-valued features.

Bibliography

[1] ABADI, M., AGARWAL, A., BARHAM, P., BREVDO, E., CHEN, Z., CITRO, C., CORRADO, G. S., DAVIS, A., DEAN, J., DEVIN, M., GHEMAWAT, S., GOODFELLOW, I., HARP, A., IRVING, G., ISARD, M., JIA, Y., JOZEFOWICZ, R., KAISER, L., KUDLUR, M., LEVENBERG, J., MANÉ, D., MONGA, R., MOORE, S., MURRAY, D., OLAH, C., SCHUSTER, M., SHLENS, J., STEINER, B., SUTSKEVER, I., TALWAR, K., TUCKER, P., VANHOUCKE, V., VASUDEVAN, V., VIÉGAS, F., VINYALS, O., WARDEN, P., WATTENBERG, M., WICKE, M., YU, Y., AND ZHENG, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] BERGSTRA, J., AND BENGIO, Y. Random search for hyper-parameter optimization. *Journal of Machine Learning Research 13*, Feb (2012), 281–305.

[3] BISHOP, C. M. Mixture density networks.

[4] BREIMAN, L. Bagging predictors. *Machine learning 24*, 2 (1996), 123–140.

[5] BREIMAN, L. Random forests. *Machine learning 45*, 1 (2001), 5–32.

[6] BUCILU, C., CARUANA, R., AND NICULESCU-MIZIL, A. Model compression. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining* (2006), ACM, pp. 535–541.

[7] BUITINCK, L., LOUPPE, G., BLONDEL, M., PEDREGOSA, F., MUELLER, A., GRISEL, O., NICULAE, V., PRETTENHOFER, P., GRAMFORT, A., GROBLER, J., ET AL. Api design for machine learning software: experiences from the scikit-learn project. *arXiv preprint arXiv:1309.0238* (2013).

[8] CARREIRA-PERPINAN, M. A., AND HINTON, G. E. On contrastive divergence learning. In *Aistats* (2005), vol. 10, pp. 33–40.

[9] CHANG, C.-C., AND LIN, C.-J. Libsvm: a library for support vector machines. *ACM transactions on intelligent systems and technology (TIST) 2*, 3 (2011), 27.

[10] CHEN, T., AND GUESTRIN, C. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining* (2016), ACM, pp. 785–794.

[11] CHOLLET, F., ET AL. Keras. `https://github.com/fchollet/keras`, 2015.

[12] CUI, X., GOEL, V., AND KINGSBURY, B. Data augmentation for deep neural network acoustic modeling. *IEEE/ACM Transactions on Audio, Speech and Language Processing (TASLP) 23*, 9 (2015), 1469–1477.

[13] DEMPSTER, A. P., LAIRD, N. M., AND RUBIN, D. B. Maximum likelihood from incomplete data via the em algorithm. *Journal of the royal statistical society. Series B (methodological)* (1977), 1–38.

[14] FRIEDMAN, J. H. Greedy function approximation: a gradient boosting machine. *Annals of statistics* (2001), 1189–1232.

[15] FRIEDMAN, J. H. Stochastic gradient boosting. *Computational Statistics & Data Analysis 38*, 4 (2002), 367–378.

[16] GLOROT, X., AND BENGIO, Y. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics* (2010), pp. 249–256.

[17] GOODFELLOW, I., POUGET-ABADIE, J., MIRZA, M., XU, B., WARDE-FARLEY, D., OZAIR, S., COURVILLE, A., AND BENGIO, Y. Generative adversarial nets. In *Advances in neural information processing systems* (2014), pp. 2672–2680.

[18] GUNN, S. R., ET AL. Support vector machines for classification and regression.

[19] HECKERMAN, D., CHICKERING, D. M., MEEK, C., ROUNTHWAITE, R., AND KADIE, C. Dependency networks for inference, collaborative filtering, and data visualization. *Journal of Machine Learning Research 1*, Oct (2000), 49–75.

[20] HINTON, G. A practical guide to training restricted boltzmann machines. *Momentum 9*, 1 (2010), 926.

[21] HINTON, G., VINYALS, O., AND DEAN, J. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531* (2015).

[22] IOFFE, S., AND SZEGEDY, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning* (2015), pp. 448–456.

[23] JONES, E., OLIPHANT, T., PETERSON, P., ET AL. SciPy: Open source scientific tools for Python, 2001–.

[24] KINGMA, D., AND BA, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[25] KRIZHEVSKY, A., SUTSKEVER, I., AND HINTON, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (2012), pp. 1097–1105.

[26] LICHMAN, M. UCI machine learning repository, 2013.

[27] LIN, H.-T., LIN, C.-J., AND WENG, R. C. A note on platts probabilistic outputs for support vector machines. *Machine learning 68*, 3 (2007), 267–276.

[28] MIRZA, M., AND OSINDERO, S. Conditional generative adversarial nets. *arXiv preprint arXiv:1411.1784* (2014).

[29] MURRAY, I. Markov chain monte carlo.

[30] NABNEY, I. *NETLAB: algorithms for pattern recognition*. Springer Science & Business Media, 2002.

[31] NAIR, V., AND HINTON, G. E. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)* (2010), pp. 807–814.

[32] NICULESCU-MIZIL, A., AND CARUANA, R. Predicting good probabilities with supervised learning. In *Proceedings of the 22nd international conference on Machine learning* (2005), ACM, pp. 625–632.

[33] PACE, R. K., AND BARRY, R. Sparse spatial autoregressions. *Statistics & Probability Letters 33*, 3 (1997), 291–297.

[34] Platt, J., et al. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. *Advances in large margin classifiers 10*, 3 (1999), 61–74.

[35] Raviv, Y., and Intrator, N. Bootstrapping with noise: An effective regularization technique. *Connection Science 8*, 3-4 (1996), 355–372.

[36] Simard, P., Victorri, B., LeCun, Y., and Denker, J. Tangent prop-a formalism for specifying selected invariances in an adaptive network. In *Advances in neural information processing systems* (1992), pp. 895–903.

[37] Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: a simple way to prevent neural networks from overfitting. *Journal of machine learning research 15*, 1 (2014), 1929–1958.

[38] Uria, B., Murray, I., and Larochelle, H. A deep and tractable density estimator. In *ICML* (2014), pp. 467–475.

[39] Vincent, P., Larochelle, H., Bengio, Y., and Manzagol, P.-A. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning* (2008), ACM, pp. 1096–1103.

[40] Walt, S. v. d., Colbert, S. C., and Varoquaux, G. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering 13*, 2 (2011), 22–30.

[41] Wold, S., Esbensen, K., and Geladi, P. Principal component analysis. *Chemometrics and intelligent laboratory systems 2*, 1-3 (1987), 37–52.

[42] Xu, J. Informatics research proposal.