Utrecht University

Master Thesis

Feature mappings for performative predictions

Author:

Maciej Makowski *(1177729)*

Supervisors:

Dr.-ing. Habil. Georg Krempl
Prof. dr. Arno Siebes

External supervisor:

MSc Business Informatics
Graduate School of Natural Sciences
Utrecht University

October 20, 2024

Makowski, Maciej:
Feature mappings for performative predictions Master Thesis, Utrecht University, 2023.

Abstract

Machine learning models have become widely used in recent years. As models are deployed and interact with real-world environments, they are susceptible to performance deterioration due to various factors such as data drift and model-induced changes in the surrounding environment. Addressing these challenges requires innovative approaches to maintain model effectiveness over time.

This research focuses on devising strategies to mitigate the effects of performative drift by exploring feature transformation techniques and robust classifier training methods. Drawing inspiration from transfer learning concepts, the study aims to find feature representations resilient to drift or capable of reversing its effects. Additionally, it investigates the feasibility of training drift-resistant classifiers in transformed feature spaces.

The research questions investigate the availability of performative data generators, methods for computing feature transformations, and the impact of these transformations on data distributions. Furthermore, the study examines the possibility of training robust classifiers independent of the strength of performative effects and explores potential modifications to improve the effectiveness of the proposed methods. The main innovation introduced in this paper is the design of an architecture capable of providing drift-resistant classification and mapping of points back to the starting distribution. The devised model is a synthesis of a domain adversarial neural network with a generative adversarial neural network.

The main experimental method used by this thesis is simulation, combining performative data generators available in the literature, existing transfer learning and newly created architecture. Finally, a series of experiments has been performed and it has been proven that under certain conditions it is possible to train a stable classifier. Alongside that classifier, a generator network is trained. That network with some approximation can reproduce the original form of the dataset, which has been influenced by performative drift.

Acknowledgements

I would like to express my gratitude towards my supervisor dr. ing. Georg Krempl for supervising this thesis. I am very grateful for many fruitful discussions and for pointing me towards interesting directions. I would also like to thank Brandon Gower-Winter for their meticulous input and suggestions for improvements in my work. Last but not least, I would like to thank my second supervisor prof. dr. Arno Siebes for his helpful attitude.

Contents

List of Figures x

List of Acronyms xi

1 Introduction 1
- 1.1 Problem statement . . . 1
- 1.2 Research questions . . . 3
- 1.3 Literature Research Protocol . . . 4

2 Background and Related Work 5
- 2.1 Learning in non-stationary environments . . . 5
  - 2.1.1 Types of drift and distribution shift . . . 5
  - 2.1.2 Performativity . . . 6
  - 2.1.3 Performative optimality and stability . . . 7
  - 2.1.4 Methods of minimisation of the performative risk . . . 7
  - 2.1.5 Performative data generators . . . 8
- 2.2 Transfer learning . . . 10
  - 2.2.1 Domain adaptation . . . 11
  - 2.2.2 Domain adaptation methods review . . . 12
  - 2.2.3 Asymmetric Regularized Cross-domain transformation . 14
  - 2.2.4 MMD-based transfer learning . . . 16
- 2.3 Domain Adversarial Neural Networks (DANN) . . . 18
- 2.4 Generative Adversarial Neural Networks . . . 20
  - 2.4.1 Conditional synthesis with generative adversarial nets . . 21
  - 2.4.2 Pix2pix image translation . . . 23

3 Method 25
- 3.1 Preliminary simulation design . . . 25
  - 3.1.1 Goal of the simulation . . . 25
  - 3.1.2 Simulation design . . . 26
  - 3.1.3 Preliminary results and motivation for the architecture design . . . 27
- 3.2 Architecture design . . . 29
  - 3.2.1 Objective . . . 30
  - 3.2.2 Training procedure . . . 32
  - 3.2.3 Network architectures . . . 35
    - 3.2.3.1 Feature extractor . . . 35
    - 3.2.3.2 Label Classifier . . . 35

3.2.3.3 Generator 36
3.2.3.4 Discriminator 37

4 Experiment and Results 39
4.1 Perdomo generator 40
4.1.1 Experiment setup 40
4.1.2 Perdomo learning curves 41
4.1.3 Perdomo results 42
4.1.3.1 Simulation with one classifier 42
4.1.3.2 Simulation with retraining 45
4.2 Izzo generator 47
4.2.1 Experiment setup 47
4.2.2 Izzo leaning curves 48
4.2.3 Izzo results 49

5 Summary 53
5.1 Conclusions 53
5.2 Research question answers 55
5.3 Limitations 55
5.4 Future Work 56

Bibliography 59

List of Figures

2.1 Exemplary Domain Adversarial Neural Network architecture source: [14] . . . 18
2.2 Achiterct　ure of a standard GAN model. . . . 20
2.3 Summary of differences between conditional and AC-GANs. . . 22
2.4 Illustration of pairing the target data with generated/real instances. Source:[21] . . . 23

3.1 Flowchart of the simulation process . . . 26
3.2 Proposed architecture for classification and mapping. . . . 29
3.3 The architecture of the feature extractor network. . . . 35
3.4 The architecture of the generator network. . . . 36
3.5 The architecture of the discriminator network. . . . 37

4.1 The loss of the discriminator over training. . . . 41
4.2 The domain accuracy over training . . . 41
4.3 The distance loss of the generator over training . . . 41
4.4 The adversarial loss of the generator over training . . . 41
4.5 Learning curves of the subparts of the entire architecture . . . 41
4.6 Results of the PCA analysis - generator Perdomo v1. . . . 43
4.7 Linegraph of accuracies - generator Perdomo v1. . . . 44
4.8 Correlation between accuracies and distances - generator Perdomo v1. . . . 44
4.9 Boxplot visualising differences between distributions - iteration 0 45
4.10 Boxplot visualising differences between distributions - iteration 9 45
4.11 Linegraph of accuracies - generator Perdomo v2. . . . 46
4.12 Correlation between accuracies and distances - generator Perdomo v2. . . . 46

4.13 The loss of the discriminator over training. 49
4.14 The domain accuracy over training 49
4.15 The distance loss of the generator over training 49
4.16 The adversarial loss of the generator over training 49
4.17 Learning curves of the subparts of the entire architecture 49
4.18 Results of Principal Component Analysis over the iterations. . . 50
4.19 Visualization of the cyclical nature of the drift. 51
4.20 Correlation of distance between distributions and accuracies of the models. 52

List of Acronyms

DANN Domain Adversarial Neural Network

EA Evolutionary algorithm

GA Genetic algorithm

GAN Generative Adversarial Network

KDE kernel density estimation

MMD Maximum Mean Discrepancy

TCA Transfer Component Analysis

TL Transfer learning

1. Introduction

1.1 Problem statement

The rapid development of machine learning techniques has revolutionised the process of decision-making across various domains. Plenty of research exists on how to utilise those technologies for various tasks. A typical data science pipeline could include data preparation, designing and training of a model, validation of the model and lastly deployment and monitoring. In those last stages, the decision-making algorithm is exposed to plenty of unforeseen factors that could deteriorate its performance.

A machine learning model is typically trained and validated on an offline dataset and only deployed once its performance matches the desired criteria. After that, the key aspect is monitoring of the chosen metrics to determine if the model is still useful. There might be multiple reasons, that stand behind the deterioration. One of them could be a change in the environment. The environment starts producing data, that originates in a data distribution significantly different than the one where the model was originally trained. The machine learning community has devised plenty of methods that can help deal with the process of deterioration. The most elementary approach is to gather the new data and retrain the model. This option might not be sustainable in the long term, as it would require constant monitoring to detect the drift efficiently. Also, it could cause overhead as all the arriving data would have to be stored and the retraining process could be costly. Another approach to dealing with that problem is online learning, which modifies the model from a sequence of data points, processing them one at a time [19]. Thanks to that the model can learn continuously. This approach has proven to be effective for many applications.

Another scenario might occur. Assume a situation in which the model that is being constantly modified induces drift on the environment itself. Such a situation could be called model drift or performativity (section Section 2.1.2 elaborates on that phenomenon). This can create a situation where the model's predictions, alter the model's input, by influencing the environment. A feedback loop is created. The dependence of the model on the environment is constantly increased and can finally

lead to the creation of a self-fulfilling prophecy. This happens when the model's predictions change the environment in such a way, that the environment begins to conform to those predictions. And the dependence between those two is constantly reinforced.

This research focuses on a scenario where the data is dependent on the predictions of any machine learning model. It attempts to find such a feature representation that will be drift-resistant or at least will allow for modifying the feature space in such a way that the effects of the drift are reversed. The idea for such an approach originated in the field of transfer learning (Section 2.2), where often knowledge inferred in one domain is transferred and utilized in another one. In this particular case, imagine that a model has been trained offline on one data distribution. Later its predictions influenced the new inputs to the model and the performance sunk. The idea is to find such a transformation between the original and the drifted data distributions that will be representative of the effects of the drift. Optimally after the transformation is applied to the data from a new distribution, the performance of the original model stays intact. The transformation allows for the reversal of the drift. Assuming such a mapping is impossible to find or the complexity of the operations is too big, another approach could be training a new classifier in the newly computed feature space. This could lead to the deployment of a classifier that is drift-resistant.

This thesis aims to devise a simulation that models the situation described above. To generate the data and replicate the dependence between the model and the environment, performative data generators available in the literature will be used. Next, various feature transformation techniques will be computed and a couple of metrics are going to be examined. Conclusions from those results are drawn and an architecture is devised. That newly devised model should allow for stable-classification and mapping of the points back to their original form. Before the drift has occurred. This architecture is evaluated in similar way the transfer learning techniques were evaluated with a couple of additional metrics.

1.2 Research questions

RQ1: What performative generators are available in the literature?

The field of *performative predictions* will be researched to find what methods of generation of synthetic data are present in the literature.

SRQ1.1: What type of drift is caused by each generator?

Understanding the relationship between the model and the environment has crucial meaning when it comes to designing the simulation. Additionally, different types of data generation processes can induce different types of drift on the data distributions, thus influencing the results of the simulation.

RQ2: What methods originating in the field of transfer learning can be used to compute a feature transformation, that will diminish the effects of the drift?

The field of *transfer learning* will be researched to find methods that prove to be effective in other applications and could be utilized to reverse the distribution shift.

SRQ2.1: To what degree the chosen methods are effective at providing a mapping?

The goal is to examine whether the selected methods can be used to compute such a mapping of the data, that will inform about the direction of the drift. Additionally, quantifying the results, the metrics used to measure the results will include the accuracy of the model trained in the original space when predicting the data after transformation.

RQ3: Is training a classifier that is robust towards model drift possible?

If the transformations do not manage to overcome the effects of the drift, the space they have created may be robust towards the drift and a classifier trained in that space could be drift-resistant.

SRQ3.1: If creating such a classifier is possible, is it independent of the strength of the performative effects?

The simulation will aim to quantify the robustness of the drift-resistant classifier.

RQ4: How to create a method that provides both drift-resistant classification and is also capable of providing mapping back to the starting distribution?

This question focuses on how to synthesise the transfer learning methods that have been reviewed before to achieve both goals.

SRQ4.1: How effective is the classification provided by the created method?

SRQ4.2: How accurate is the mapping provided by the created method?

1.3 Literature Research Protocol

Given the research questions stated in the section above, the literature research protocol needs to be defined. The chosen method for the process of performing the literature review is backwards and forwards snowballing. The entry point is the paper by Perdomo et al. [39] that establishes the framework for all the research dealing with performative effects. That paper allowed for performing a search for performative generators and standard strategies for dealing with the distribution shift induced by the model. This research aims to combine findings from various fields. Starting points for the snowballing process of each field were different surveys, which picture state-of-the-art methods in that field, or established textbooks. The literature review provides a steady foundation for the further research.

2. Background and Related Work

2.1 Learning in non-stationary environments

2.1.1 Types of drift and distribution shift

Most of the machine learning models are designed for real-world scenarios. The processes that occur in non-digital reality very often have random, non-linear characteristics. Change is an integral part of the human world. That can be disadvantageous to machine learning models, as they function well under the assumption, that the training and testing data originate from the same distribution and feature space. That exposes the models to plenty of deteriorating factors. One of them is model drift. In [10] types of drifts are categorised. However, that classification mainly focuses on intrinsic drift. This drift is caused by the changes in the environments surrounding the models and is ubiquitous, and appears in many situations. According to the author, the drift can be split into data, concept and dual drift [10].

In literature data drift and covariate shift names function interchangeably. The term is used to describe a difference in the distribution of an input variable between learning and a generalization phase [48]. To formalize the notation, if x is assumed to be the explanatory variable, $P_0(x)$ is the density of probabilities in the observed set and $P_1(x)$ would be the corresponding value for the test set. Then, if data drift has occurred, $P_0(x) \neq P_1(x)$.

Concept drift is also an effect of a changing environment. However, the implications it may have on data can be different. One of the common scenarios is that the relationship between the inputs and the targets changes. Which essentially deteriorates the performance of the model. Gama et al. provide a formal definition of concept drift [13].

$$\exists X : P_0(X, y, t_0) \neq P_1(X, y, t_1) \tag{2.1}$$

Where P_0 denotes the joint distribution between the input variable X and the target y at time t_0, and P_1 is the corresponding value at the time step t_1. The authors of

the very same paper also provide a couple of reasons for the concept drift occurrence. Mainly, a change in prior probabilities $p(y)$, class conditional probabilities $p(X|y)$ or a change in the posterior probabilities $p(y|X)$ [13].

2.1.2 Performativity

Especially important for this line of research is a special kind of drift, which originates in the model itself. The concept of *performativity* has been researched in the area of policy-making [18], but in the area of machine learning is still fairly novel. Main contributions to the field have been made by Perdomo et al. in their paper *"Performative prediction"* [39]. The authors have proposed a framework to describe the term in the context of machine learning, defined several terms such as *performative optimality, performative stability* and devised two methods to minimize the risk of the model influencing the predictions.

To illustrate the concept of performativity an example originating in the housing market will be considered. Assume the goal of a trained model is to predict property prices. The dataset, on which the model is trained includes numerous features. One of them is the safety of the location district of the property. The feature can take different values ranging from the most dangerous to the most safe. In that case, a logical conclusion is that the predicted by the model price for the assets, which are located in safer areas, would be higher. This could lead to a situation, where mostly people with lower incomes purchase those properties. Statistically speaking in districts with lower average income the probability of crime rates being high could be increased. This might affect the safety rating of the district and finally the model predictions in the future. This phenomenon could be treated as a self-fulfilling prophecy. A model's prediction influences the environment, creating a never-ending feedback loop.

Similar processes to the one described above, appear in the real world daily. In literature, the process is usually modelled as a *Stackelberg game* [45]. A game where one player (usually the leader) moves first and the second one replies. In this setup, the first player could be an institution deploying a classifier and the second one is the environment which attempts to adapt to achieve a more favourable verdict. This is the most standard way of modelling that situation. However, in their work, Zrnic et al. prove that the follower can reverse the roles, as the agents usually possess the ability to learn faster [54]. Nevertheless, actors responsible for deploying classifiers usually try to act upon drift detection. One of the most commonly undertaken actions is retraining the model on newly available data to adjust it accordingly to the environment changes.

Perdomo et al in their paper introduce the notion of performative risk [39]. Which is measured by a loss function l evaluating the influence of the current classifier's parameters on the future data distribution.

$$PR(\theta) = \mathop{\mathbb{E}}_{Z \sim D(\theta)} l(Z; \theta) \tag{2.2}$$

Where θ is a vector of model parameters, Z is the observed distribution $Z = (X, Y)$ of input variables over the targets. $D(\theta)$ is the map of distributions of Z that are caused by making decisions according to a model with parameter set θ.

2.1.3 Performative optimality and stability

The goal of supervised learning is to train such a classifier that will select the model parameters in such a way that the expected loss, measured for the prediction task, will be minimal. In the context of performativity, the task is very similar. The difference is the construction of the loss function, which takes into account mapping between the model parameters and data distributions (Eq. 2.2). A classifier is performatively optimal when it minimizes the expected performative risk.

$$\theta_{PO} = \underset{\theta}{argmin} \underset{Z \sim D(\theta)}{\mathbb{E}} l(Z;\theta) \tag{2.3}$$

Perdomo et al define the notion of *performative stability* as the fixed point of risk minimisation [39]. In other words, the data seen at the stable point does not provide a reason to deviate from the model. However, if a model is performatively stable it is still possible that there exists another model that would minimize the value of the loss function. Optimal classifiers do not necessarily need to be stable and the other way around. A formal definition, in its essence, is very similar to Eq. 2.3.

$$\theta_{PS} = \underset{\theta}{argmin} \underset{Z \sim D(\theta_{PS})}{\mathbb{E}} l(Z;\theta) \tag{2.4}$$

The difference is that instead of sampling from all models we the Eq. 2.4 will hold once the model parameters are sampled from a space of performatively stable classifiers. The definitions provided in this section aim to provide more clarity to the process of minimizing the performative risk, which is further described in the following sections.

2.1.4 Methods of minimisation of the performative risk

After introducing the concept of *perfromative risk* Perdomo et al. also describe the procedure that has to be performed to minimize it and select the most suitable classifier for the current environment state. It is called *Repeated Risk Minimization*, and it illustrates the procedure where at each iteration an optimal set of model parameters is selected. The result of the minimization is a time series of models, that provided the best loss function values at each iteration.

$$\theta_{t+1} = G(\theta_t) = \underset{\theta \in \Theta}{argmin} \underset{Z \sim D(\theta_t)}{\mathbb{E}} l(Z;\theta) \tag{2.5}$$

However, to enable the process a couple of very strict assumptions need to be met. Assumptions:

- The distribution D needs to be Lipschitz continuous. If the decisions are made based on previously deployed models, the outcome distributions over the instances should also resemble similarity [39].
- The loss function has to be jointly smooth
- The loss function has to be strongly convex

Once those assumptions are met, Perdomo et al. prove that the classifier only needs to be retrained a small number of times and after that, it will converge to a *performatively stable point*.

Retraining could be considered the most straightforward strategy for discovering a stable classifier. Researchers have described many possible approaches for performing that search. Follow-up studies have covered methods such as Repeated Gradient Descent (RGD) [39], Stochastic Optimization (SO)[27], Derivative-Free Optimization [22, 29], Bandit Optimization [23]. RGD and SO methods perform updates of the model, based on calculating the derivative of the performative risk function, this implies that the function has to be differentiable, which enforces additional assumptions. Derivative-free optimization and Bandit optimization also require imposing additional assumptions due to their mathematical structure.

2.1.5 Performative data generators

Gathering data, that illustrates performativity well, can be a gruelling task. It would require an institution to comprehensively record not only the data but also, which model was deployed at which time, how the model changed and how it influenced the surrounding environment. Due to those factors, most of the algorithms described in Section 2.1.4 prove their theorems on synthetic data. One of the most popular data generators that has been widely used is based on a dataset *Give Me Some Credit*, publicly available on *Kaggle* [8]. The generator has been devised by Perdomo et al. and can be implemented using a Python package *WhyNot* [28]. The articles, which did use that specific data simulator, include [6, 22, 27].

The *Give Me Some Credit* dataset has been adopted for the needs of the simulator. Mainly, 18357 points have been sampled so that the class imbalance problem is solved (the data represents binary classes, 45% of instances are assigned class 1, the rest 0). Each instance possesses 11 features. The data has been scaled so that it has zero mean and unit variance. The simulator simulates the influence of a model on the data, by modifying three features, which are considered to be strategic (utilization of credit lines, number of open credit lines, and number of estate loans). The strategic response can be described in the following way.

$$x_{t+1} = x - \epsilon B\theta \tag{2.6}$$

Where B is a matrix that informs about features which are considered strategic, θ is a vector of model parameters (in this case logistic regression), ϵ is a scalar value and describes the performative power, so the degree to which the model influences the environment. It is crucial to note, that in the case of this generator, the labels originate from the GiveMeSomeCredit dataset and do not change over time.

The simulator described above is one of the only publicly available data generators based on real-world data. Most of the other approaches focus on generating synthetic data, that would resemble performativity. For instance, Miller et al propose a process for generating data for linear regression [29]. The aim is to create feature label pairs (x,y). Firstly, draw features x from a Gaussian distribution with a zero mean and covariance matrix Σ_x, mathematical notation - ($x \sim \mathcal{N}(0, \Sigma_x)$). Where the covariance matrix is subject to being a symmetric positive definite matrix with

operator norm equal to 0.01. Generate the labels by sampling from a conditional probability distribution $y|x \sim g(x)+\mu^T\theta+U_y$, where $g(x) = \beta^T x$ and beta is sampled from a Gaussian distribution with zero mean and unit variance. U_y is white noise generated identically. θ again depicts the coefficients of the predicting model and $\epsilon = ||\mu||_2$. Meaning that the Frobenius norm of μ is equal to the coefficient describing how powerful the influence of the model on the environment is.

The third approach to the data-generating process is described by the authors of [22, 27]. The approach differs from the two, that have been described above as the researchers present themselves with the challenge of estimating the mean of Gaussian random variable, which depends on the classifier.

$$x \sim \mathcal{D}(\theta) = \mathcal{N}(\mu + \epsilon\theta, \sigma^2) \tag{2.7}$$

Similar to the examples above, θ symbolizes the parameters of the model and ϵ the strength of performative effects. In their research, the authors do not attempt to overcomplicate the data generation process, as their focus is on measuring the pace of convergence to a performatively stable point and not analyzing the correlations between the features. Researchers analyse only a single variable but the process could be extended to generate a whole dataset. Furthermore, it is proven, that for this sampling process, there exists a unique stable point as long as the performative strength is not too powerful, strictly when $\epsilon < 1$.

Another approach has been described by Izzo et al and attempts to model performativity in the spam detection domain [22]. The reasoning behind the data generation process is that spammers will do their best to deceive the classifier, i.e. they will attempt to modify their features. Meanwhile, regular users will stick to their regular behaviour and do not modify their features to a significant degree. Based on this observation the following idea for data generation is put forward.

$$x|y = 0, \theta \sim \mathcal{N}(\mu_0, \sigma_0^2) \qquad x|y = 1, \theta \sim \mathcal{N}(f(\theta), \sigma_1^2) \tag{2.8}$$

What can be inferred from the equations above, is that the performative effect will only apply to instances which are classified as spam ($y = 1$). The effect is introduced by moving the average of the normal distribution the data is sampled from. The shift is induced on the distribution by $f(\theta)$, the researchers define this function in a similar way to Perdomo et al. [39]. Specifically, $f(\theta) = \mu_1 - \epsilon \cdot \theta$, the resemblance with Eq. 2.6 is significant, however in the case of this data generator, all features are modified.

2.2 Transfer learning

Transfer learning similar to many other concepts in modern machine learning originates in psychology. An example, which illustrates the idea well, could be a person who is exceptional at snowboarding and wants to learn kitesurfing or wakeboarding. It would probably be much easier for this person than for a person, who is inexperienced in any of those areas. Many muscle memory movements and balance could be adopted from one sport to another. Due to the similarity of the domains, the old experiences could benefit the process of acquiring knowledge to a great extent. This is just an example of a real-world situation, which is only applicable to humans. In a machine learning scenario, the outline of the problem could be defined differently.

Pan et al. [37] provide a formal definition of the term.

Definition 2.1 (Transfer learning (**TL**)). *Given a source domain $\mathcal{D}_s$ and a learning task $\mathcal{T}_s$, a target domain $\mathcal{D}_t$ and learning task $\mathcal{T}_t$, transfer learning aims to help improve the learning of the target predictive function $f_t(\cdot)$ in $\mathcal{D}_t$ using the knowledge in $\mathcal{D}_s$ and $\mathcal{T}_s$, where $\mathcal{D}_t \neq \mathcal{D}_s$ or $\mathcal{T}_t \neq \mathcal{T}_s$.*

Transfer learning has been widely adopted since multiple large databases for training models have become available, such as ImageNet [41], Caltech-256 [16], Places [50]. Transfer learning can provide a solution to the problem of learning a predictive function for a dataset with limited data or limited labels. Instead of gathering more data or hiring an expert for labelling, a model trained on one of the huge datasets can be fine-tuned to help with solving the task in the smaller, limited target domain. The described situation is one particular scenario of transfer learning. Zhang et al., take the labelling perspective into account, and categorize transfer learning into [53]:

- **transductive TL** - the label is only available in the source domain
- **inductive TL** - the labels are available in both source and target domains
- **unsupervised TL** - the labels are not available for any of the domains

The latter categorization only aims to give a general perspective into the division, more precise and accurate definitions have been provided by [37]. Another fundamental division of TL could be made, that division takes into account the way of transformation of source and target domains. The categories, which could be distinguished from it are presented below:

- **Instance Weighting Approach** - assigns weights to instances to adapt the marginal distributions in source and target domains
- **Feature Transformation Approach** - creates a new feature representation
- **Parameter Based Approach** - focuses on transferring the knowledge through model parameters
- **Relational Based Approach** - tries to transfer the logical rules from one domain to another

For the line of research, described in this paper, the most crucial part of transfer learning will be the inductive feature transformation approach.

2.2.1 Domain adaptation

According to Pan et al. domain adaptation is a special case of transfer learning [37]. Specifically when the data in the target and source domains is different, however, the learning tasks are identical. If two domains are related to each other, there is a large probability that there exist factors underlying the correlation. Some of those factors may cause the differences between data distributions, while others might capture the intrinsic nature of the data and can be further analyzed to understand the influence [36]. Traditional domain adaptation methods attempt to minimize the shift between the distributions. Mottiian et al. categorize those attempts into finding a mapping between the distribution **(1)**, finding a shared latent space between distributions **(2)** and regularizing the classifier trained on the source distribution to work well on the target data **(3)**[31].

Usually, minimizing the distance between the distributions is the objective, either in regular or higher-dimensional space. However, it is crucial to answer the question of which data distributions should be taken into account. There exist several approaches, to name them:

- MDA - Marginal Distribution Adaptation - needs to be performed when $P_s(x) \neq P_t(x)$
- CDA - Conditional Distribution Adaptation - assumptions are opposite to the ones of MDA $P_s(x) \approx P_t(x)$, but $P_s(y|x) \neq P_t(y|x)$
- JDA - Joint Distribution Adaptation - utilizes both MDA and JDA to estimate the joint probability distribution $D(P_s(x,y), P_t(x,y)) \approx D(P_s(x), P_t(x)) + D(P_s(y|x), P_t(y|x))$

Zhao et al. have proved that considering not only marginal but also conditional distribution might provide better results [49]. Plenty of literature sources describe the methods of domain adaptation listed above [26, 47, 52]. However, a more flexible approach is called Dynamic Distribution Adaptation (DDA) Eq. 2.9. This approach can fluently adapt the importance ratio between marginal and conditional distributions [46]. This effect is achieved by the introduction of a parameter $\mu \in [0, 1]$. If the parameter is close to 0, it assigns more importance to the marginal distribution, if its value is in proximity of 1, the conditional distribution contributes more to the distance metric.

$$D(\mathcal{D}_s, \mathcal{D}_t) \approx (1-\mu)D(P_s(x_s), P_t(x)) + \mu D(P_s(y|x), P_t(y|x)) \tag{2.9}$$

This way of calculating the distance between distributions provides a solution to a problem when marginal and conditional distributions do not contribute in the same way to the overall distance metric [46].

2.2.2 Domain adaptation methods review

As mentioned above there exist multiple approaches to domain adaptation, a short description and classification are provided in Table 2.1 and Table 2.2. The review is not extensive, but it aims at representing different approaches. An important factor that has to be considered is that the differences between the methods might be very problem-specific and the preselection of one optimal method might be a challenging task. Two of the methods will also be described in more detail in the following paragraphs. Here a reasoning for the selection of those two methods is given. The methods that will be described further are Asymmetric Regularized Cross-domain transformation [42] and Transfer Component Analysis [36] or rather to be more specific Maximum Mean Discrepancy (TCA is part of the MMD-methods). To specify, in this section we refer only to TCA, however all MMD based methods are similar and only differ, when it comes to the type of probability that is measured. First of all, to be demonstrative one representative from each category of methods had to be picked. Considering the techniques, which aim to learn feature mapping, ARC-t is the only one that takes non-linearity into account. It also is the extension and generalization of Regularized Cross-Domain Transform. That is why, it might be widely applicable and suit more cases than the other methods.

Method	**Type**	**Technique**	**Description**	**Optimization Function**
Regularized Cross-Domain Transform	1	Linear transformation	Goal is to learn a linear transformation W between X and Y originating from source and target domains. Form a similarity function and utilize it in a classification or clustering. [42]	$min_W \quad r(W) + \lambda \sum_i c_i(X^T W Y)$ where c_i - constraints functions
Asymmetric Regularized Cross-domain Transform ARC-t	1	Non-linear kernel transformation	Extension of [42]. Kulis et al. prove that the optimization function from a cell above can be solved in kernel space for a wide class of regularizers. The optimization is reformulated using inner products, which are later substituted for kernel functions [25].	$min_L \quad r(L) + \lambda \sum_i c_i(K_X^{\frac{1}{2}} L K_Y^{\frac{1}{2}})$ where K are kernels and $W = X K_X^{\frac{-1}{2}} L K_Y^{\frac{-1}{2}} Y^T$
Unsupervised subspace alignment	1	Subspace linear transformation	Firstly for both source X_S and target X_T domains subspaces are generated by using PCA to generate eigenvectors, which act as bases for subspaces. Then data from original distributions is projected into subspaces. Finally, a linear transformation M is learnt that aligns the source subspace to the target one. [12]	M is learnt by minimizing the Bregman matrix divergence presented below $F(M) = \|\|X_S M - X_T\|\|_F^2$ the optimal M solution is $M^* = argmin_M\ (F(M))$

Table 2.1: Domain Adaptation: Feature mappings methods

If the second category (methods that find shared latent space between distributions and optimize in it) is taken into account, TCA and DICA could be considered similar. It is even stated by the authors of the paper, that methods are closely related [32].

Method	Type	Technique	Description	Optimization Function
Domain Invariant Component Analysis - DICA	2	Orthogonal transform onto a low dimensional subspace that minimizes the distributional variance	Finding a transformation $\mathcal{B}$ in $\mathcal{H}$ that minimizes the distance between empirical distributions of the original samples transformed into $\mathcal{H}$. Where $\mathcal{H}$ is a reproducing kernel Hilbert space with a mapping $\mathcal{F}$ which kernelizes $\mathcal{X} \rightarrow \phi(\mathcal{X}) \in \mathcal{H}$. The transformation does not only minimize the variance but also preserves the functional relationship, i.e. $X \perp\!\!\!\perp Y \mid \mathcal{B}$ [32].	The optimization function forces the complexity $\mathcal{B}$ to be small and allows for finding a solution, that satisfies the objectives described in the cell on the left, however the elaborateness of the optimization function exceeds the scope of this table.
Transfer Component Analysis - TCA	2	Set of transfer components that project features onto latent space and later minimise the distance between domains	Goal of this method is to find a non-linear mapping ϕ that embeds both target and source domains into a low-level space. Then a matrix W is found that minimises MMD (Maximum Mean Discrepancy) in the subspaces. A constraint that aims to preserve the properties of X_s and X_t is added to preserve the variance in the latent space [36].	$min_W tr(W^T KLKW) + tr(W^T W)$ where K is the kernel matrix, L is a matrix of coefficients and W is the transformation from one subspace to another. The TCA method is further described Section 2.2.4.
Model Transfering for DA	3	Model Adaptation	Learning from the source model w^s by regularizing the distance between the learned model w and w_s [3]. Aytar et al. provide a couple of modifications of learning objectives functions for support vector machines, that can optimize the classifier in the new domain.	The functions used for twisting the models are a way of domain adaptation, however elaboration about them is not necessary for this line of research.

Table 2.2: Domain Adaptation: Kernelized low-dimensional space and model modification methods

Nevertheless, DICA focuses more on the separation of classes within each domain. Meanwhile, TCA utilizes the Maximum Mean Discrepancy (MMD) to minimize the distance between the shared low dimensional latent spaces, which could be a provided angle. TCA is also adapted to a wider range of tasks.

No elaboration is provided on methods that do not try to perform domain adaptation through feature modification but focus on the model parameters instead. This thesisaims to provide more insight into how feature mappings could be utilized in a scenario where *performative predictions* might occur.

2.2.3 Asymmetric Regularized Cross-domain transformation

Kulis et al. have described a method of learning an asymmetric non-linear transformation, which maps points from one domain to another using labelled data [25]. Researchers were trying to extend and generalize the method of Saenko et al. [42]. Notably, they aimed to broaden and refine the technique by relaxing the constraints imposed by the regularizer described in the preceding work. They achieved the generalization by solving a linear optimization problem Eq. 2.10 in a kernel space.

$$\min_{W} r(W) + \lambda \sum_{i} c_i(X^T W Y) \tag{2.10}$$

Where W is a linear transformation matrix, X, Y represent the points in source and target domains, c_i represents loss functions over the constraints and r is a term, which regularizes matrix W. The authors of the paper describe one particular example, where the regularization term is a squared Frobenius norm and constraint represent similarities between X and Y. To be exact, in the process of constraint generation such a similarity function is used, so that the differences between same-category pairs and different-category pairs in the source and target domains are highlighted. The mathematical formula for constraint generation for the (x,y) pairs from the same category.

$$c_i(X^T W Y) = (max(0, l - x^T W y))^2$$

And for pairs from different categories.

$$c_i(X^T W Y) = (max(0, x^T W y - u))^2$$

Kulis et al. provide proof, that computing the transformation is possible using only kernel functions. They project the problem from Eq. 2.10 to a space constructed by kernelization and show that solutions are equivalent. That holds but only if a few assumptions hold, mainly kernels are strictly positive definite and regularizer r is convex.

$$\begin{gathered} \min_{L} r(L) + \lambda \sum_{i} c_i(K_X^{\frac{1}{2}} L K_Y^{\frac{1}{2}}) \\ then \\ W^* = X K_X^{-\frac{1}{2}} L^* K_Y^{-\frac{1}{2}} Y^T \end{gathered} \tag{2.11}$$

Eq. 2.11 provides the optimization function in the transformed subspace. K_X and K_Y are the kernel matrices, applied to the points in source and target domains. The kernel used for transformation from regular feature space to kernel space can be arbitrary, in the paper the authors use an RBF Gaussian kernel. W^* is an optimal solution to Eq. 2.10 and L^* is an optimal solution for the equation Eq. 2.11. It is important to note that the constraints generation also needs to be mapped to kernel

space, for instance, the similarity function for instances from the same category presented above would get mapped to $(max(0, l - e_i^T K_A^{1/2} L K_B^{1/2} e_j))^2$.

The introduction of this method of computing the transformation matrix allows for non-linear solutions and assures the matrix is independent of the dimensions of the domain. To solve the optimization problem authors of the paper suggest using Bregman's algorithm. At the same time, they also proclaim that solving it with simple or stochastic gradient descent approaches is possible.

2.2.4 MMD-based transfer learning

As mentioned in Section 2.2.1 the usual goal in Domain Adaptation is to minimize the distance between 2 probability distributions. A metric, that is often used to express this distance is Maximum Mean Discrepancy. The MMD takes advantage of a kernel trick, it was originally used as a statistical test, that informed whether two samples originate in the same distribution [5]. Currently, MMD methods are widely present in transfer learning, as well as in the procedure of training adversarial neural networks.

MMD compares the mean embeddings of two distributions in a kernel space. Let $\mathcal{H}_k$ denote a reproducing kernel Hilbert space, defined by a characteristic kernel k. $d_k(p, q)$ will denote the MMD value between two distributions p and q [46].

$$d_k^2(p, q) = ||\mathbb{E}_{x \sim p}[\phi(x)] - \mathbb{E}_{x \sim q}[\phi(x)]||^2_{\mathcal{H}_k} \tag{2.12}$$

Where ϕ is the transformation function that maps data from original spaces into a space defined by kernel k. In that case, the kernel function could be defined in the following way. [46]

$$k(x_i, x_j) = < \phi(x_i), \phi(x_j) > \tag{2.13}$$

The most popular kernel functions include linear, polynomial and Gaussian kernel. An approach to compare two distributions in a kernel space has been described. Recalling the Eq. 2.9, the optimization problem can be created. A key idea is to introduce a transformation matrix A, that denotes the feature transformation in the kernel space, the matrix A becomes the learning objective. The next step is to compute the distances between source and target transformed data. This can be achieved in the following way for the marginal distributions.

$$MMD(P_s(x), P_t(x)) = ||\frac{1}{N_s}\sum_{i=1}^{N_s} A^T x_i - \frac{1}{N_t}\sum_{j=1}^{N_t} A^T x_j||^2_{\mathcal{H}} \tag{2.14}$$

And in the following way for the conditional distributions. The derivation of the marginal one is a bit more intuitive. To achieve the result below Bayes theorem is used and the fact that $P(y|x) = P(y)P(x|y)$ and also the conditional probabilities take into the division of classes that is why instances representing the same category need to be summed.

$$MMD(P_s(y|x), P_t(y|x)) = \sum_{c=1}^{C} ||\frac{1}{N_s^{(c)}}\sum_{x_i \in \mathcal{D}_s^{(c)}} A^T x_i + \frac{1}{N_t^{(c)}}\sum_{x_j \in \mathcal{D}_t^{(c)}} A^T x_j||^2_{\mathcal{H}} \tag{2.15}$$

If the two above equations are added together a general optimization formula is achieved. It seems that it might be difficult to get a solution to that problem. However, Wang et al. provide proof that it also can be reformulated and simplified [46]. They show that the solution to the following equation provides the same result as the solution of the optimization problem.

$$min\ tr(A^T X M X^T A) \tag{2.16}$$

Where tr(.) denotes the trace of the outcome matrix, X is a matrix that combines data points from both source and target domains and M is a matrix of MMD coefficients (representing the constraints that stand before the summation marks in Eq. 2.14 and Eq. 2.15), computed in the following manner.

$$M = (1-\mu)M_0 + \mu \sum_{c=1}^{C} M_c \tag{2.17}$$

The resemblance of the Eq. 2.9 and Eq. 2.17 is not coincidental, as the meaning of the parameter μ in this case is identical as the meaning described in Section 2.2.1. Additionally, M_0 is filled with coefficients $\frac{1}{N_S^2}$, $\frac{1}{N_T^2}$ and $\frac{-1}{N_S N_T}$, placced in a way corresponding with the result of $A^T X$. M_C contains identical components but distinguishes the classes. Adjustments of the value of μ allow for manipulation in the methods of distribution adaptation. For instance, if μ is set to 0 then the method is regular TCA - Transfer Component Analysis [25], changes in the value of this parameter will rebalance the ratio of importance between marginal and conditional probability distributions.

The Eq. 2.16 states the problem that needs to be solved to achieve the proximity of distributions. The optimization is still subject to constraints, mainly to maximize the data variance in the RKHS, Wang et al introduce a centring matrix **H**, where $\mathbf{H} = \mathbf{I} - (1/n)\mathbf{1}$ [46]. The H matrix is used to compute the scatter matrix, which enables estimating the covariances between features. The scatter matrix in the original feature space would be computed accordingly, $S = XHX^T$. Then the problem of maximizing the variance in the RKHS becomes.

$$max(A^T X)H(A^T X)^T \tag{2.18}$$

By combining equations Eq. 2.16 and Eq. 2.18 and applying the Rayleigh quotient to the result of this combination, it is possible to formulate the final constrained optimization problem. That can later be solved with the Lagrangian method for optimization [1].

$$\begin{aligned} min \quad & tr(A^T X M X^T A) + \lambda ||A||_F^2, \\ s.t. \quad & A^T X H X^T A = I \end{aligned} \tag{2.19}$$

Solving that problem is not complicated, as when the gradient of the Lagrangian is set to 0, the solution becomes a system of equations. Later the best solutions to the system of equations need to be selected and that allows us to construct the transformation matrix.

2.3 Domain Adversarial Neural Networks (DANN)

Past years have seen rapid development of deep learning techniques, that are currently applied to a wide range of tasks. Domain adaptation problems have also been under scientists' radar. Domain Adversarial Neural Networks are a solution that applies deep learning to deal with the issue of distribution shift between the domains. Ganin et al. in their paper *Domain-Adversarial Training of Neural Networks* have focused on designing a training process that would embed domain adaptation into learning the feature representation. Essentially, the disparity between two distributions is measured based on a deep discriminatively trained classifier [14]. The method is distinct from the ones described in the previous sections as instead of directly comparing the distance between distributions, it aims to train a feature extractor, label predictor and domain classifier that together will be capable of distinguishing between the domains.

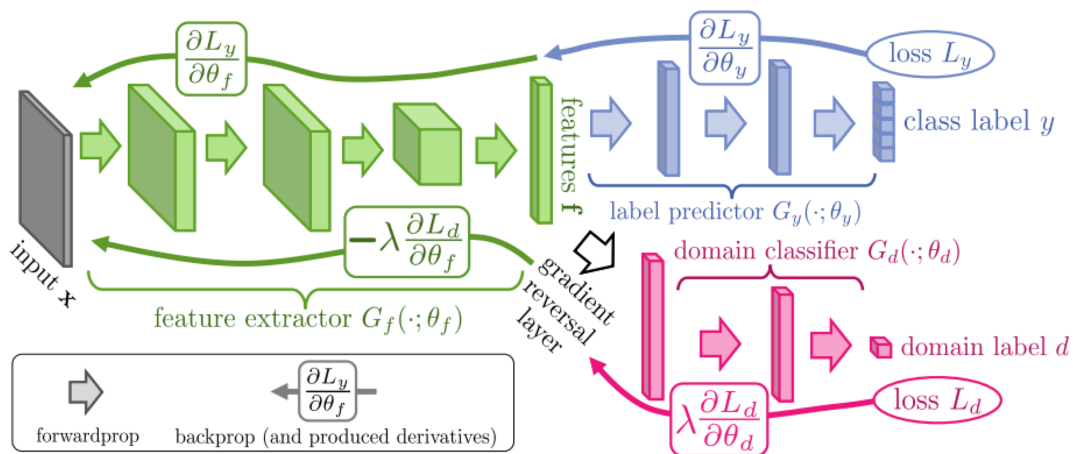

Figure 2.1: Exemplary Domain Adversarial Neural Network architecture source: [14]
.

Figure 2.1 illustrates a possible architecture of a DANN. To illustrate the mechanisms behind the architecture the following notation will be introduced. Let $G_f(\cdot, \theta_f)$ denote a neural network which is responsible for extracting the features from data (on the Figure 2.1 the green part, however, the architecture does not necessarily need to be convolutional). $G_y(\cdot, \theta_y)$ is the part of the architecture, that is responsible for predicting the output class label (blue part). Together, green and blue networks would constitute a fully connected neural net. The differentiating factor is the implementation of the classifier computing the domain prediction output, depicted in pink, the formal term describing it is $G_d(\cdot, \theta_d)$. Now let each point in a dataset consist of a tuple (x_i, y_i, d_i), a set of features, the label and the domain label. Then the prediction loss and the domain loss could be denoted as follows.

$$\mathcal{L}_y^i(\theta_f, \theta_y) = \mathcal{L}_y(G_y(G_f(x_i; \theta_f), \theta_y), y_i) \tag{2.20}$$

To explain the notation, first each instance x_i goes through the layers of the features extractor and a new feature representation is created. Then the output of $G_f(x_i;\theta_f)$ passes through the fully connected layer G_y characterized by a set of parameters θ_y and finally the predicted label is compared with the actual one y_i and the loss function is computed.

$$\mathcal{L}_d^i(\theta_f,\theta_d) = \mathcal{L}_d(G_d(G_f(x_i;\theta_f);\theta_d),d_i) \tag{2.21}$$

The loss function for the domain classifier is constructed correspondingly. After passing through the feature extractor every instance's originating domain is predicted and depending on that prediction a loss function is computed. The training process of a DANN combines Eq. 2.20 and Eq. 2.21. In the end, it is equivalent to solving the following optimization problem.

$$E(\theta_f,\theta_y,\theta_d) = \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}_y^i(\theta_f,\theta_y) - \lambda(\frac{1}{n}\sum_{i=1}^{n}\mathcal{L}_d^i(\theta_f,\theta_d) + \frac{1}{n'}\sum_{i=n+1}^{N}\mathcal{L}_d^i(\theta_f,\theta_d)) \tag{2.22}$$

In the above Eq. 2.22 it is important to note that n denotes the number of instances originating from the source domain, n' denotes the instances originating from the target domain and $N = n + n'$ is the sum of both. The manner of calculating the updates of the weights of each part of the architecture is also presented on Figure 2.1. The process resembles a classical Stochastic Gradient Descent. However, there is one important difference in the gradients from the class and domain predictor nets are substracts. Due to that fact, the entire model can extract features, which are generic, not discriminative when it comes to domain. The update rule for the feature extractor can be described with the following equation.

$$\theta_f \leftarrow \theta_f - \mu(\frac{\partial \mathcal{L}_y^i}{\partial \theta_f} - \lambda\frac{\partial \mathcal{L}_d^i}{\partial \theta_f}) \tag{2.23}$$

Due to technicalities, such as the implementation of Stochastic Gradient Descent in machine learning libraries. Commonly, the update rule is replaced by the introduction of a gradient reversal layer. This reversal layer does not have any parameters. During forward propagation, it acts as an identity matrix and in the backpropagation stage the sign is reversed. The outcome is multiplication by -1. This way, the update rule presented in Eq. 2.23 can be achieved. For a more mathematical description of the reversal layer, please refer to [14].

2.4 Generative Adversarial Neural Networks

This section dives into a topic, that at first glance is significantly different from the ones described above. However, as over the previous years, generative AI tools have gained significant recognition, it would be foolish to overlook their usefulness. The Generative Adversarial Network (*GAN*)s prove to be proficient at modelling high-dimensional data distributions [9], which might be immensely beneficial in mapping the points back to their original format and might help in finding a suitable transformation.

The concept of GANs has been introduced by Goodfellow et al. in *"Generative Adversarial Nets"*, the authors put forward a *adversarial* framework and model it as a two-player game [15]. The first player is the generator, whose task is to produce an output that will resemble the desired features. Meaning that the output of the generator will be as close to the target distribution as possible. The second player is the discriminator, whose task is to tell apart real and fake samples and learn how to distinguish between them. Both models are trained simultaneously and compete with each other. During training, only the discriminator has access to instances sampled from real distribution, which are later paired with synthetic instances produced by the generator. In a classical setup, the input to the generator is white noise. The Figure 2.2 illustrates the most standard scenario. The "OR" gate on the chart models the fact that the discriminator's inputs are equally split between the generated samples X' and points from the original distribution X.

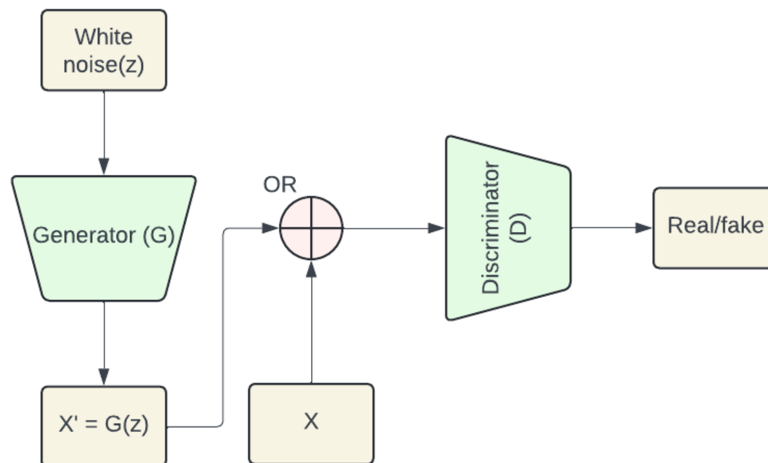

Figure 2.2: Achitercture of a standard GAN model.

To describe the training process in more detail and formalize the notation the following notation, describing the loss functions, is introduced. Discriminator aims to maximize the probability of assigning the correct label to X and X'. Meanwhile, the goal of the generator is to minimize $log(1 - D(G(Z))$. So the goal of G is to trick D into predicting that the true label of X' is 1 (the instances sampled from X

are denoted with domain source labels 1, the generated ones with zeros). The total loss function of a GAN is often described as [15]

$$\min_G \max_D F(D, G) = \mathbb{E}_{X \sim P(X)} \left[log(D(X)) \right] + \mathbb{E}_{z \sim P(z)} \left[log(1 - D(G(Z)) \right] \quad (2.24)$$

Due to some issues that can occur at the beginning of the training process the updates of the weights of the components of the GAN model are usually performed in a slightly different manner, which is still equivalent to Eq. 2.24. To be exact, the updates are performed in the following way for the discriminator.

$$\theta_D \leftarrow \theta_D + \nabla \frac{1}{n} \sum_{i=1}^{n} \left[log(D(x^{(i)})) + log(1 - D(G(z^{(i)}))) \right] \quad (2.25)$$

And for the generator

$$\theta_G \leftarrow \theta_G - \nabla \frac{1}{n} \sum_{i=1}^{n} \left[log(1 - D(G(z^{(i)}))) \right] \quad (2.26)$$

In the present literature, one can find plenty of examples of the usage of GANs, as well as an abundance of different types of networks. Authors of the paper [17] attempt to classify the architectures based on their structures and applications. They classify GANs into 3 main representative variants.

1. **InfoGAN** [7]- standard GAN, where the object function is extended with mutual information, between the original class and the generated class.

2. **Conditional GANs - cGANs** [30]- that can generate examples conditioned on class labels. Both the discriminator and generator are conditioned on extra information y.

3. **CycleGAN** [51]- mostly used for image-to-image translation, in situations where data is unpaired. Their architecture utilises additional generators and discriminators to mitigate that issue.

Further subsections will elaborate on types of GANs specifically important for this line of research.

2.4.1 Conditional synthesis with generative adversarial nets

The typical GAN architecture described in the section is limited to the amount of information it can carry. Meaning that the only information that is provided to the model is the labelling of fake and real instances and certainly the instances themselves. Mirza et al. propose to condition the model on additional data (such as class labels or other data modalities)[30]. The result of that is that the objective function that has been described by Eq. 2.24 changes to the following equation.

$$\min_G \max_D F(D, G) = \mathbb{E}_{X \sim P(x)} \left[log(D(\mathbf{x}|\mathbf{y})) \right] + \mathbb{E}_{z \sim P(z)} \left[log(1 - D(G(\mathbf{z}|\mathbf{y})) \right] \quad (2.27)$$

What that means for the architecture, is that both generator and discriminator are given additional inputs y. The generator is provided with white noise and an additional embedding that is concatenated with the white noise. That embedding provides information to which class should the produced instance belong. The discriminator is also provided with the very same information so that it can predict whether the instance is real or fake given **y**, however in this case the information is just provided as an input to the model. For the conditional network, the only difference is that labels are embedded into arbitrary form and fed to the models, which due to that start conditioning their predictions on that additional information.

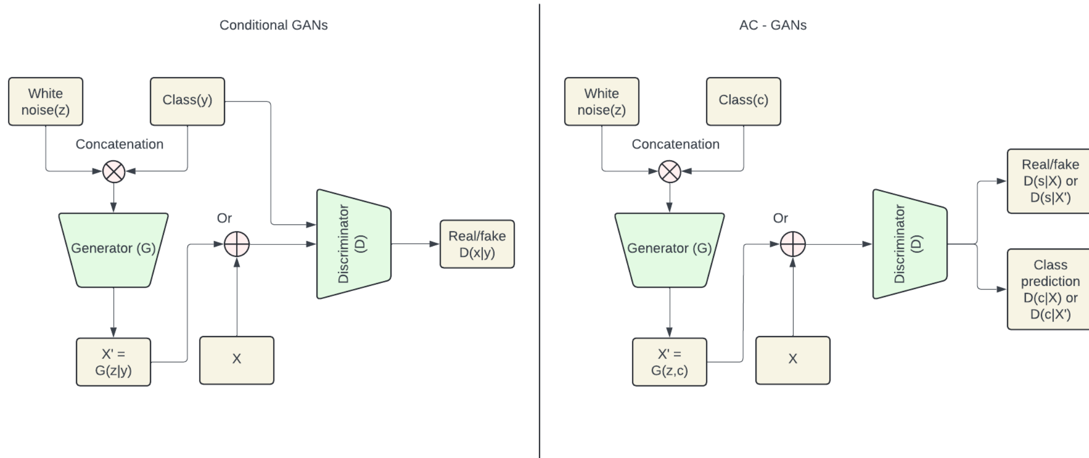

Figure 2.3: Summary of differences between conditional and AC-GANs.

Odena et al. in their paper *"Conditional Image Synthesis with Auxiliary Classifier GANs"* propose an extension to the conditional framework, by introducing an auxiliary classifier [34]. They call their architecture AC-GANs, which stands for auxiliary classifier GAN. The generator job does not change much, however, the discriminator now is presented with the task of outputting two probability distributions. One describes the probability of the instance being real or fake and the second one that predicts the chances of an instance belonging to a certain class. The equation Eq. 2.28 shows the dual nature of the loss of the discriminator in this architecture.

$$L_{source} = \mathbb{E}_{X \sim P(x)} \left[log(P(S = real|x)) \right] + \mathbb{E}_{z \sim P(z)} \left[log(P(S = fake|G(z, c)) \right] \tag{2.28}$$

$$L_{class} = \mathbb{E}_{X \sim P(x)} \left[log(P(C = c|x)) \right] + \mathbb{E}_{z \sim P(z)} \left[log(P(C = c|G(z, c)) \right]$$

What is crucial during the training procedure of training an AC-GAN, is the fact that the discriminator attempts to maximize $L_D = L_{source} + L_{class}$ and the generator aims to maximize $L_G = L_{class} - L_{source}$. This means, that both of the models work together to optimize predicting classes, but the adversarial part, where one of them tries to fool the other one is still present.

2.4.2 Pix2pix image translation

Generative adversarial networks have a wide scope of applications. It is especially common to see them applied in the area of image construction. Another interesting application is image translation. For instance, with small architecture adjustments, it is possible to adopt a GAN, so that it can be used to recognize edges or to translate online maps into satellite images. Those examples seem to be very loosely connected with finding transformation for *performative predictions*. However, with justified alterations, those solutions could be implemented in theoretically simple cases such as datasets without spatial structure.

Isola et al. devise a method, that can be used for image-to-image translations [21]. The approach is based on the idea of conditional GANs, which have been described in the subsection above. The notation introduced in this paragraph slightly differs from those in the preceding parts, so for clarity reasons, the following occurs.

- **Standard GAN** - learns a mapping from noise z to an output $y, G : z \to y$ [15]

- **Conditional GAN** - learns a mapping from an observed modality x and random noise z to an output y, $G : \{x, z\} \to y$ [21]

The idea that the researchers have put forward is to change the inputs to the discriminator. Extend those inputs by pairing the real observed data x with the target images y or the output of the generator $G(x)$. This way the generator should learn how to produce target images from pieces of information included in x. The Figure 2.4 visualizes the idea of matching of the data points. As with every new

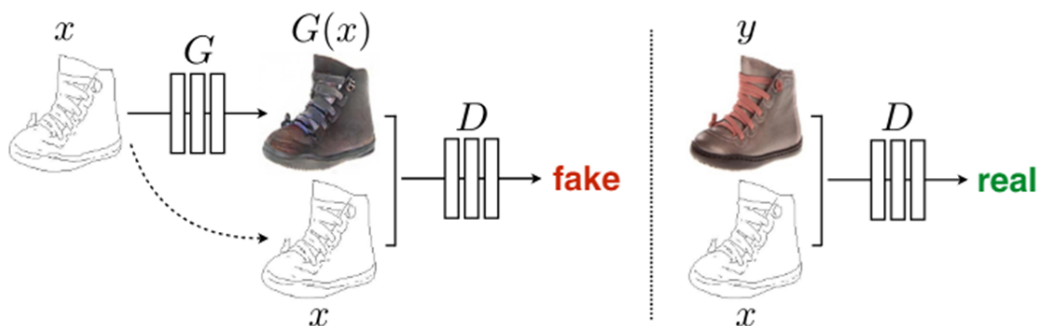

Figure 2.4: Illustration of pairing the target data with generated/real instances. Source:[21]

architecture the objective function of the model (which originally was described by Eq. 2.24) changes. The Eq. 2.29 models the relationship presented in the figure mathematically.

$$\mathcal{L}(G, D) = \mathbb{E}_{x,y}\left[log(D(x, y))\right] + \mathbb{E}_{x,z}\left[log(1 - D(x, G(x, z))\right] \tag{2.29}$$

It has been proven that the translation can achieve better results if the objective function takes into account the distance between the translated image $G(x, z)$ and the target image y. Specifically, in [38] the researchers show that the usage of L2 distance improves the results. Isola et al. opt for using an L1 measurement, as they argue it proves to be better for specific image cases. After the introduction of the distance metric the optimal generator is selected by optimization of the loss described with Eq. 3.2.

$$G^* = arg\ \min_G \max_D \mathcal{L}(G, D) + \lambda \mathcal{L}_{L1}(G)$$

$$Where \tag{2.30}$$

$$\mathcal{L}_{L1}(G) = \mathbb{E}_{x,y,z}[||y - G(x, z)||_1]$$

λ is a regularization parameter, that regulates the strength of penalization of the instances, which are far from the target distribution. The larger the λ the bigger the penalty for deviations. It can be treated as an additional hyperparameter, that needs to be tuned.

3. Method

This chapter describes the proposed framework designed to mitigate the performative drift. At first, the initial simulation design is presented, and then its preliminary results are analysed. They constitute the foundation and motivate the further presented ideas. Section Section 3.2 depicts the components of the newly created architecture. Later, mathematical equations describing the training procedure are listed with explanations. Finally, we describe each component of GDAN in detail and present the layer setups.

3.1 Preliminary simulation design

3.1.1 Goal of the simulation

The simulation aims to answer the sub-research questions by providing answers to the following:

1. How quickly does the performance of the original logistic regression model deteriorate when data distributions are influenced by the performative drift?
2. Is it possible to create a representation of data, that will allow for slower deterioration than the one of the original model?
3. Is the performance of the classifier trained on the representation created by the feature extractor superior to other methods?
4. How well does the mapping work? Visual inspection is performed. Each method is examined whether it is capable of reversing the influence of the performative drift.

To answer those questions, the results will be compared method to method and then conclusions will be drawn. For each method, accuracies will be collected and compared. Also, it is important to be able to understand what happens to the

data during the transformation. Principal Component Analysis [20] will be utilized and each data distribution will be visualized. Additionally, boxplots showing how a stretch of each feature has changed after drift has been induced will be created. This way the effects of the simulations can be observed with a bare human eye. The drawn conclusions will later be used to devise a method that synthesises concepts and extracts the best features of the examined methods.

3.1.2 Simulation design

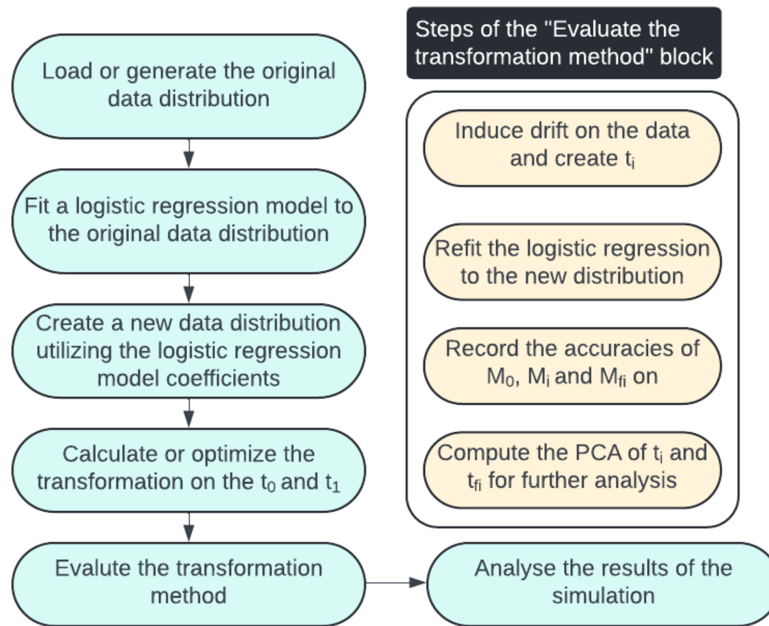

Figure 3.1: Flowchart of the simulation process

.

Section Section 2.1.5 describes multiple data generation processes the simulation will be performed for the generators designed by Perdomo et al.[39] and Izzo et al. [22].

The simulation begins by generating synthetic data using performative data generators, depending on the technique it might be based on the available datasets that mimic real-world data such as *Give Me Some Credit* [8] or fully synthetic. This data from now on will be referred to as the original data distribution and serves as the foundation. Initially, a logistic regression model is fitted to this dataset, establishing a baseline for performance evaluation. Subsequently, the trained model's parameters are used to influence the original data distribution. The nature of this influence is dependent on the scheme of a data generator, however, all of them utilize model parameters to move the data distribution. One iteration symbolizes one movement, and iteration will be noted with the letter t, meaning that the original distribution gets the symbol t_0, a distribution, that has been moved once t_1, twice t_2, and in the same manner for further iterations. After the new dataset, influenced by the drift, is created, the logistic regression model is retrained on the modified

dataset to adapt to the evolving environment. Next, based on the original data distribution and data from iteration number one, a feature transformation is computed. Then another logistic regression classifier is trained in the space created by the feature transformation. In each iteration four distribution/model combinations are evaluated:

- Model M_0 on the distribution t_i
- Model M_0 on the distribution t_{fi}
- Model M_i on the distribution t_i
- Model M_{f1} on the distribution t_i

To explain the above notation, i denotes the number of iterations or the number of times the distribution has been moved. The subscript f means that either the model has been trained in the transformed space or the distribution t_i has been transformed. The main metric used in the simulation is the model's accuracy, the accuracies are collected for all the combinations listed above and later compared. Figure Figure 3.1 visualises the simulation process.

3.1.3 Preliminary results and motivation for the architecture design

The applied methods did not provide meaningful results, a very brief summary of the results of the simulation described in the section above is provided in Table 3.1. The values in the table represent the accuracies of the model M_0 on the transformed feature representation. The results were produced by performing the simulation on a slightly adjusted Perdomo generator (described in more detail in Section 4.1.1). The baseline accuracy for that case is around 72% and with retraining of the logistic regression model, the accuracy can be sustained.

Iter No \ Method	**ARC-t**	**JDA**	**DANN**
1	50	48.05	45.52
2	53	47.5	45.52
3	54	53	45.52
4	53	50	45.52
5	52	48.5	45.52
6	50	46	45.52
7	50	46.5	45.52
8	50	49	45.52
9	55	44.5	45.52
10	52	43.5	45.52

Table 3.1: Results of different methods over 10 iterations, evaluation of M_0 on the distribution t_{fi}

What can be inferred from the results presented in the table above is the fact that those methods do not provide a valid transformation. Even though, Table 3.1 only presents the accuracies, the results of the PCA analysis were also unsatisfactory. The representation that is calculated with those methods does not help the original logistic regression to regain its performance. It can not be understood by it. Most of the methods are not significantly better than pure guessing, as the accuracies oscillate around 45-55%. However, there is one method that is distinct from the others. Both ARC-t and JDA allow only for calculating a fixed transformation. The *Domain Adversarial Neural Network* also allows for the computation of a new feature representation but also provides a classifier. That classifier is optimized to be able to extract information from that newly created representation. The accuracies of the label classifier compared with the accuracies of the original logistic regression model trained on t_0 are presented in Table 3.2.

Iter No \ Method	**Label classifier**	**Original LR**
1	**72.1**	**71.64**
2	**72.09**	**71.14**
3	**72.04**	70.1
4	**71.99**	69.1
5	**71.96**	68.06
6	**71.95**	66.96
7	**71.98**	66.02
8	**71.91**	65.19
9	**71.98**	64.27
10	**71.93**	63.2

Table 3.2: Accuracies of the label classifier predictions over 10 iterations, corresponding to evaluating M_{f1} on the distribution t_{fi}.

Comparing the results in Table 3.1 and Table 3.2 a considerable jump in performance can be observed. The decay in accuracy is still present, however it proceeds at a notably slower pace. Those results can lead to the conclusion that training a classifier that is drift-resistant with the usage of Domain Adversarial Neural Networks is achievable. As the results presented here are only preliminary, they are not used to draw overall conclusions. The goal is rather to create a foundation and later build on top of it.

Even if, training a classifier immune to drift effects would be possible, that still does not provide any information about the direction of the movement between distributions. While training such a predictor could be considered as a strategy to mitigate model drift, our research aims to uncover a mapping that explains the direction of the drift in the data. Hence, the subsequent sections will build upon the possibility of training a model proficient at handling drift and extend it with an architecture capable of mapping points back to the original distribution

3.2 Architecture design

This section will describe an architecture, that was devised to provide both a stable classifier that would be resistant towards the drift induced on the environment and a transformation from the newly acquired data to the original distribution, observed before any models were deployed. The architecture combines concepts from *Domain Adversarial Neural Networks* (Section 2.3), *Generative Adversarial Neural Nets* (Section 2.4), *Conditional GANs* (Section 2.4.1) and models used for pixel-to-pixel translation (Section 2.4.2). The goal of combining those notions is to use DANN to achieve a stable, drift-resistant classifier and the GAN to provide a model that can map the current data distribution into the original one. To combine those architectures some adjustments are needed, which is where conditional GANs and pixel-to-pixel translation methods come into play.

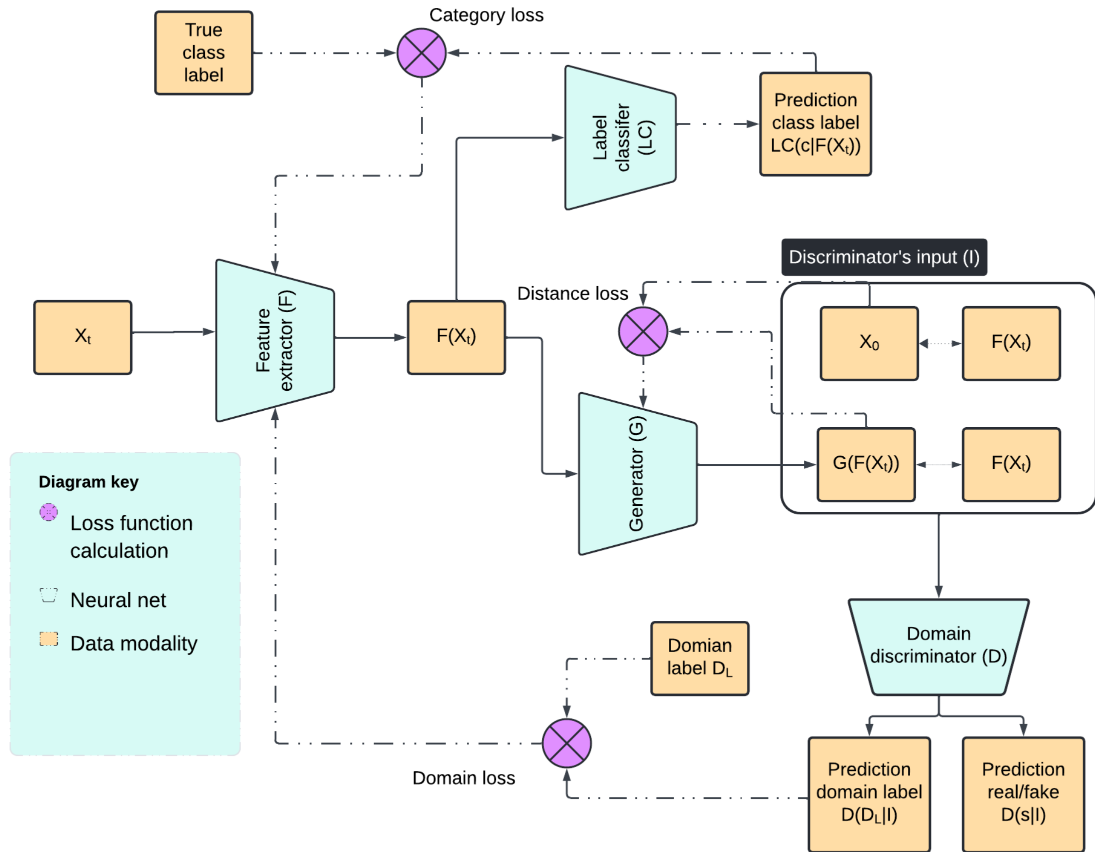

Figure 3.2: Proposed architecture for classification and mapping.

Figure 3.2 illustrates the combined architecture flowchart. Dotted lines on the figure symbolize calculations of loss functions, and the continuous lines model input/output relationships. X_t denotes a single point in time originating in data distribution t. The index t describes how many times the distribution has been influenced by the drift. Each point in a dataset can be described with a tuple (X_t, D_l, c), where X_t symbolizes the feature data, D_l corresponds to t (indicating the label of the distribution the point is sampled from). The letter c is the class, the target variable of

the classification process. First, each point is put through the feature extractor F. The feature extractor should modify the point in such a way that the origin of the point is hard to predict. So the output of the network $F(X_t)$ should have features, that are generic and non-domain specific. The transformed point is then used as an input to the label classifier LC, which predicts the correct class of the instance. That network LC is responsible for providing the stable classification process. Simultaneously, $F(X_t)$ serves as the input to the generator G. The generator's goal is to transform the given input in such a way, that the drift that has occurred is reversed. To put it in simpler terms, the generator aims to map the point back to the original distribution. The final element of the architecture is the discriminator D. It combines the two different GANs frameworks, AC-GAN [34] and pix-to-pix translator architecture [21]. This means that 50% inputs for the discriminator are real pairs of data points and the other half are the generated pairs. A pair consists of either a point from the original distribution and its non-domain-specific representation or a generated point and its non-domain-specific representation. The discriminator's task is to output two probability distributions: one predicting the origin of the point (D_l) and the other indicating whether the point is from a real distribution or generated by G.

3.2.1 Objective

The Section 3.2 aims to provide a general overview of the architecture and tasks of its components. In this section, more formal notation will be introduced to provide a stricter frame for the architecture.

The objective function of the proposed architecture is quite complex as it needs to take into account all the components. The derivation process of that function will now be considered. Starting by analysis of the discriminator's task, it is necessary to combine Eq. 2.28 and Eq. 2.29. The output of this can be described as follows.

$$L_{source} = \mathbb{E}_{X_t,X_0}[log(P(S = real|X_0, F(X_t))] + \mathbb{E}_{X_t,z}[log(P(S = fake|G(F(X_t)), F(X_t))]$$

$$L_{domain} = \mathbb{E}_{X_t,X_0}[log(P(D_l = d_l)|X_0, F(X_t))] + \mathbb{E}_{X_t,z}[log(P(D_l = d_l|G(F(X_t)), F(X_t))] \tag{3.1}$$

L_{source} denotes the log-likelihood of predicting the correct source, whether the point has been sampled from a real distribution or generated by the model. L_{domain} denotes the log-likelihood of predicting the correct domain, whether the point was sampled from a distribution with $t = 0$ or $t = 1$. The symbol z in the subscript of the expected value symbolizes the noise imposed on the generated form by the generator. It is also important to note, that if I symbolizes the concatenated inputs to the discriminator, then the discriminator output can be written as $D(I) = P(S|I), P(D_l|I)$. Similarly to what has been described in Section 2.4.1, the goal of the discriminator is to maximize $L_{source} + L_{domain}$, meanwhile the goal of the generator is to maximize $L_{domain} - L_{source}$. The generator aims to fool the discriminator by producing instances that are hard to distinguish from real ones.

Outputs of the discriminator are not crucial by themselves, they are necessary, as they enable the adversarial training process of the generator and discriminator.

First, let us consider the case of the generator. It has two tasks: the aforementioned fooling of the discriminator and producing an output that is as close to the original distribution as possible. Therefore, the generator needs to be penalized based on the distance between the target point and the point it has generated. Mathematically, this can be expressed as:

$$\begin{gathered} G^* = arg \min_G \max_D \mathcal{L}(G, D, F) + \lambda \mathcal{L}_{L1}(G) \\ Where \\ \mathcal{L}(G, D, F) = L_{source} + L_{domain} \\ \mathcal{L}_{L1}(G) = \mathbb{E}_{X_0, X_t}[||X_0 - G(F(X_t)||_1] \end{gathered} \tag{3.2}$$

Finally, let us consider the case of the feature extractor and label classifier models. The feature extractor is trained based on the loss functions of the label classifier and the correctness of predictions of the domain label produced by the discriminator. Recalling the equations Eq. 2.22 and Eq. 2.23, the approach remains mostly unchanged. The only factor that has changed is that now the discriminator is a more complex model. So for the feature extractor update rule not the entire discriminator's loss is taken into account, but only the part that has to deal with calculating the *Domain loss* (visible on Figure 3.2).

$$F^* = arg \min_F \max_D \mathbb{E}(\theta_F, \theta_{LC}, \theta_D) \tag{3.3}$$

The expected value of Eq. 3.3 is almost equivalent to Eq. 2.22 but with the aforementioned change in the calculation of the discriminator's loss function. The feature extractor attempts to maximize the accuracy of the label classifier while simultaneously confusing the discriminator, ensuring that the extracted features are generic and non-domain-specific.

Optimization of the label classifier is the most straightforward process. It follows the classic supervised learning paradigm in machine learning. The label classifier's goal is to minimize its prediction loss, which involves adjusting its parameters to correctly predict the class labels of the instances based on the features provided by the feature extractor. This process ensures that the label classifier becomes highly accurate in its predictions.

To summarize the approach, during training there are two adversarial two-player games ongoing. The first one involves the feature extractor and the part of the discriminator responsible for predicting the domain, and the second one is the generator vs the discriminator. These adversarial interactions are crucial for the overall training process. They ensure that the components of the architecture—feature extractor, label classifier, generator, and discriminator—are all optimized in a manner that leads to a stable, drift-resistant classifier capable of mapping current data distributions back to the original one.

3.2.2 Training procedure

The main focus of the previous two sections was to introduce the architecture and explain the update rules and the tasks of each component. This section will primarily focus on formulating an algorithm to clarify the order of operations and provide deeper insight into the actions executed during architecture training. The pseudocode of the process is summarised by Algorithm 1. The notation used therein follows the standards introduced in previous sections.". Additionally, θ denotes the weights of a model, with the subscript indicating the specific model. The parameter μ is the learning rate, again that can be model specific. λ_1 and λ_2 are regularization parameters, treated as hyperparameters, regulating the degree of penalization for a model. To summarize all of the training variables and hyperparameters the Table 3.3 was created.

To provide more clarification symbols L in section Section 3.2.1 refer to log-likelihood. In this case, maximization of the log-likelihood is equivalent to minimisation of the loss function, so that is why in the Algorithm the gradients are subtracted from the vectors of weights of the models. The $\mathcal{L}$ symbols used in 1 denote loss functions. Another important detail is the fact that when calculating $\mathcal{L}^{m_1}_{source}$ the target variable is a vector of ones, even though the instances are produced by the generator so they should be labelled with zeros. This means that the predictions of the discriminator are compared against artificially supplied labels. So minimisation of this function is equivalent to maximising a function supplied with true labels. That explains the addition sign "+" standing before $\mathcal{L}^{m_1}_{source}$.

Algorithm 1 Gradient Descent Training of Domain-Generative Adversarial Nets

Input: Number of training iterations N, minibatch size m, hyperparameter k (the number of steps after which the updates of the generator start and updates of the feature extractor stop)

Create a combined dataset of X_0 and X_1 : $X_{combined}$, associate each point with class labels C and domain labels D_l

for number of training iterations **do**

- Sample minibatch of $\frac{1}{2}m$ examples $\{x^{(1)}, \ldots, x^{(\frac{1}{2}m)}\}$ from the combined dataset $X_{combined}$

- Arbitrarily pair each sample with a point from the original distribution X_0

- Put each of the $\frac{1}{2}m$ examples through the network components and let them produce their outputs

- Generate a minibatch of $\frac{1}{2}m$ examples by inputting the $F(x^{(\frac{1}{2}m)})$ into the generator, assign correct domain labels, generator loss will only be calculated based on those instances - later denoted as m_1. The discriminator is trained on both real and generated images, but the generator only on the generated ones.

if n < k **then**

- Update the weights of F, LC, and D according to the following:

$$\theta_F \leftarrow \theta_F - \mu_F(\frac{\partial \mathcal{L}^m_{LC}}{\partial \theta_F} - \lambda_1 \frac{\partial \mathcal{L}^m_{domain}}{\partial \theta_F})$$

$$\theta_{LC} \leftarrow \theta_{LC} - \mu_{LC}(\frac{\partial \mathcal{L}^m_{LC}}{\partial \theta_{LC}})$$

$$\theta_D \leftarrow \theta_D - \mu_D(\frac{\partial (\mathcal{L}^m_{source} + \mathcal{L}^m_{domain})}{\partial \theta_D})$$

end if

if n >= k **then**

- Update the weights of G and D according to the following:

$$\theta_G \leftarrow \theta_G - \mu_G(\frac{\partial (\mathcal{L}^{m_1}_{domain} + \mathcal{L}^{m_1}_{source} + \lambda_2 \mathcal{L}_{L1})}{\partial \theta_G})$$

$$\theta_D \leftarrow \theta_D - \mu_D(\frac{\partial (\mathcal{L}^m_{source} + \mathcal{L}^m_{domain})}{\partial \theta_D})$$

end if

end for

Output: Trained feature extractor F, label classifier LC, generator G and discriminator D

Parameter	Description	Type	Notes	Value
θ_D	Feature extractor weights	Variable	Model-specific	-
θ_{LC}	Label Classifier weights	Variable	Model-specific	-
θ_D	Discriminator weights	Variable	Model-specific	-
θ_G	Generator weights	Variable	Model-specific	-
$\mathcal{L}^m_{LC}$	Label classifier loss on a batch m	Variable	Batch-specific	-
$\mathcal{L}^m_{domain}$	Discriminator domain loss on a batch	Variable	Batch-specific	-
$\mathcal{L}^m_{source}$	Discriminator source loss on a batch	Variable	Batch-specific, prediction whether real or fake	-
λ_1	Regularization parameter 1	Hyperparameter	Controls the importance of domain penalty	0.01
λ_2	Regularization parameter 2	Hyperparameter	Controls the distance penalty between the generated and the target	100
μ_F	Learning rate F	Hyperparameter	Model-specific	0.001
μ_{LC}	Learning rate LC	Hyperparameter	Model-specific	0.001
μ_G	Learning rate G	Hyperparameter	Model-specific	0.001
μ_D	Learning rate D	Hyperparameter	Model-specific	0.0001
N	Number of training steps	Hyperparameter	Data generator specific	-
m	Batch size	Hyperparameter	Unified for all cases	1024
m_1	Batch size for training of the generator	Hyperparameter	Unified for all cases	$\frac{m}{2} = 512$
k	Training threshold	Hyperparameter	The number of completed steps before the generator updates start (the number of updates of F and LC)	-

Table 3.3: Summary of training variables and hyperparameters

3.2.3 Network architectures

In this section, the composition of each network will be explained. A reasoning behind the choice of layer types, activation functions, etc. will be provided. The goal is to provide more clarity into the task that each component aims to learn.

3.2.3.1 Feature extractor

Figure 3.3 visualizes the layers the feature extractor network is composed of. The data that this research deals with is 1-dimensional, meaning that each point has a certain amount of features, but it can also be represented as a vector of size - (number of features) x 1, in this case the number of features is equal to 11. The goal of the network is to create a representation with domain-invariant features, that is of the same size as the initial representation, so it can be used by the originally trained logistic regression classifier. The first two layers are Transposed Convolutional layers, their goal is to get each point to a higher dimensionality representation [11]. The first layer has 32 feature maps, a kernel of size 6 and an applied stride is 2, this transforms from the dimension of (1,11,1) to (32,26,1). Second, transposed convolutional layer further upsamples the data to (108,64,1), by operating with kernel size 8 and stride 4. The goal of projecting data to higher dimensions is to extract patterns which might not be noticeable in its regular form. The rest of the process is downsampling the data back to its original shape, by performing standard convolution with kernel size 6 and stride 14. That is followed by average pooling, so taking a rectangle of size 37x1 averaging the values in that area and then sliding that rectangle by 20 units. The output of that operation is again of size (1,11,1), it gets reshaped and connected to one final dense layer and finally, it is normalized. Additionally, batch normalization layers are added after transposed convolutions to make the learning process smoother [2]. After every single convolution, a rectified linear unit is used, as that function is commonly utilised for convolutional neural networks [33].

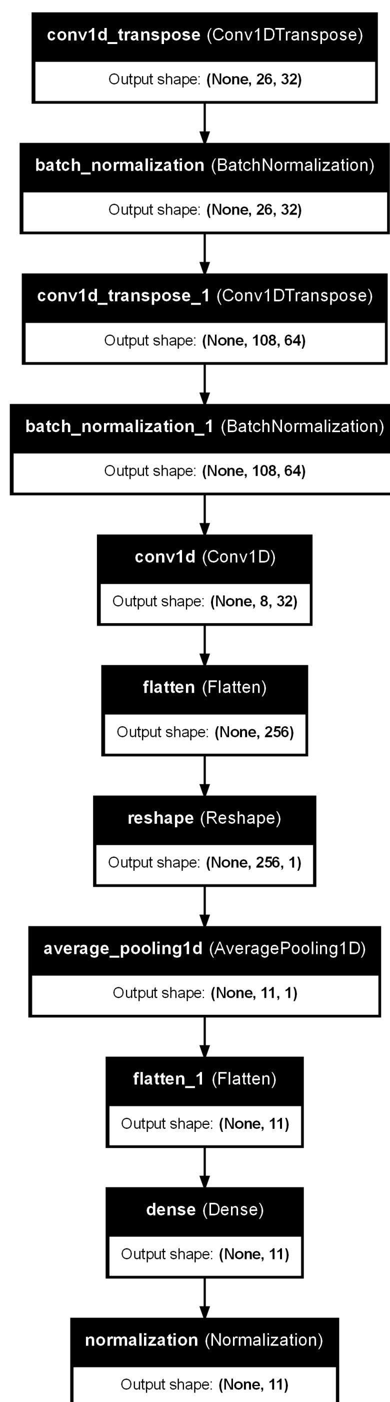

Figure 3.3: The architecture of the feature extractor network.

3.2.3.2 Label Classifier

The task of the label classifier is more straightforward compared to the task of the feature extractor, due to that fact, the architecture can also be more simplistic. The label classifier is just a simple feed-forward neural network [35] with four layers. The

number of parameters follows a pattern of 256-512-64-1, each dash separates layers. The final layer is of size one as it has to output the binary probability, the activation function for the final layer is sigmoid, as it is commonly used for binary classification problems [43]. After layers dense_1, dense_2 and dense_3 (so the first three fully connected layers) the rectified linear unit function is applied.

3.2.3.3 Generator

The layers of the generator network have been visualised on Figure 3.4, the goal of the network is similar to the feature extractor's aim described above. The idea for the layers setup was inspired by the U-Net architecture [40]. Ronnenberger et al. devised a model that first downsamples/encodes the input, to locate the spatial features and later upsamples/decodes it to create a segmented translation of the input. Their work also utilises skip connections to better match the features between the encoding and decoding. That solution has been mostly applied to image segmentation problems. In the case of this research, where we deal with one-dimensional data the architecture needs to be adjusted.

The concept is reversed, so first each data point (the output of the feature extractor) will be transformed to a higher dimension and later transformed back to the original dimensions. This is achieved again by the usage of first transposed convolutional layers and later regular convolutions. Figure 3.4 shows how the dimensions of the data point change as it is fed through the network. Below more detailed information about each layer parameters is presented.

- C-Transpose_2 **kernel size**: 4 **stride**: 2
- C-Transpose_3 **kernel size**: 4 **stride**: 4
- C-Transpose_4 **kernel size**: 4 **stride**: 4
- Convolution_1 **kernel size**: 8 **stride**: 4
- Convolution_2 **kernel size**: 6 **stride**: 2
- Convolution_3 **kernel size**: 4 **stride**: 2

Finally, after those operations are performed the output is flattened and connected to one final dense layer of size number of features x 1, in this

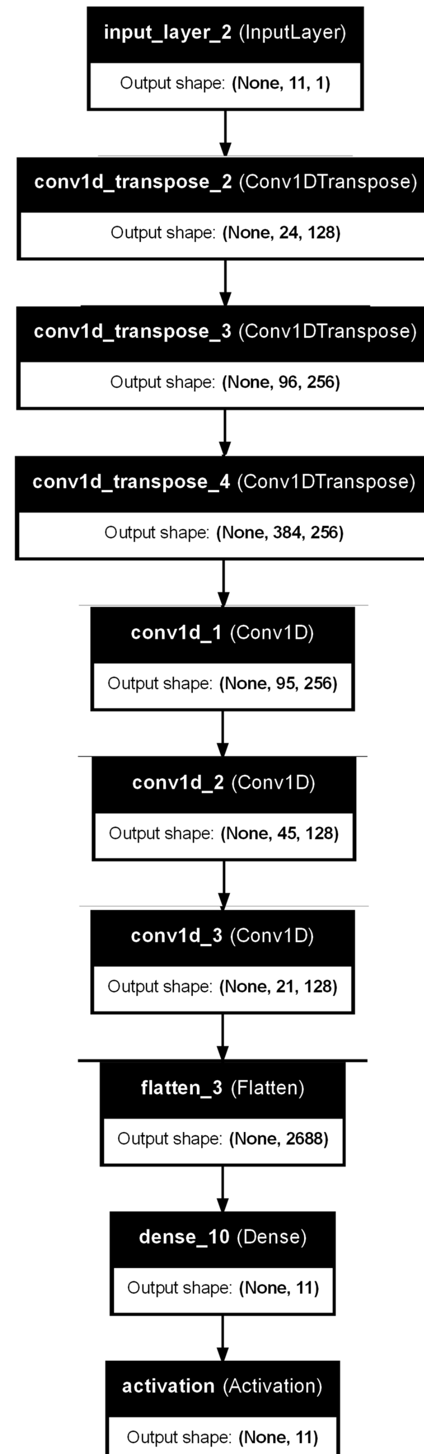

Figure 3.4: The architecture of the generator network.

case (1,11,1). The activation function applied to the final output is just linear, so the generator can replicate any values and is not restricted to a strict range.

An important detail, that is not depicted on Figure 3.4 is the fact that after each transposed convolution or a regular convolution, to optimize the training process and prevent vanishing or exploding gradients, batch normalization is performed. Each one of those layers has also a LeakyRelu activation function, it is used for the same reason. It has been proven that the usage of LeakyRelu can prevent gradient death [24].

3.2.3.4 Discriminator

The discriminator is presented with two tasks as described before (Section 3.2, it also takes two inputs. Due to that, the first layer concatenates the inputted representations. What follows, is four fully connected dense layers (characterized by the following numbers of parameters 64-128-512-512). Between the fully connected layers, there are several regularization operations performed, the aim of which is to prevent overfitting and stabilize the training process [4]. To list them, after layer dense_5 batch normalization. After each layer a dropout of 25%, which means that during training random 25% of connections between layers are treated as if they did not exist [44]. Also, during the forward propagation of an instance each layer uses a LeakyRelu activation function. Finally, after dense_5 the data is flattened and two output layers are plugged in. The first one is responsible for predicting whether the input pair originates in the original distribution or has been generated by the generator. The second one is responsible for predicting the domain label. Both of the output layers use a sigmoid activation function.

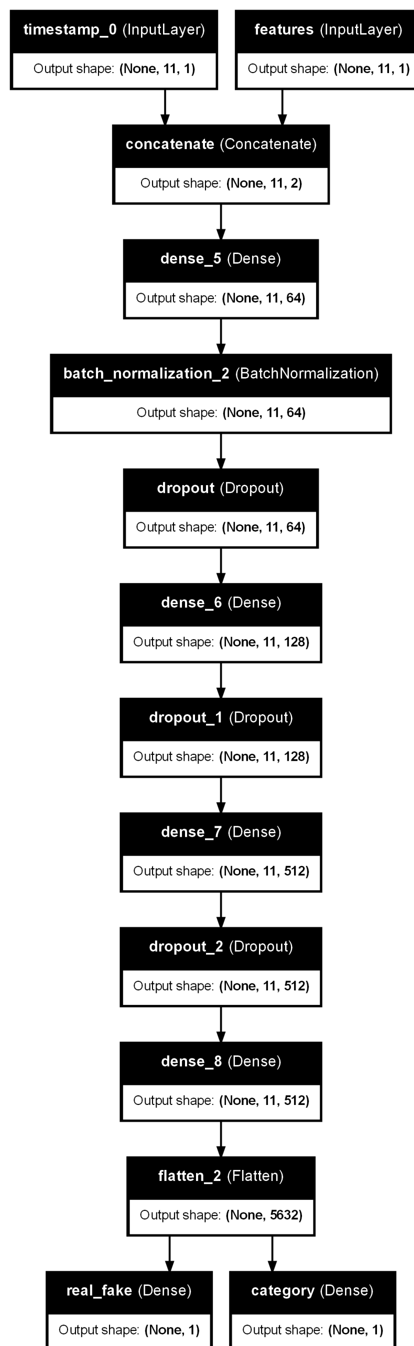

Figure 3.5: The architecture of the discriminator network.

4. Experiment and Results

This chapter will describe the experiment setups and show the results method described above. To generalize the results as much as possible, the method has been tested on three data generators. The review of generators present in the literature has been presented in Section 2.1.5. Each generator can induce different types of drift on the data, which is why conclusions should not only be drawn based on one example. Additionally, as the differences between the data-generating processes are significant, they might require different experimental setups. For instance, it might not be feasible to match datapoint from the distribution $t = i$ with its corresponding format originating in $t = 0$. The table Table 4.1 presents the major differences between the data generators. Furthermore, different training strategies may offer better results considering the nature of the data-generating process, that fact is also considered in the later subsections of this chapter.

The metrics that will be taken into account are the accuracy of the *Label classifier*, the accuracy of the logistic regression model and the accuracy of the original logistic regression with the data that was produced by the *Generator*. Apart from the accuracies, the distances between the distributions t_i, $G(F(X_{t_i}))$ and t_0 are also analysed. To clarify the notation, the following symbols are introduced $L_1(t_i, t_0) = \sum_{j=1}^{n} |t_{i,j} - t_{0,j}|$ and $L_1(G(t_i), t_0) = \sum_{j=1}^{n} |G(t_{i,j}) - t_{0,j}|$.As the distance metric does not ensure the fact that the distributions are identical, PCA will be performed on each of the above. Additionally, if it is possible a boxplot showing how the datasets have changed will be shown. This way, it can be observed whether the generator can reverse the effects of the drift.

Data generator	Point-to-point matching	Constant type of drift
Perdomo - v1	feasible	yes (no retraining)
Perdomo - v2	feasible	no (LR retrained)
Izzo	not feasible	no (LR retrained)

Table 4.1: Comparison of data generators

4.1 Perdomo generator

4.1.1 Experiment setup

Let us recall the equation Eq. 2.6, which shows the main rule behind how the new data distributions are created. In [39] every time a new classifier is deployed the starting distribution t_0 is changed and a new dataset is created.

$$x_{t+1} = x_0 - \epsilon B\theta \quad \text{(Eq. 2.6)}$$

For the sake of this research, two versions of this data generator will be tested, to see if the model can cope with changing direction of the drift. Both of them are based on the generator devised by Perdomo et al., which was used in multiple papers available in the literature. The scenarios differ from each other, because of the retraining strategy. The first one deploys just one classifier and during the second one the predictor is retrained every iteration.

That first scenario deals with a situation, where one predictor is deployed and it keeps influencing the environment. So still every iteration new data distribution is created, however, the type of movement between them does not change. That is achieved by implementing the following equation into practice. In that scenario, the logistic regression is not retrained, so the parameter vector θ is constant.

$$x_{t+1} = x_t - \epsilon B\theta \quad (4.1)$$

The second scenario is very similar just every iteration t, θ is retrained according to the state-of-the-art method of minimizing loss of the logistic regression. A key mechanism in both of those data-generating processes is the fact that the points are modified. Due to that, in the stage of training, matching points one-to-one is easily implementable. Eq. 2.6 and Eq. 4.1 include a symbol B. It denotes the matrix (filled with 0s and 1s) that determines, which features are strategic - therefore can be influenced by the model drift. In this research, the original approach is followed and the strategic features are the ones with indexes belonging to [1,6,8]. Table 4.2 summarizes all parameters of the generator with their description and values.

Parameter	Meaning	Value
ϵ	The strength of performativity	10.0
n	number of features	11
B	performative array,, informs whether feature is prone to drift	-
no samples train	number of samples generated for training	18357
no samples test	number of samples generated in each testing iteration	10000
no iters	number of testing iterations	10
k	Table 3.3	350
N	Table 3.3	100 epochs = 3500 steps

Table 4.2: Summary of the data-generation process parameters.

4.1.2 Perdomo learning curves

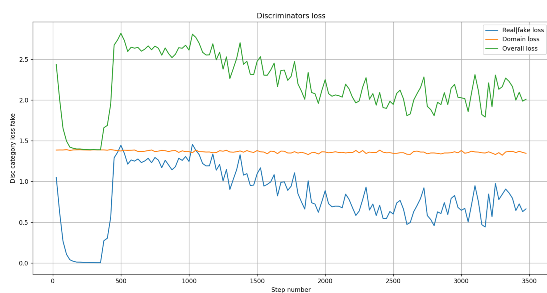

Figure 4.1: The loss of the discriminator over training.

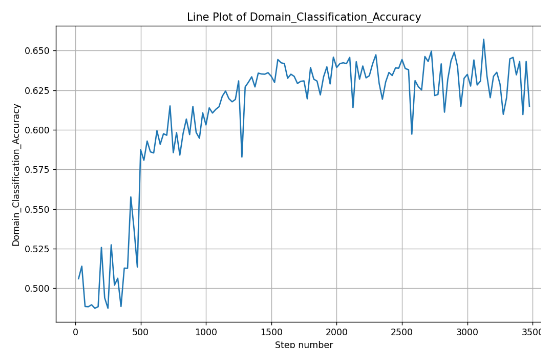

Figure 4.2: The domain accuracy over training

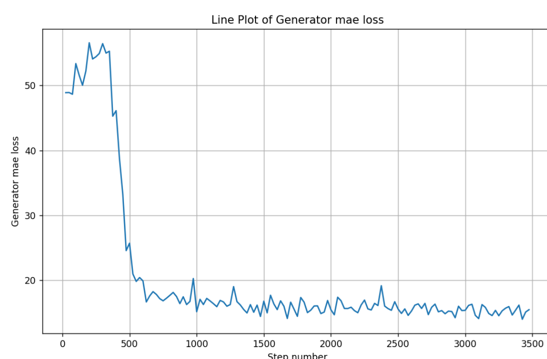

Figure 4.3: The distance loss of the generator over training

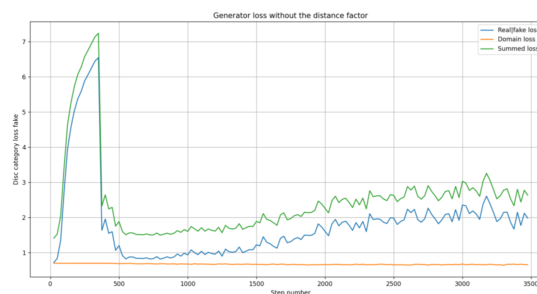

Figure 4.4: The adversarial loss of the generator over training

Figure 4.5: Learning curves of the subparts of the entire architecture

This section provides more insight into how the training process ran. A difficult part, when it comes to training the model's architecture is the fact that it should be performed in an adversarial manner. This means that devising a stop criteria, such as no improvement on the loss function or non-significant improvements, is challenging. The process was supervised differently, by analysing learning curves after an arbitrary number of epochs and comparing the results.

Figure 4.1, Figure 4.2, Figure 4.3, Figure 4.4 illustrate either losses or accuracies achieved over training. On all of the figures, a switch in behaviour can be noticed, as that is when the update rule is changed (described in 1). Considering the loss of the discriminator, it can be seen that until the 350th step, it completely overpowers the generator as until then generator is not learning. But later once the generator starts updating its weights and gets better at creating fake images, the loss of the discriminator spikes rapidly. For the rest of the process, the curve fluctuates to a significant degree, which is understandable as the two networks compete with each other. An analogous process can be observed on Figure 4.4. It seems that in the end, the discriminator is a bit more powerful than the generator, but not to a degree that would harm the results. The learning rate of the discriminator is 10 times higher than the one of the generator, however, setting that value of the hyperparameter

was the only discovered setup, in which the generator was not overpowered by the generator. Another observation can be derived from Figure 4.2. Until the updates of the feature extractor are not switched off the domain classification accuracy oscillates around 50%, meaning that the feature extractor can fool the discriminator. That is necessary so that the created representation is domain invariant. Later the discriminator starts to learn to tell the origin of an instance apart. Finally, as for the distance penalty of the generator (illustrated on Figure 4.3), as soon as the generator is allowed to start learning it starts minimising that distance. The pace of this process is very quick at the beginning and later the network starts to struggle and it seems to be stuck close to a local optimum. That could indicate that the model is not complex enough for the task it is presented with.

4.1.3 Perdomo results

4.1.3.1 Simulation with one classifier

This section presents the results of the experiment, where only one classifier is deployed, meaning that θ stays constant. Figure 4.6 illustrates the results of the Principal Component Analysis. That technique is used so that the dimensionality of data can be reduced and it could be visualized on a 2D plot. In essence, the principal components are the directions, that capture the most variance in the dataset. For this analysis, the 2 directions with the largest variance are captured and visualised. Applying this method allowed for the creation of plots like those in Figure 4.6.

The figure depicts the first, the last and the 5th iteration's clusters. Analysing them in depth, it can be seen that the blue and red clusters are overlapping throughout the simulation. The reason behind that might be the fact that only three out of eleven features of the dataset are modelled as strategic. That is why, Principal Component Analysis struggles to show differences between the red and blue clusters. The green clusters, which are the visualization of generated data, are not stretched enough. It seems like they try to imitate the original distribution, but the model has not learnt enough to be able to introduce enough variance into the created dataset. A grasping observation is that there is no visible deterioration of the green clusters over the iterations. The replication of the starting data seems to be similar, regardless of the number of times, a distribution has been moved. That could suggest that, due to the constant direction of the drift, the model can still provide meaningful mappings even in later iterations.

Figure 4.7 depicts how the accuracies of different models on different representations change as new data distributions are generated. The grey dotted line shows the baseline accuracy. It symbolizes the correctness of a classifier trained on t_0. This model is used to change the data. The grey line is shown on the graph just for reference to indicate how well the model performed initially. The red line represents the accuracy achieved with the *Label classifier*, it can be observed that initially, its accuracy is even better than the baseline. Unfortunately, it does deteriorate quickly. After four iterations the model is practically outdated and is being outperformed even by the original logistic regression predictor (the blue line on the graph). The green line illustrates the performance of the original LR on the representation created by the *Generator* and as can be seen even though that accuracy starts decreasing

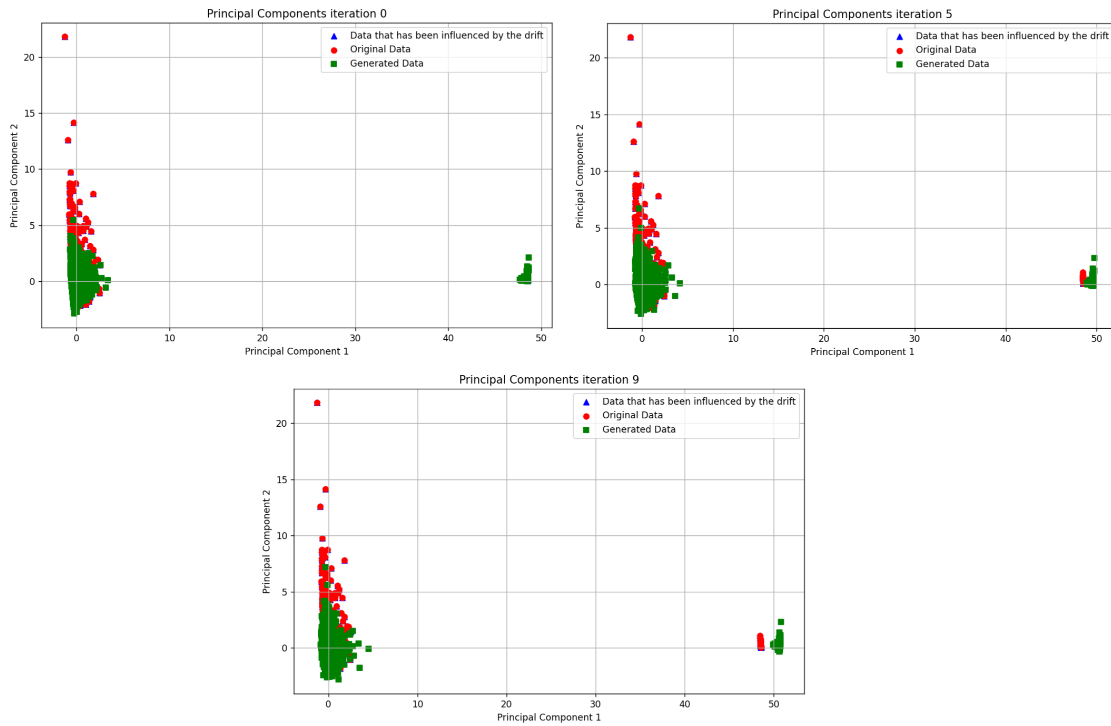

Figure 4.6: Results of the PCA analysis - generator Perdomo v1.

in later iterations at the beginning it matches the baseline almost perfectly. This means that the created mapping is sufficient for the original model to, for at least a few iterations, maintain its performance. After four movements have occurred, the accuracy starts to decline. However, this decline is notably less steep compared to the other models. The retraining accuracy (orange line), is constant and at some point outperforms all other predictors. It is important to state that this model possesses a certain advantage, as it is retrained every iteration. Compared to the other only being trained once at the very beginning of the simulation.

Figure 4.8 aims to show the correlations between the distances $L_1(G(t_i), t_0)$, $L_1(t_i, t_0)$ and performance of the *label classifier* and the *generator*. The first fact, that should be observed is that the red line depicting $L_1(t_i, t_0)$, has a constant slope, which shows that indeed the changes in distance between distribution $t_i - t_0$ and $t_{i+1} - t_0$ are very similar. This shows that the drift induced in every iteration has a consistent direction. The second observation is that the distances are not correlated with each other. The red and purple curves seem to be diverging from each other, and the slope of the curve picturing distance $L_1(G(t_i), t_0)$ is significantly less steep. Even when the distribution t_i is quite far from the distribution t_0, the generator can create a representation, that is located not far from the starting distribution. This could indicate, that in this case the generator has learnt the direction of the drift and is capable of reversing it throughout the simulation, regardless of the number of times it has been induced on the data.

Finally, as the Figure 4.6 does not provide that much insight into how well the generated dataset resembles the starting distribution, boxplots visualizing each feature of the datasets have been created. Figure 4.9 and Figure 4.10 compare present the

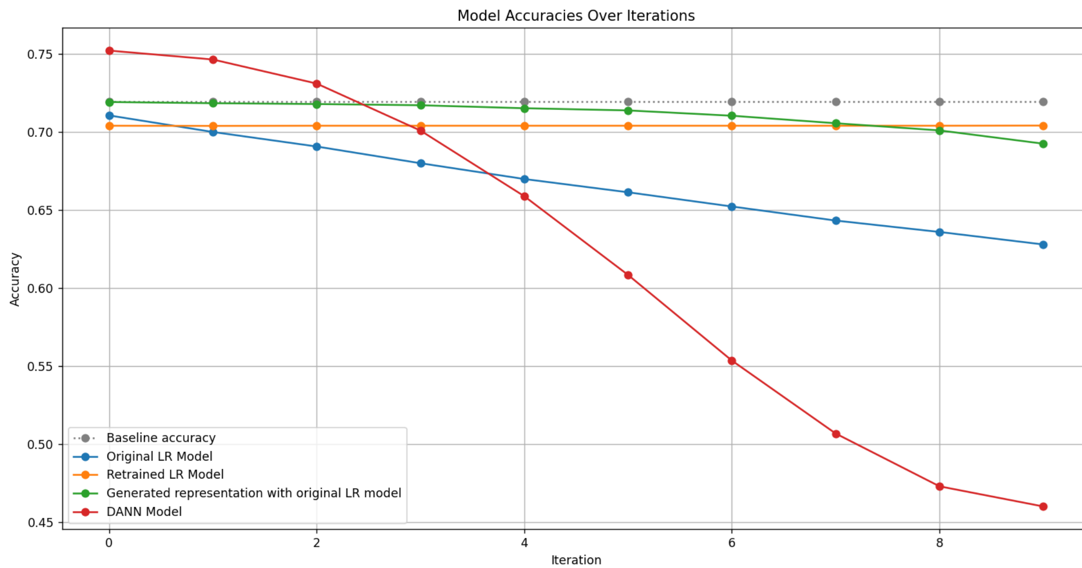

Figure 4.7: Linegraph of accuracies - generator Perdomo v1.

spread of each feature and compare those between t_0, t_1 and t_9. At the same time, they show how well the *Generator* reverses the drift. First of all, recalling the fact that only three features are being drifted, the drift is only observable when looking at features with indexes (1,6,8). The effect of the drift is most clearly visible in the plot of feature eight. Giving this subplot a more in-depth look, in both iterations 0 and 9 the *Generator* pushes the median in the correct direction. Once again this shows that it is capable of providing a meaningful mapping. However, an interesting incident occurs when we look at the feature 6. In the first iteration, the created representation is not an accurate mapping, but it seems like it should be heading towards the correct direction. Surprisingly, in the ninth iteration, the median is moved completely to the wrong side. This could explain why the deterioration in

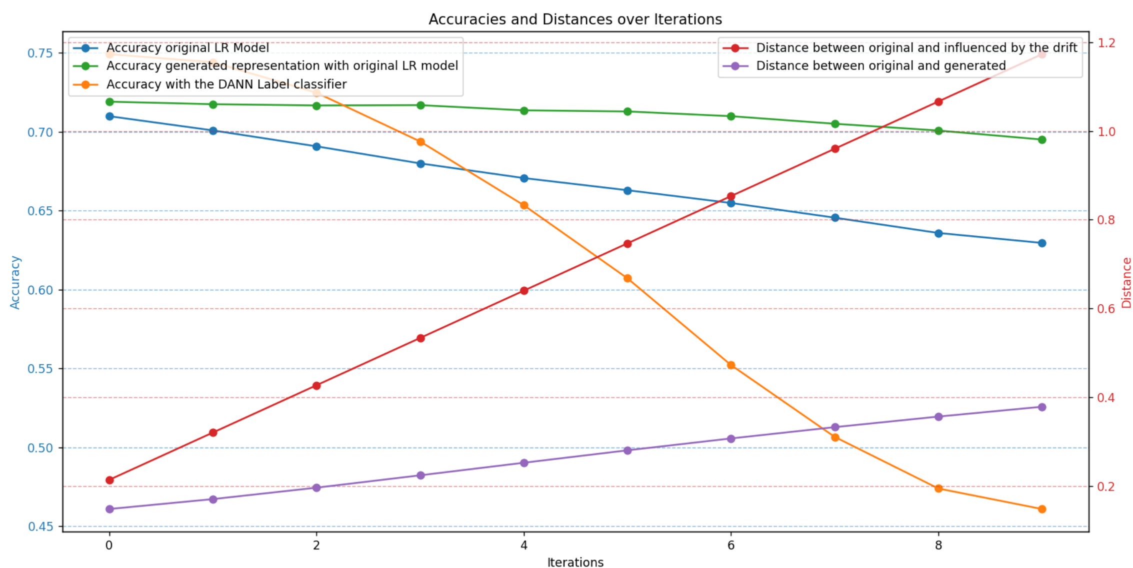

Figure 4.8: Correlation between accuracies and distances - generator Perdomo v1.

performance is observed on Figure 4.7. Overall, the boxplot illustrates that the model struggles with introducing variance into the created datasets. Features with indexes two and ten are replicated more accurately, as they are less stretched and more centred around their median. The features, which have plenty of outliers such as features one or five, are considerably more difficult to replicate. To accompany those statements with numbers, if taken the maximum range for features 2 and 10. It would be $[-2.5; 6]$ if the same operation were performed for features 1 and 5 it would be $[0; 40]$, which explains the differences in performance.

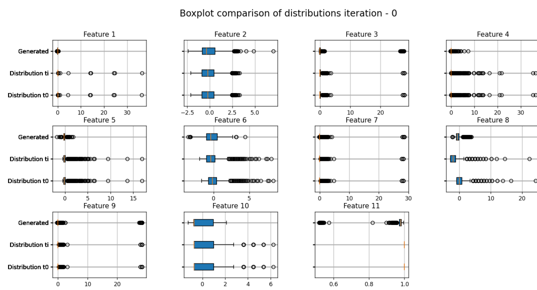

Figure 4.9: Boxplot visualising differences between distributions - iteration 0

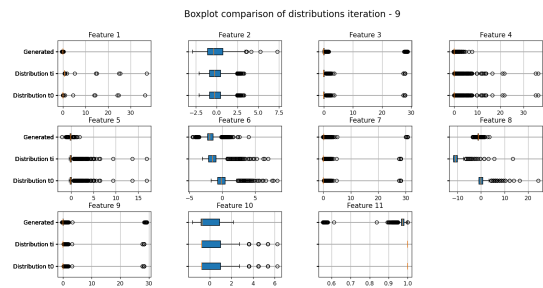

Figure 4.10: Boxplot visualising differences between distributions - iteration 9

4.1.3.2 Simulation with retraining

This section shows the result of the experiment, where θ is updated every iteration. Let us begin with the PCA metric analysis. The results achieved in this version of the experiment are extremely similar to what has been presented on Figure 4.6, that is why the PCA plot for this version of the experiment is not included here. The green clusters are not elongated enough, which could mean that the complexity of the task is too large for this architecture. And the deterioration in resembling the starting distribution is very similar. This can be caused by the fact that even though the logistic regression is retrained every single iteration, the parameter changes are not so rapid, they are rather small adjustments. The result of this is that the results are rather alike for both version of the experiment.

Figure 4.11 shows the values of accuracies of different models and representations. The accuracies achieved by the trained architecture in this scenario are slightly less promising than the ones achieved before. At iterations 0 to 2, both the *label classifier* and the *generator* perform reasonably well. However, as soon as the classifier gets retrained three times and the direction of the drift changes considerably, the accuracies start deteriorating. The pace of the drop of the performance of the *label classifier* is more rapid. Compared to the original logistic regression (blue line), the outputs of the proposed architecture still seem to provide better classification and reasonable mappings for the first few iterations. However, in this scenario, it seems that retraining the logistic regression might be a superior method. Throughout the entire simulation, it almost matches the baseline accuracy, achieved at t_0.

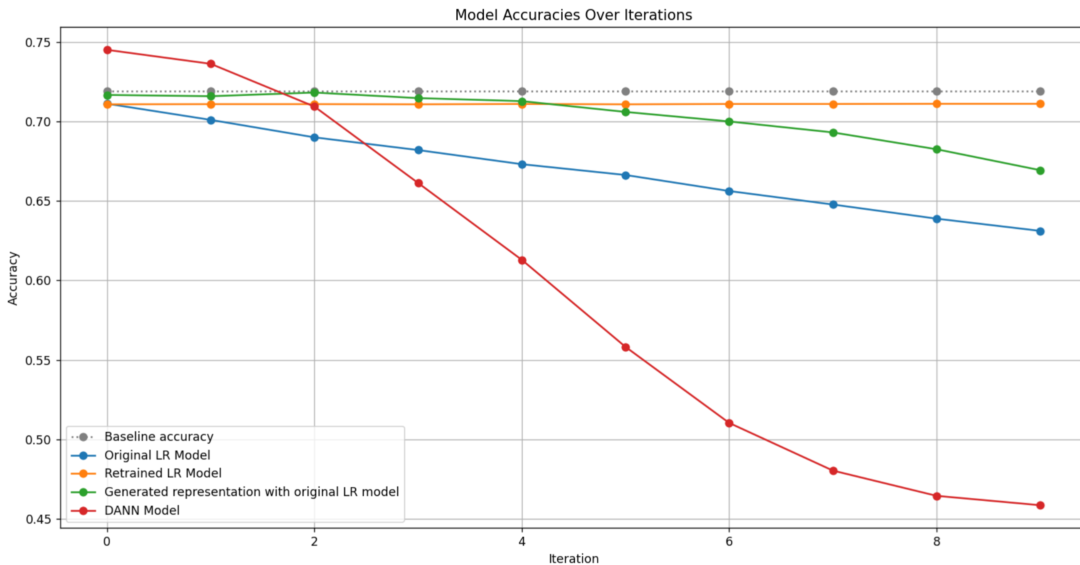

Figure 4.11: Linegraph of accuracies - generator Perdomo v2.

Moving on, the Figure 4.12 will be discussed. That figure depicts a few interesting relationships, which could explain the behaviour of the model. First of all, it is important to notice that the *generator*, especially at the beginning succeeds at minimizing the distance between the distributions. On the graph, the purple line is always significantly closer to 0 compared to the red line. The second correlation is that as soon as the distance $L_1(G(t_i), t_0)$ (purple) starts to increase quicker, the pace of decrease in performance of the original logistic regression tested on the generated data starts accelerating. A very similar correlation can be observed between the distance $L_1(t_i, t_0)$ (green) and the accuracy of the *Label classifier*. As soon as the distance increases, the features extracted by the *feature extractor* are not understood that well and it leads the an increased rate of misclassifications.

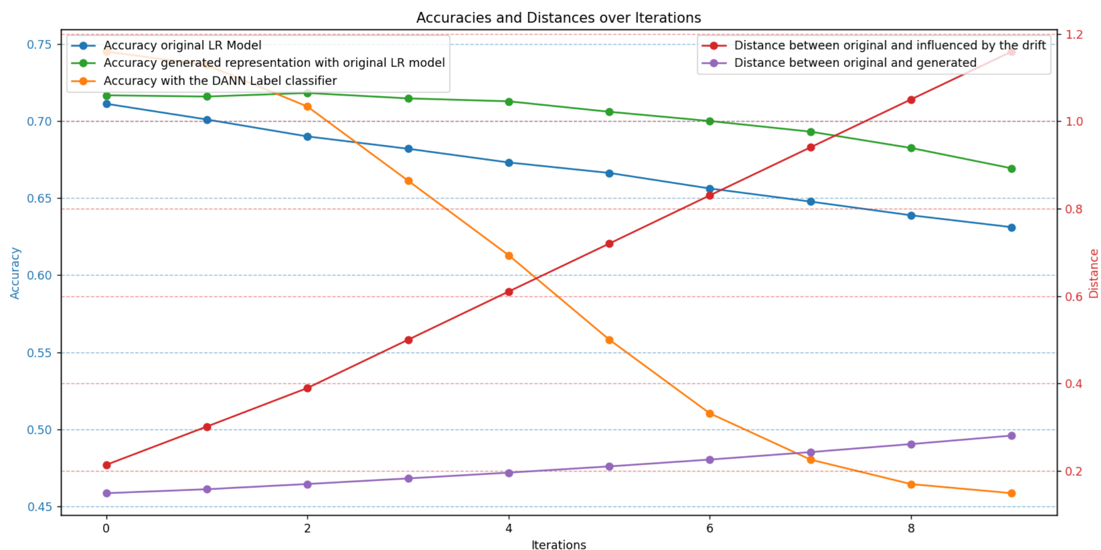

Figure 4.12: Correlation between accuracies and distances - generator Perdomo v2.

Finally, if a boxplot such as the ones on Figure 4.9 and Figure 4.10 was created, the results presented on it would be almost identical. The reason for this is again most likely the slight changes in the parameters of the logistic regression. During the retraining, the change in parameters only needs to account for changes in the three strategic features and not the entire dataset. This causes the performance visualised in the graphs above to deteriorate. However, the changes are not severe enough to be visible either on the PCA clusters or the boxplots, which is why they were not visualized in this section.

4.2 Izzo generator

4.2.1 Experiment setup

Let us recall the Eq. 2.8, which described the data-generating process. The main goal is to model a real-world spam detection situation, where only one class (the spammers) try to modify their behaviour, to deceive the predictor.

$$x|y=0,\theta \sim \mathcal{N}(\mu_0,\sigma_0^2) \qquad x|y=1,\theta \sim \mathcal{N}(f(\theta),\sigma_1^2) \tag{Eq. 2.8}$$

That is achieved, by sampling points from two different distributions one that remains unchanged over the iterations. And another one, the mean of this distribution changes according to the vector of parameters of the currently deployed classifier - θ. The Eq. 4.2 describes how exactly the new value of the distribution mean is calculated, μ_1 is a constant value equivalent to the mean of the starting distribution t_0, ϵ is a parameter describing performativity strength. To put it into simpler terms, ϵ controls how severe the influence of the model is. The larger its value, the bigger the distance between t_i and t_{i+1}.

$$f(\theta) = \mu_1 - \epsilon \cdot \theta \tag{4.2}$$

In [22] the performed experiment was using a one-dimensional dataset ($x \in \mathbf{R}$). For the sake of this research that was adapted so currently, the dataset can have an arbitrary number of features ($x \in \mathbf{R^n}$). An additional assumption is that not all of the features are prone to drift (similar to the Perdomo generator). This modification is described by the following equation.

$$\mu_{i,n} = f(\theta_n) = \mu_1 - \epsilon \cdot \theta_n \cdot B_n \tag{4.3}$$

Eq. 4.3 shows that for the spam class, we get n means in each iteration. B_n is a parameter that denotes whether the feature n is performative or not. In this experiment features from where $n \in [0,5]$, are performative, so $B_n = 1 \; for \; n \in [0,5]$ and else $B_n = 0$. Table 4.3 summarizes all the parameters of the generator, all the values used during the experiment are presented in that table as well. That table is also, extended with parameters from Table 3.3, which values are specific to the data generator.

Another important detail is the fact that in the case of this generator, the logistic regression model, that causes drift, is only retrained every 3 iterations. This means that the direction of the drift does not change every single time, but those changes are rather cyclical.

Parameter	Meaning	Value
μ_0	The stationary mean of the non-spam distribution	1.0
σ_0^2	The standard deviation of the non-spam distribution	0.25
μ_1	The mean of the spam distribution t_0	-1.0
σ_1^2	The standard deviation of the spam distribution	0.25
ϵ	The strength of performativity	10.0
n	number of features	11
B	performative array,, informs whether feature is prone to drift	-
no samples	number of samples generated in each iteration	20 000
no iters	number of testing iterations	20
k	Table 3.3	150
N	Table 3.3	50 epochs = 1950 steps

Table 4.3: Summary of the data-generation process parameters.

4.2.2 Izzo leaning curves

The goal of this section is similar to what has been presented in Section 4.1.2, showing how each of the components of the architecture was trained and how the training influences the results. Figure 4.17 provides snapshots of the learning curves, illustrating the progress of the training process. The first detail, that catches the eye's attention is that before the 150th step (value of k from Table 3.3) all of the curves behave differently than later. That is caused, by the switch in the update rules described in algorithm 1. To be exact, when the generator updates are switched on, the distance loss and the generator's real/fake loss suddenly drop, and the opposite happens to the discriminator's real/fake loss and the domain classification accuracy. The effect of changing the strategy of the updates is not visible on the loss functions that deal with domain classification. It might seem, that later the values of the loss functions fluctuate, which in a most standard supervised machine learning setup would not be desired. However, those oscillations have to do with the adversarial nature of the process, as the networks compete with each other if one of them gets better, it is expected that the performance of the other one will deteriorate. It has been observed that stopping the training process after 20 epochs instead of 50, would be detrimental for the results, which are presented in Section 4.2.3. What is also important to notice is that on the Figure 4.14 until the kth step is reached the domain accuracy stays at around 50%, the probability of pure guess in binary classification. That shows that the feature extractor until that time manages to fool the discriminator and create a representation that is not determinant when it comes to the domain, in which a point originates.

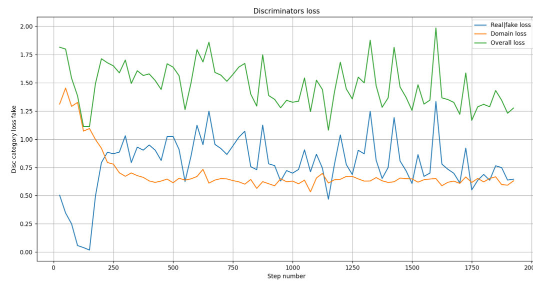

Figure 4.13: The loss of the discriminator over training.

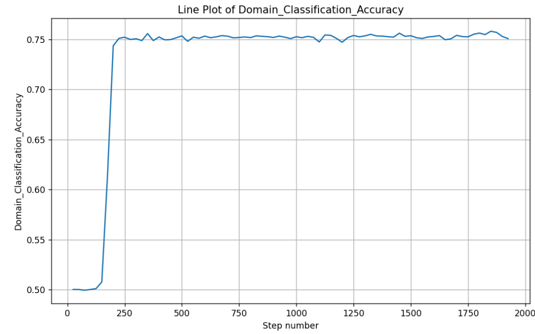

Figure 4.14: The domain accuracy over training

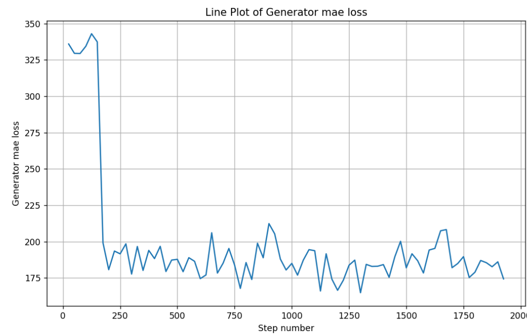

Figure 4.15: The distance loss of the generator over training

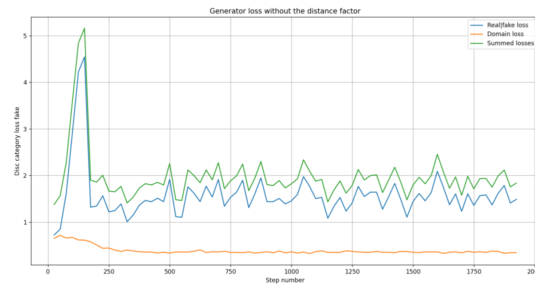

Figure 4.16: The adversarial loss of the generator over training

Figure 4.17: Learning curves of the subparts of the entire architecture

4.2.3 Izzo results

After the architecture was trained, by using the data from distributions t_0 and t_1, it was evaluated by generating 20 iterations of data. The results of that evaluation are presented below. Figure 4.18 illustrates the results of the Principal Component Analysis. In the case of this generator, every third iteration has been captured on the plot, as that is the time when the direction of the drift changes. Taking a closer look at those plots, it is possible to observe that in each distribution there are 2 clusters (red - original data t_0, blue - data influenced by the drift t_i, green - data generated by the model). Those clusters probably represent the two classes (spam and non-spam). Red clusters are stationary, and the blue ones are constantly being shifted. That occurs, because the model causing the drift is retrained. The shift by itself is not surprising, however the directions of the movement are. It seems that drift causes the clusters to oscillate. To show that effect, Figure 4.19 was created. From iteration 1 (blue) to 4 (green) the clusters are being pushed away from each other, but from 4 (green) to 7 (red) they are being squeezed together again. This phenomenon keeps reoccurring, as the iterations progress. The effect of that event might be that a distribution influenced a couple of times by the drift is more similar to the starting one than one that has only been influenced once. This fact might also have further implications, which will be discussed later.

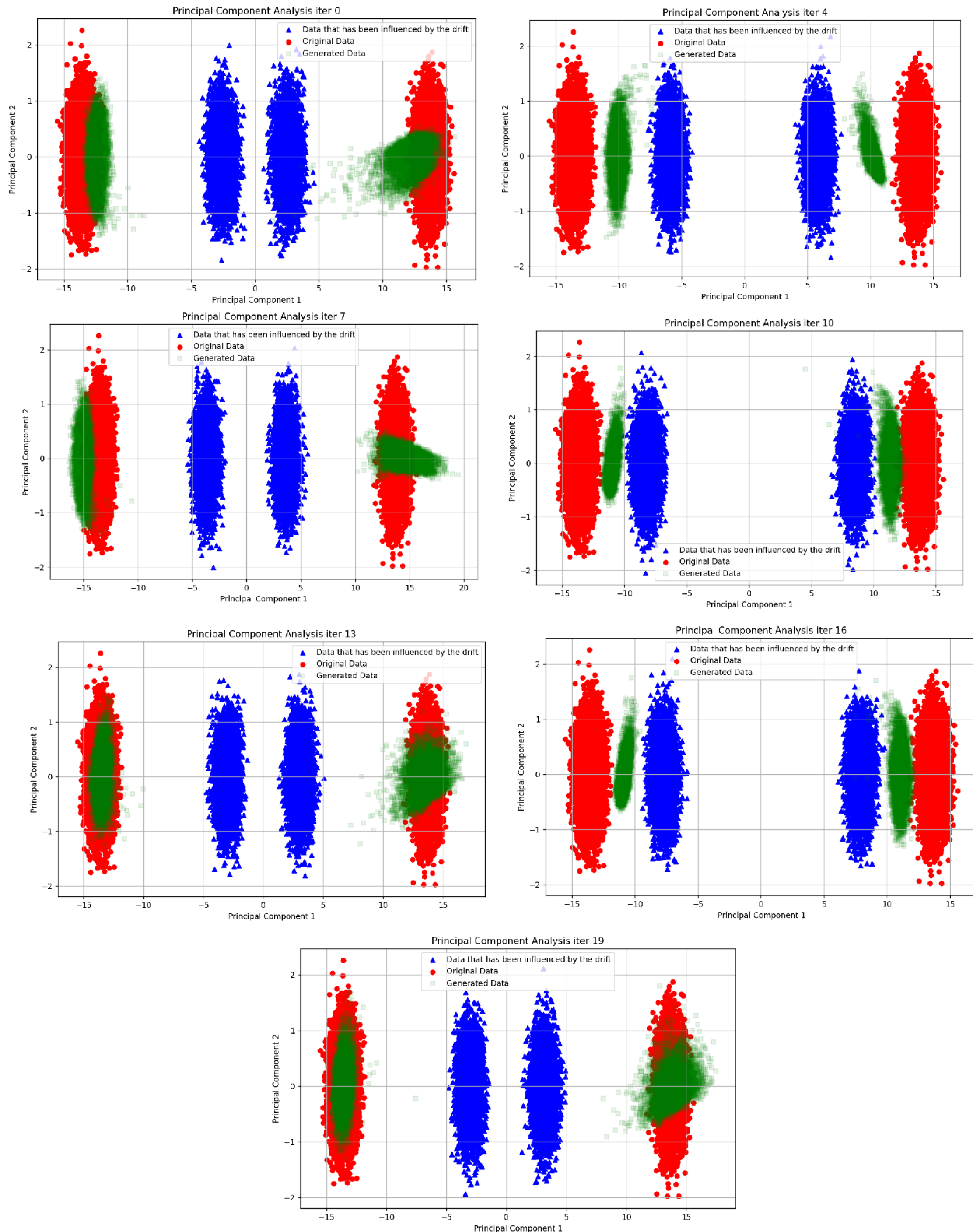

Figure 4.18: Results of Principal Component Analysis over the iterations.

From the analysis of the Figure 4.18 it can be inferred that the generator usually performs better at the task of reconstruction of the left cluster. The generated clusters placed at around -15 on the scale of Principal Component 1 resemble their corresponding red clusters in a much better way than their counterparts on the right. They are more elongated, meanwhile, the ones placed at around 15 are much more concentrated at one spot. This could mean that the clusters on the left represent

the non-spammers class, which is not influenced by the drift, it would explain why the generator is performing better at the reconstruction of those points.

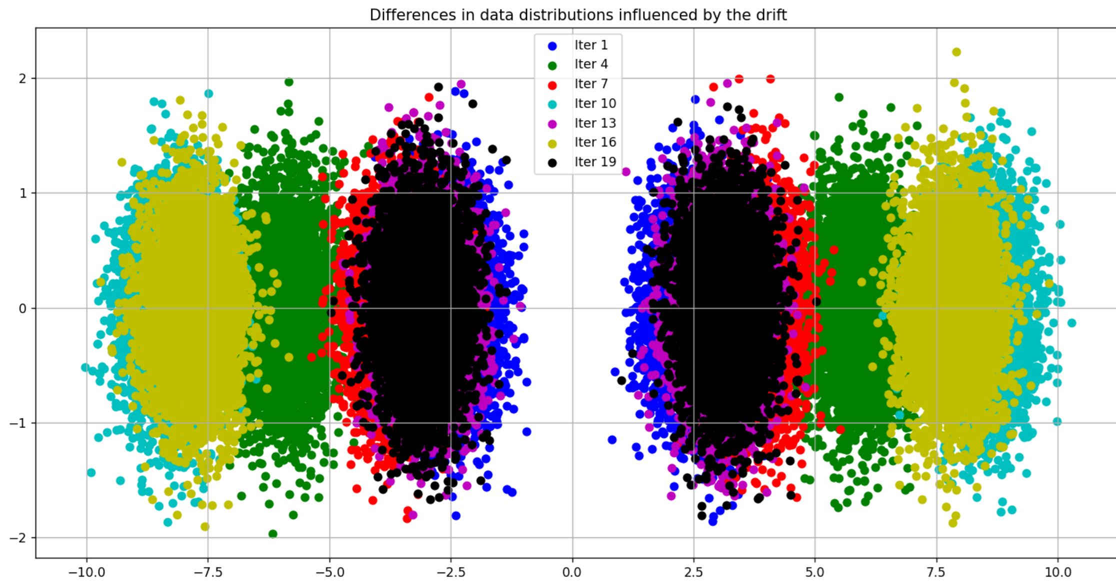

Figure 4.19: Visualization of the cyclical nature of the drift.

The Figure 4.20 pictures the correlation between two distances: red - $L_1(t_i, t_0)$ and purple $L_1(G(t_i), t_0)$ and accuracies of the accuracy of the original logistic regression model (blue) and accuracy of that same model but when the inputs are the generated points (green). The green line stays extremely close to 1 throughout the simulation, meaning that the created by the generator representation is perfectly interpretable by the logistic regression and allows for accurate classification. The blue line fluctuates to a significant degree, for the first nine iterations it stays close to 0.5, which shows that the performance of the model has deteriorated from the initial 100% of accuracy. What is odd, is the fact the cyclical nature of the drift pictured on Figure 4.19 causes the accuracy to spike for 3 iterations at $i = 10$ and $i = 16$. It can be observed that in the moments, when the accuracy of the original LR peaks, the distance $L_1(t_i, t_0)$ drops to its lowest values. An explanation for that would be that the cyclical drift caused the newly created distribution to be similar to the starting one. As for the distances, the distance between the generated distribution $L_1(G(t_i), t_0)$is always smaller than the $L_1(t_i, t_0)$. However, a correlation can be observed, when the red line achieves its largest values there occurs a noticeable spike on the purple curve too (it is especially visible at $i = 6, 7, 8$ but also when $i = 14, 19$). A reasoning behind that could be, that the model is only trained to understand one type of drift. If all of a sudden the direction of the movement changes completely, then the generator still can provide meaningful results, but that is mostly because it knows how the starting distribution looked like and not because it understands the direction of the movement.

Summing up, the results achieved in this experiment still show that it is possible to train a model, that will be able to map the points back to the starting distribution. It has not been shown above, but the accuracies achieved by the *Label Classifier* are the same as the green line on Figure 4.20, and the accuracy remains at 100%

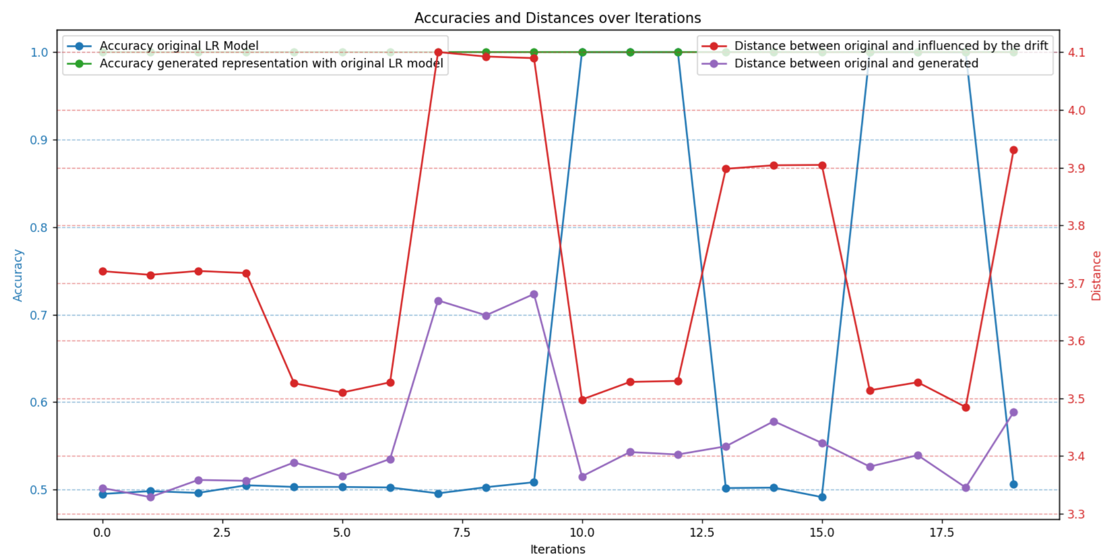

Figure 4.20: Correlation of distance between distributions and accuracies of the models.

for the entire duration of the simulation. So a stable, drift-resistant classifier is provided and the mapping created by the generator also provides insight into the direction of the movement. The results of the projection are sufficient for the original logistic regression model to classify all instances correctly. The generator succeeds at minimizing the distance, however, the generated distributions are still not identical as t_0.

5. Summary

An architecture, which could allow achieving both stable classification and learning about the direction of the movement, which has been caused by model drift, has been devised. Training of a model, which has been described in this thesis could help mitigate the problem of dealing with the model drift. It provides an alternative strategy to constant retraining of the deployed models. The architecture has been evaluated by designing a series of experiments, with the usage of performative data generators. The results show that with sufficient training time and model complexity, the proposed architecture could provide valid results. However, as in any research, there still is plenty of room for improvement.

5.1 Conclusions

This section aims to summarize the findings presented in Chapter 4 and to draw conclusions from the graphs, tables and descriptions included there. The synthesis of key results should help get a better understanding of the entire experiment.

To begin with, if the performance of the *generator* is taken into account. In all experiments that have been performed either the PCA analysis or the analysis of boxplot charts (that compared distributions in each iteration) have shown that the network struggles to introduce enough variance into the newly created representation. Either PCA clusters are not elongated enough. Or the outliers on the boxplot are omitted. This problem can be caused by multiple factors. Those might include, feeding the model not enough data during the training process, the training period lasting too short, the network architectures being not complex enough, and a suboptimal choice of hyperparameters.

Followingly, when it comes to providing stable, drift-resistant classification with the *Label classifer*. In both cases, when the experiment was based on Perdomo's data generator, a decay in the performance of that predictor has been observed. On the contrary, in the case of Izzo's data generator, the performance peaked throughout the simulation. The difference in those results could be caused by the nature of the data generation process. In the case where the results are better a cyclical drift

occurs, in the other experiment the data is modified based on the ith (t_i) and not the starting distribution (t_0. All of those facts lead to the conclusion, that definitely in certain cases it is possible to achieve stable classification. However, the stability depends on numerous factors such as the direction of the drift, its strength and the way it is imposed on the environment.

Subsequently, the *generator* attempts to learn a mapping from a distribution, which has been affected by the drift back to its original form. However, the problem with this phenomenon is that the generator is a neural network, which essentially is a black box. Meaning that it does not explain the drift in such a way that it is understandable to humans. Moreover, in the scenario presented in this thesis, where the architecture is trained on iterations t_0 and t_1, the generator should only learn one type of drift. Assuming a scenario, where the direction of the drift changes after the architecture is trained. Even if the results are provided by the model (what happened in the simulations where the LR model was retrained), it is because the model has seen the starting distribution during training and not because it understands the drift that has occurred.

Training of the architecture described in this thesis could be computationally very expensive. As generally, the generative networks provide better results, after being trained for a large number of epochs and if the model architectures are complex. Retraining a simple model could simply turn out to be less costly than devising a rather complicated architecture, tuning it to specific circumstances. As every experiment has shown, retraining still can be considered as the more straightforward solution, which still proves to be effective. If the costs are a large factor in a project, the architecture could be altered and the *generator* could be omitted (as the aim of the generator is to understand and be able to reverse the drift). This way, just the standard DANN is trained and as it has been shown after tweaking it to the right situation, it could act as a stable classifier.

Finally, getting back to the origins of the thesis. The transfer learning methods which have been applied to compute a new representation of the dataset did not provide valid results, that has been described in Section 3.1.3. That leads to a conclusion, that they should not be applied to understand or reverse the *performative* effects of a drift.

To summarize the conclusions, we have demonstrated several scenarios, where the GDAN architecture provides valid results. It can be treated as an alternative to retraining. It comes with its limitations. The architecture is complex and requires a larger computational cost than retraining. However, it can excel in a situation where retraining is not feasible. For instance, because new labels arrive with delay, or no domain expert would provide the ground truths. As for training of the architecture, only two iterations of data are needed. The findings of this thesis could also be simplified. Consider, a situation where no insight into the direction of the drift is needed, then deploying only the Domain Adversarial Part of GDAN could be a solution, providing stable classification, but less complicated than the full GDAN.

5.2 Research question answers

RQ1: A review of the generators available in the literature has been performed and explained in detail in Section 2.1.5. The generator, which has been most widely used is the one devised by [39]. A total of four generators have been described, alongside the way the interaction between the environment and the model is defined.
SRQ1.1: First of all, all the mechanisms have been described, most of them utilize the model parameters to either move the current distribution or change the base one. Furthermore, during the experiments, the specific direction of the drift induced on data by two generators was examined. The direction of the movement has been visualized either by a PCA analysis or a boxplot chart.

RQ2: A few transfer learning methods have been handpicked (the motivation behind that described in the literature review chapter) and applied. The methods, which have been selected include, ARC-t, JDA and Domain Adversarial Neural Networks.
SRQ2.1: Unfortunately, the mappings computed with the usage of those methods were not of any value. The results were presented in Section 3.1.3, but most methods did not surpass the pure guess benchmark.

RQ3: Training of a stable, drift-resistant classifier is possible, however, it might be dependent on the surrounding circumstances.
RQ3.1: An elaborate answer to this question is provided in the third paragraph of the Section 5.1.

RQ4: A method that should allow for both stable classification and mapping the distribution back to its original form has been devised. The proposed architecture combines the concepts of DANNs and GANs. It was described in Section 3.2 and later evaluated by performing a series of experiments.

5.3 Limitations

Apart from providing valid results, which allow for drawing plenty of conclusions, there exist limitations to the method and the experiments that have been performed. This section will elaborate on those limitations.

Synthetic data
First of all, as mentioned before collecting real-world datasets, which illustrate the performative effects is a gruelling task. The dataset needs not only to include data but also the model parameters, times of retraining and preferably the time when the shift has been detected. Due to those reasons, the availability of such datasets is very limited. That is why, for the sake of this thesis data generators have been used. Usage of them introduces limiting factors by itself, as something created synthetically can only attempt to mimic the real-world scenario, but will never fully resemble the reality.

Simplifed interactions between model and environment
Another limitation is connected with the previous one. The way the interaction

between the model and the environment has been designed in the simulation is simplified. Again the processes, which occur in everyday life can be highly non-linear, and hardly predictable. Usage of the generators can resemble them to some degree and definitely can show some correlations that happen. However, machine learning models are often exposed to interaction with humans, whose behaviour might not be logical or possible to describe with mathematical equations. Due to that, the modelled dynamics between the model and the environment will always be an approximation.

Limited computational power
Finally, a technical restriction occurred. The development of the architecture and especially the training of the models is a task, that requires plenty of computational power. All the training and evaluation have been performed on a local desktop. To show how that limits the possibilities, an example will be provided. The trained generator has 1,596,621 total parameters. That was the largest model, which could have been handled by the available resources. Any other model with more parameters would result in crashes during training. Also, due to the size of the model, when PCA analysis was performed points had to be subsampled to decrease the size of the dataset. Otherwise, the available system memory would be overloaded and the analysis would fail.

Full parameter search
The parameters of the training and setup of layers presented in earlier sections have been selected arbitrarily, by testing and evaluating a limited amount of configurations. The number of parameters allows for numerous combinations and most likely the chosen one is suboptimal. The solution to this issue would be performing a full parameter search, examining most of the possible scenarios.

5.4 Future Work

Due to a variety of reasons several research directions were left unexplored. There is plenty of space for broadening the scope of this thesis. The section will attempt to elaborate on a few of them.

Evaluation with real-world data
As mentioned before, in the limitations section, the lack of a performative dataset is considered a setback. If possible, evaluation of the model with the usage of data collected empirically would be beneficial and it would add even more scientific value to the area.

Introduction of more variety into the generator
As mentioned in the conclusions, there is quite a noticeable problem when it comes to generators replicating very diverse data points. One potential research direction would be to explore the possible network architectures. Meaning, changing the order of the layers, their size, and their hyperparameters. But most importantly, probably the biggest upgrade potential could be found in trying a variety of activation functions. Currently, the generator utilizes almost the simplest approach with the linear activation function. Designing a custom equation, that would help to deal with the problem of insufficient variety, could provide prominent results.

Smoothing the training procedure

Even though a few regularization techniques were applied during this research, the learning curves, especially in the case of the Izzo experiment, leave a lot to be desired. Attempting a different combination of those or using a different architecture could provide significantly better results. In the literature, there exists plenty of evidence on how the training procedure can be bettered. Just to name a few, Wasserstein GANs (WGAN), Boundary Equilibrium GANs (BEGAN), and dual discriminator GANs, but the options are almost endless. Implementation of those techniques could improve the performance of the devised architecture.

Usage of more complex models

Finally, as memory and computational power have been an issue throughout this research, training a more complex model with similar architecture on a powerful cluster, could also provide better results.

Explanation of the generator's behaviour

The previous chapters have shown that even though the generator is capable of providing insightful mapping, is still a black box. This means that the transformation it performs is just a result of the weights that have been trained. It does not provide an explanation of how the drift occurred or in which direction it acted, which is understandable for humans. Applying explainable AI(XAI) techniques to show how it truly works, could provide more transparency to the process.

Bibliography

[1] (2008). *Lagrange Multiplier*, pages 292–294. Springer New York, New York, NY. (cited on Page 17)

[2] Awais, M., Iqbal, M. T. B., and Bae, S. (2020). Revisiting internal covariate shift for batch normalization. *IEEE Transactions on Neural Networks and Learning Systems*, 32:5082–5092. (cited on Page 35)

[3] Aytar, Y. and Zisserman, A. (2011). Tabula rasa: Model transfer for object category detection. In *2011 International Conference on Computer Vision*, pages 2252–2259. (cited on Page 13)

[4] Badola, A., Nair, V. P., and Lal, R. (2020). An analysis of regularization methods in deep neural networks. *2020 IEEE 17th India Council International Conference (INDICON)*, pages 1–6. (cited on Page 37)

[5] Borgwardt, K. M., Gretton, A., Rasch, M. J., Kriegel, H.-P., Schölkopf, B., and Smola, A. J. (2006). Integrating structured biological data by kernel maximum mean discrepancy. *Bioinformatics*, 22(14):e49–e57. (cited on Page 16)

[6] Brown, G., Hod, S., and Kalemaj, I. (2022). Performative prediction in a stateful world. In *International Conference on Artificial Intelligence and Statistics*, pages 6045–6061. PMLR. (cited on Page 8)

[7] Chen, X., Duan, Y., Houthooft, R., Schulman, J., Sutskever, I., and Abbeel, P. (2016). Infogan: Interpretable representation learning by information maximizing generative adversarial nets. In Lee, D., Sugiyama, M., Luxburg, U., Guyon, I., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc. (cited on Page 21)

[8] Credit Fusion, W. C. (2011). Give me some credit. (cited on Page 8 and 26)

[9] Creswell, A., White, T., Dumoulin, V., Arulkumaran, K., Sengupta, B., and Bharath, A. A. (2018). Generative adversarial networks: An overview. *IEEE Signal Processing Magazine*, 35(1):53–65. (cited on Page 20)

[10] Dragomiretskiy, S. (2022). Influential ml: Towards detection of algorithmic influence drift through causal analysis. Available athttps://studenttheses.uu.nl/handle/20.500.12932/369. (cited on Page 5)

[11] Dumoulin, V. and Visin, F. (2018). A guide to convolution arithmetic for deep learning. (cited on Page 35)

[12] Fernando, B., Habrard, A., Sebban, M., and Tuytelaars, T. (2013). Unsupervised visual domain adaptation using subspace alignment. In *2013 IEEE International Conference on Computer Vision*, pages 2960–2967. (cited on Page 12)

[13] Gama, J., Žliobaitė, I., Bifet, A., Pechenizkiy, M., and Bouchachia, H. (2014). A survey on concept drift adaptation. *ACM Computing Surveys (CSUR)*, 46. (cited on Page 5 and 6)

[14] Ganin, Y., Ustinova, E., Ajakan, H., Germain, P., Larochelle, H., Laviolette, F., March, M., and Lempitsky, V. (2016). Domain-adversarial training of neural networks. *Journal of Machine Learning Research*, 17(59):1–35. (cited on Page ix, 18, and 19)

[15] Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial networks. (cited on Page 20, 21, and 23)

[16] Griffin, G., Holub, A., and Perona, P. (2022). Caltech 256. (cited on Page 10)

[17] Gui, J., Sun, Z., Wen, Y., Tao, D., and Ye, J. (2023). A review on generative adversarial networks: Algorithms, theory, and applications. *IEEE Transactions on Knowledge and Data Engineering*, 35(4):3313–3332. (cited on Page 21)

[18] Healy, K. (2015). The performativity of networks. *European Journal of Sociology*, 56(2):175–205. (cited on Page 6)

[19] Hoi, S., Sahoo, D., Lu, J., and Zhao, P. (2018). Online learning: A comprehensive survey. *Neurocomputing*, 459:249–289. (cited on Page 1)

[20] Hotelling, H. (1933). Analysis of a complex of statistical variables into principal components. *Journal of educational psychology*, 24(6):417. (cited on Page 26)

[21] Isola, P., Zhu, J.-Y., Zhou, T., and Efros, A. A. (2018). Image-to-image translation with conditional adversarial networks. (cited on Page ix, 23, and 30)

[22] Izzo, Z., Zou, J., and Ying, L. (2022). How to learn when data gradually reacts to your model. In *International Conference on Artificial Intelligence and Statistics*, pages 3998–4035. PMLR. (cited on Page 8, 9, 26, and 47)

[23] Jagadeesan, M., Zrnic, T., and Mendler-Dünner, C. (2022). Regret minimization with performative feedback. *CoRR*, abs/2202.00628. (cited on Page 8)

[24] Jiang, T. and Cheng, J. (2019). Target recognition based on cnn with leakyrelu and prelu activation functions. *2019 International Conference on Sensing, Diagnostics, Prognostics, and Control (SDPC)*, pages 718–722. (cited on Page 37)

[25] Kulis, B., Saenko, K., and Darrell, T. (2011). What you saw is not what you get: Domain adaptation using asymmetric kernel transforms. In *CVPR 2011*, pages 1785–1792. (cited on Page 12, 14, and 17)

[26] Long, M., Wang, J., Ding, G., Sun, J., and Yu, P. S. (2013). Transfer feature learning with joint distribution adaptation. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*. (cited on Page 11)

[27] Mendler-Dünner, C., Perdomo, J., Zrnic, T., and Hardt, M. (2020). Stochastic optimization for performative prediction. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H., editors, *Advances in Neural Information Processing Systems*, volume 33, pages 4929–4939. Curran Associates, Inc. (cited on Page 8 and 9)

[28] Miller, J., Hsu, C., Troutman, J., Perdomo, J., Zrnic, T., Liu, L., Sun, Y., Schmidt, L., and Hardt, M. (2020). Whynot. (cited on Page 8)

[29] Miller, J. P., Perdomo, J. C., and Zrnic, T. (2021). Outside the echo chamber: Optimizing the performative risk. In *International Conference on Machine Learning*, pages 7710–7720. PMLR. (cited on Page 8)

[30] Mirza, M. and Osindero, S. (2014). Conditional generative adversarial nets. (cited on Page 21)

[31] Motiian, S., Piccirilli, M., Adjeroh, D. A., and Doretto, G. (2017). Unified deep supervised domain adaptation and generalization. (cited on Page 11)

[32] Muandet, K., Balduzzi, D., and Schölkopf, B. (2013). Domain generalization via invariant feature representation. In Dasgupta, S. and McAllester, D., editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 10–18, Atlanta, Georgia, USA. PMLR. (cited on Page 13)

[33] Obla, S., Gong, X., Aloufi, A., Hu, P., and Takabi, D. (2020). Effective activation functions for homomorphic evaluation of deep neural networks. *IEEE Access*, 8:153098–153112. (cited on Page 35)

[34] Odena, A., Olah, C., and Shlens, J. (2017). Conditional image synthesis with auxiliary classifier GANs. In Precup, D. and Teh, Y. W., editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 2642–2651. PMLR. (cited on Page 22 and 30)

[35] Ojha, V., Abraham, A., and Snásel, V. (2017). Metaheuristic design of feedforward neural networks: A review of two decades of research. *Eng. Appl. Artif. Intell.*, 60:97–116. (cited on Page 35)

[36] Pan, S. J., Tsang, I. W., Kwok, J. T., and Yang, Q. (2011). Domain adaptation via transfer component analysis. *IEEE Transactions on Neural Networks*, 22(2):199–210. (cited on Page 11, 12, and 13)

[37] Pan, S. J. and Yang, Q. (2010). A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359. (cited on Page 10 and 11)

[38] Pathak, D., Krahenbuhl, P., Donahue, J., Darrell, T., and Efros, A. A. (2016). Context encoders: Feature learning by inpainting. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2536–2544. (cited on Page 24)

[39] Perdomo, J., Zrnic, T., Mendler-Dünner, C., and Hardt, M. (2020). Performative prediction. In III, H. D. and Singh, A., editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 7599–7609. PMLR. (cited on Page 4, 6, 7, 8, 9, 26, 40, and 55)

[40] Ronneberger, O., Fischer, P., and Brox, T. (2015). U-net: Convolutional networks for biomedical image segmentation. (cited on Page 36)

[41] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. (2015). ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252. (cited on Page 10)

[42] Saenko, K., Kulis, B., Fritz, M., and Darrell, T. (2010). Adapting visual category models to new domains. In Daniilidis, K., Maragos, P., and Paragios, N., editors, *Computer Vision – ECCV 2010*, pages 213–226, Berlin, Heidelberg. Springer Berlin Heidelberg. (cited on Page 12 and 14)

[43] Specht, D. (1990). Probabilistic neural networks. *Neural Networks*, 3:109–118. (cited on Page 36)

[44] Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15:1929–1958. (cited on Page 37)

[45] Von Stackelberg, H. (2010). *Market structure and equilibrium*. Springer Science & Business Media. (cited on Page 6)

[46] Wang, J. and Chen, Y. (2023). *Introduction to Transfer Learning: Algorithms and Practice*. Springer Nature. (cited on Page 11, 16, and 17)

[47] Wang, J., Chen, Y., Hu, L., Peng, X., and Yu, P. S. (2017). Stratified transfer learning for cross-domain activity recognition. (cited on Page 11)

[48] Y, G., Nair, N., Satpathy, P., and Christopher, J. (2019). Covariate shift: A review and analysis on classifiers. pages 1–6. (cited on Page 5)

[49] Zhao, H., Coston, A., Adel, T., and Gordon, G. J. (2020). Conditional learning of fair representations. (cited on Page 11)

[50] Zhou, B., Lapedriza, A., Xiao, J., Torralba, A., and Oliva, A. (2014). Learning deep features for scene recognition using places database. *Advances in neural information processing systems*, 27. (cited on Page 10)

[51] Zhu, J.-Y., Park, T., Isola, P., and Efros, A. A. (2017). Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*. (cited on Page 21)

[52] Zhu, Y., Zhuang, F., Wang, J., Ke, G., Chen, J., Bian, J., Xiong, H., and He, Q. (2021). Deep subdomain adaptation network for image classification. *IEEE Transactions on Neural Networks and Learning Systems*, 32(4):1713–1722. (cited on Page 11)

[53] Zhuang, F., Qi, Z., Duan, K., Xi, D., Zhu, Y., Zhu, H., Xiong, H., and He, Q. (2020). A comprehensive survey on transfer learning. (cited on Page 10)

[54] Zrnic, T., Mazumdar, E., Sastry, S., and Jordan, M. (2021). Who leads and who follows in strategic classification? In Ranzato, M., Beygelzimer, A., Dauphin, Y., Liang, P., and Vaughan, J. W., editors, *Advances in Neural Information Processing Systems*, volume 34, pages 15257–15269. Curran Associates, Inc. (cited on Page 6)