FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Generování tabulkových dat se zachováním závislostí
Student:	Jakub Renc
Vedoucí:	Ing. Miroslav Čepek, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Znalostní inženýrství
Katedra:	Katedra aplikované matematiky
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

Nemožnost sdílet data, ať už z důvodu ochrany osobních údajů nebo obchodního tajemství, omezuje mnoho výzkumných a vývojových aktivit. Cílem práce je prozkoumat možnosti generování tabulkových dat, která budou reflektovat závislosti a distribuce jako originální, nedostupná, data. V rámci projektu se zaměříte na generování numerických a kategorických dat. Ideálním výsledkem projektu je možnost přenést nalezené výsledky a modely přímo na originální data bez nutnosti nebo jen s minimálním transfer learning.

Elektronicky schválil/a Ing. Karel Klouda, Ph.D. dne 9. února 2022 v Praze.

Bakalářská práce

Generování tabulkových dat se zachováním závislostí

Jakub Renc

Katedra aplikované matematiky
Vedoucí práce: Ing. Miroslav Čepek, Ph.D.

8. května 2022

Poděkování

Tímto bych rád poděkoval mému vedoucímu Ing. Miroslavu Čepkovi, Ph.D., který se mě ujal a důkladně vedl po celý semestr. Bez jeho pomoci by práce neměla výslednou podobu. Dále bych chtěl poděkovat své rodině a přítelkyni za neuvěřitelnou podporu během psaní bakalářské práce a značnou pomoc při korektuře gramatických chyb. Ná závěr bych rád poděkoval Jakubu Rencovi za pevné nervy a vynaložené úsilí na této práci.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 8. května 2022 . . . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií

Odkaz na tuto práci

Renc, Jakub. *Generování tabulkových dat se zachováním závislostí.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2022.
Zdrojové kódy jsou dostupné online na adrese `https://github.com/JkbRnc/Data_Generators`

Abstrakt

Tato práce se zabývá problematikou generování tabulkových dat se zachováním závislostí. Cílem práce je vytvoření generátoru, který umožní syntézu dat se stejnými statistickými vlastnostmi jako mají data původní. Dále musí být model schopný generovat jak numerická, tak i kategorická data.

Práce se podrobněji věnuje implementaci dvou generativních modelů, a to variačnímu autoenkodéru a *Generative adversarial networks*. V experimentální části práce jsou tyto modely porovnávány s vybranými generátory z knihovny SYNTHETIC DATA VAULT. Výstupy z jednotlivých modelů jsou hodnoceny na základě úspěšnosti při klasifikačních a regresních úlohách. Přesto, že *Generative adversarial networks* opakovaně dosáhly nejlepších výsledků, nelze jednoznačně určit lepší model. Oba generátory prokázaly, že jsou schopné úspěšně syntetizovat tabulková data se zachováním potřebných vlastností.

Klíčová slova generování tabulkových dat, GANs, autoenkodéry, variační autoenkodéry, hluboké učení

Abstract

This thesis processes the problematics of tabular data generation while preserving dependencies. The main aim of the thesis is to create a generator, which can generate data with the same statistical properties as original data. In addition, the model must be able to generate both numerical and categorical data.

The present thesis contains a detailed implementation of two generative models, namely Variational autoencoder and Generative adversarial networks. The experimental part compares these models with chosen generators from the SYNTHETIC DATA VAULT library. Models outputs are evaluated based on results from classification and regression tasks. Although Generative adversarial networks repeatedly scored the best results, it is impossible to determine a better model. Both generators proved that they can generate tabular data while preserving desired features.

Keywords tabular data generation, GANs, autoencoders, variational autoencoders, deep learning

Obsah

Seznam obrázků

Úvod

Informace ovlivňují život lidstva už od nepaměti. Ti, kteří jsou schopni je správně využít, získávají výhodu nad ostatními. Proto jsou často informace a data dobře chráněna. V praxi dochází k restrikci volného šíření dat, a to například kvůli ochraně osobních údajů nebo ochraně obchodního tajemství. To vede k omezení možností analýzy různých problematik, výzkumu a vývoje. Za použití dnešních technologií jsme schopni nová data uměle generovat. Nelze však data pouze náhodně generovat, protože by nemusela odpovídat realitě. Proto je třeba vytvořit datový generátor, který bude generovat data se stejnými statistickými vlastnostmi, jako mají reálná data.

Téma jsem si vybral, kromě výše uvedených důvodů, protože se mnohdy setkáváme s generátory textu a obrázků, ale ne tak často s generátory numerických a kategorických dat. Zároveň se domnívám, že s přibývajícími omezeními na zpracování osobních údajů roste důležitost této problematiky.

Hlavním cílem mé práce je vytvoření datového generátoru. Tento generátor bude schopný na základě vstupní tabulky vytvořit nová data, která budou mít stejné statistické rozdělení, jako mají původní data. Oproti původním datům lze vygenerovaná data volně používat.

Cílem teoretické části práce je vytvoření přehledu technik pro generování dat. Následovat bude vyčlenění potenciálních řešení problematiky generování tabulkových dat a zformulování teoretického podkladu pro tvorbu modelů. Tomu se budu věnovat v prvních dvou kapitolách – **Techniky pro generování dat** a **Použité techniky**.

Praktická část má za cíl implementaci zvolených metod z teoretické části. Konkrétně se jedná o vytvoření modelů pro generování tabulkových dat takových, že vygenerovaná data budou reflektovat závislosti a distribuce původních dat. Implementaci modelů – variačních autoenkodérů a *Generative adversarial networks* – se budu věnovat ve 3. kapitole. Nakonec, ve 4. kapitole, vyhodnotím vlastnosti vygenerovaných dat a jejich úspěšnost při klasifikačních a regresních úlohách.

Kapitola 1

Techniky pro generování dat

V této kapitole se zaměřím na existující nástroje pro generování dat. Popíšu jednotlivé metody včetně jejich výhod a nevýhod. Jako příklady z praxe uvedu různé implementace v knihovnách či používaných softwarech. Samotné generátory budou rozděleny podle konceptů, na kterých se zakládají.

1.1 Distribuce pravděpodobnosti

Do této skupiny můžeme zařadit nejvíce datových generátorů. V nejjednodušších případech se jedná o náhodné generátory, které jsou založené na některé ze známých distribucí (Gaussova, Poissonova, Bernulliho) [3]. Jedním typem takových generátorů mohou být Gaussovské generátory náhodných čísel, jako například algoritmus Monty Python, GRAND nebo Wallacův algoritmus [4]. Tyto metody je možné kombinovat. Existují případy, kdy jednotlivé sloupce tabulkových dat byly generovány podle různých rozdělení. Takový jev se vyskytuje u generátoru Spaten [5], který slouží k syntéze geo-sociálních dat. K tomu využívá 3 různá rozdělení – rovnoměrné pro generování domova, Gaussovo pro ohodnocení a Bernulliho pro rozhodnutí, zda uživatel vyrazí na cesty, nebo ne.

Výše zmíněné generátory nelze vždy použít, protože původní data neodpovídají žádné ze známých distribucí. Z toho důvodu jednodušší generátory velmi často slouží jako vstup do složitějších. Jmenovitě se jedná například o iterativní algoritmus Ruscia a Kaczetowa [6] nebo o Fleishmanovu metodu [7].

Pro generování dat, která neodpovídají známým distribucím, se v praxi často využívají kovariační matice. Ty popisují vzájemné závislosti jednotlivých prvků náhodného vektoru. Pro náhodný vektor $\mathbf{X} = (X_1, X_2, \ldots, X_n)$ je

definována kovariační matice [8]:

$$\Sigma_X = \begin{bmatrix} \mathrm{cov}(X_1, X_1) & \mathrm{cov}(X_1, X_2) & \dots & \mathrm{cov}(X_1, X_n) \\ \mathrm{cov}(X_2, X_1) & \mathrm{cov}(X_2, X_2) & \dots & \mathrm{cov}(X_2, X_n) \\ & \dots & & \\ \mathrm{cov}(X_n, X_1) & \mathrm{cov}(X_n, X_2) & \dots & \mathrm{cov}(X_n, X_n) \end{bmatrix}$$

S využitím kovariačních matic pro syntézu dat jako první přišli Vale a Maurelli (VM) [9], kteří vytvořili generátor pro vícerozměrná data s nenormální distribucí. Algoritmus VM je kombinací polynomiální transformace a dekompozice kovariační matice, vymodelované na základě vstupních dat. Tento přístup je inspirován algoritmem Allena Fleishmana. S implementací se můžeme setkat v softwarových balíčcích Mplus nebo lavaan[1] [10]. Od publikace v roce 1983 došlo k mnoha modifikacím VM transformace. Jednou z nich byl výše zmíněný iterativní algoritmus Ruscia a Kaczetowa [6].

Zajímavou alternativou k přístupu VM je transformace nezávislým generátorem (IG), kterou navrhli Foldnes a Olsson [10]. Tento algoritmus se opírá o kovariační matici, která umožňuje syntézu dat s nenormální distribucí, ale přidává do výpočtu náhodný vektor vytvořený IG. Nová data se získají lineární kombinací nezávislého vektoru a dekompozicí kovariační matice [10]. Největší výhodou oproti VM transformaci je větší rozptyl dat. Nevýhodou transformace IG je její výpočetní náročnost.

1.2 Evoluční přístup

Evoluční přístup ke generování dat využívá *Genetic doping algorithm* (GenD). Jedná se, stejně jako u genetického algoritmu (GA), o optimalizační metodu, která se zakládá na evoluci limitované populace. Oproti GA si GenD udržuje vnitřní nestabilitu. Právě ta umožňuje postupný vývoj evoluce a pozvolný růst biodiverzity [11].

Na začátku algoritmu dojde k vytvoření několika neuronových sítí a to za pomoci různých způsobů učení a architektur sítí. Nově vytvořené sítě jsou opakovaně trénovány na různých částech trénovacího datasetu a následně testovány. Cílem je získat nejlepší možné sítě pro každou z kombinací architektur a způsobů učení. Každá z nich se poté využije ke zhodnocení výsledku GenD [12].

Díky různorodosti výsledků získaných pomocí GenD nám vznikne dataset, ve kterém mají nezávislé proměnné téměř nulovou korelaci. To ale může vést ke vzniku dat, která nemohou v reálném světě existovat (například těhotný muž nebo hledání prvního zaměstnání v 80 letech) [12], což je značnou nevýhodou pro využití v praxi.

[1] Programy pro statistické modelování.

1.3 Aproximace funkcí

Funkční aproximace je známým problémem z aplikované matematiky. Cílem aproximace je najít výpočetně nenáročnou funkci, která nejblíže odpovídá svými hodnotami funkci původní. V procesu učení to znamená hledání funkce, která nejlépe popisuje vztah mezi vstupními a výstupními daty [13].

Na této myšlence jsou postaveny *Radial basis function* (RBF) sítě. RBF sítě původně sloužily pouze jako nástroj pro aproximaci [13]. Dnes jsou součástí standardních metod pro učení neuronových sítí a nacházejí se v různých balíčcích a knihovnách (např. *Stuttgart neural network simulator*) [14]. RBF sítě obvykle mají 3 vrstvy – vstupní, 1 skrytou a výstupní. Právě aktivační funkce skryté vrstvy jsou různé RBF, podle kterých se sítě nazývají [14, 15]. Generování dat je zajištěno generováním nezávislých proměnných jako vstupních parametrů pro síť [14].

1.4 Machine learning

Ve strojovém učení se pro generování dat nejčastěji využívají modely postavené na hlubokém učení. Mezi nejúspěšnější modely hlubokého učení patří variační autoenkodéry (VAE) a *Generative adversarial networks* (GANs). S oběma modely se nejčastěji setkáme při generování obrázků, ale mají své využití i pro syntézu tabulkových dat.

GANs vznikly roku 2014 jako nový způsob využití hlubokého učení pro generování dat [16]. Výchozí myšlenkou je striktně kompetitivní hra dvou hráčů. Mějme tedy hráče generátor (G) a distraktor (D). G soupeří spolu s D do té doby, než se dostanou do Nashova ekvilibria [17]. G se tedy snaží vytvořit nová data co nejblíže odpovídající distribuci původních dat, a D má za cíl rozeznat vygenerovaná data od dat původních.

U VAE opět dochází k využití dvou modelů jako u GANs. Zatímco GANs vycházejí z teorie her, myšlenka VAE vznikla v souvislosti s redukcí dimenzionality[2]. Právě jednotlivé modely VAE – enkodér a dekodér – slouží jako nástroje na redukci dimenzionality. Cílem enkodéru je převést původní data do reprezentace v nižší dimenzi s co nejmenší ztrátou informací. Dekodér by pak měl převést tuto reprezentaci zpět do původní podoby [19]. Oproti původním autoenkodérům přidávají VAE tzv. latentní prostor, který je regularizovaný. Tato vlastnost umožňuje generování dat dle normální distribuce v latentním prostoru a následné dekódování za účelem tvorby dat [20].

Rozhodl jsem se pro podrobnější analýzu GANs i VAE, a to kvůli jejich dosavadním výsledkům v oblasti generování syntetických dat. Z toho důvodu

[2] Prokletí dimenzionality je fenomén, kdy s každou přidanou dimenzí obtížnost učení roste exponenciálně [18]. K řešení této problematiky se využívají například autoenkodéry nebo *Principal component analysis*.

se budu v dalších kapitolách podrobně věnovat struktuře a teoretickému podkladu obou modelů i následně jejich implementaci.

Dalším, speciálním, případem jsou generátory pro přirozený jazyk (*Natural language generators*, NLG). Výstupy z NLG mají podobu jak textovou (slovo, věta), tak grafickou (graf, diagram, tabulka) [21]. Jedním takovým typem jsou statistické *n*-gram modely. *N*-gram je sekvence znaků o délce *n*, která je odvozena z textu (typicky slovo o délce alespoň *n* znaků) [22, 23]. Při tvorbě modelu nejprve dojde k výpočtu pravděpodobnosti výskytu bigramů a trigramů na základě dat z *Wall Street Journal* [24]. Poté dojde k úpravě dat za použití *enhanced Good-Turing method* [25]. Tato metoda nám kromě úpravy stávajících pravděpodobností přiřadí nenulovou pravděpodobnost pro *n*-gramy, které se v trénovacích datech nenacházely (a to na základě *m*-gramů; m < n) [24]. S pomocí takto vytvořených pravděpodobností lze generovat jednotlivá slova, a tedy i věty. Příkladem implementace *n*-gramového modelu je *IBM Air Travel Reports* [26].

Kapitola 2

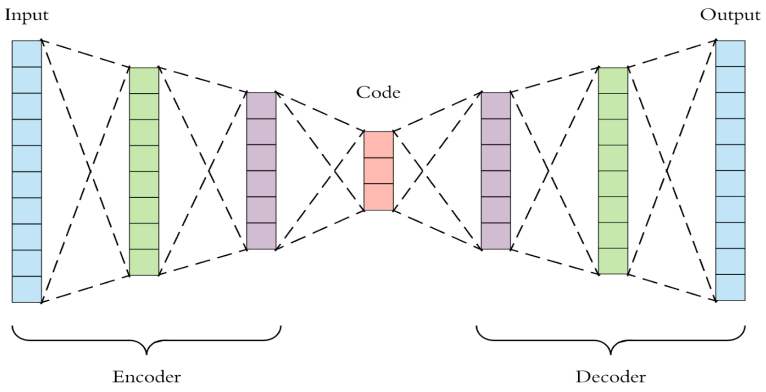
Vybrané techniky

Ve 2. kapitole se budu věnovat teoretickému základu zvolených technik, a to variačnímu autoenkodéru (VAE) a *Generative adversarial networks* (GANs). Nejprve představím základní strukturu autoenkodéru, na které staví VAE, a poté popíšu optimalizační problém VAE včetně hlavní myšlenky používané účelové funkce. Nakonec se zaměřím na využití VAE pro syntézu tabulkových dat. V druhé části kapitoly se budu věnovat GANs. Opět představím matematický základ a poté se budu věnovat jednotlivým modifikacím.

2.1 Variační autoenkodéry

Variační autoenkodér (VAE) je umělá neuronová síť, která je schopná zakódovat vektor do reprezentace v jiné normalizované dimenzi a následně jej dekódovat. Jako první VAE představili Diederik P. Kingma a Max Welling v roce 2014 [27]. Jedná se o modifikaci klasického autoenkodéru.

Hlavním cílem autoenkodéru je redukce dimenzionality, tedy nalezení reprezentace původního vektoru v nižší dimenzi, tzv. latentním prostoru. Dů-



Obrázek 2.1: Architektura autoenkodéru [1]

ležité je, aby se během procesu zakódování a dekódování ztratilo co nejméně informací o původním vektoru. Pro tyto účely se využívají dvě hlavní části autoenkodéru – enkodér a dekodér (viz obrázek 2.1). Enkodér i dekodér jsou plně propojené dopředné neuronové sítě. Autoenkodéry se učí za použití *back-propagation* a lze je použít pouze na redukci dat podobných těm, na kterých se model naučil [1].

Při využití autoenkodérů pro syntézu dat se ukáže, že po dekódování mnoho kódů z latentního prostoru neodpovídá smysluplnému vektoru. To je dáno tím, že u autoenkodérů latentní prostor není regularizován. Proto VAE zavádějí regularizaci během trénování modelu.

Regularizace je dána distribucemi původního a latentního prostoru. Nechť x je prvkem náhodné veličiny s rozložením $p^*(x)$, které není známé. Poté můžeme aproximovat zvolený model $p_\theta(x)$ parametrem θ:

$$x \sim p_\theta(x)$$

Za předpokladu, že z je prvkem latentního prostoru a $p(z)$ jeho distribucí, bude podmíněná distribuce $p(z|x)$ podle Bayesovy věty definována vztahem:

$$p_\theta(z|x) = \frac{p_\theta(x|z)p(z)}{p_\theta(x)} = \frac{p_\theta(x,z)}{p_\theta(x)}$$

Tento vztah lze dopočítat ze známých modelů. Výše zmíněné výpočty mají vysokou časovou náročnost [28], a proto se u VAE využívá aproximace inferenčním modelem $q_\phi(z|x)$. Inferenční model odpovídá enkodéru u výše zmíněných standardních autoenkodérů (obrázek 2.1). Cílem je optimalizace variačního parametru ϕ tak, aby platilo:

$$q_\phi(z|x) \approx p_\theta(z|x)$$

Inferenční model lze vyjádřit jako orientovaný graf:

$$q_\phi(z|x) = q_\phi(z_1, z_2, ...z_M|x) = \prod_{j=1}^{M} q_\phi(z_j|Pa(z_j), x)$$

kde $Pa(z_j)$ je množina rodičů z_j v orientovaném grafu. Dále lze inferenční model vyjádřit za použití neuronových sítí [28], a to jako:

$$(\mu, \log \sigma) = \text{EncoderNeuralNet}_\phi(x)$$

$$q_\phi(z|x) = \mathcal{N}(z, \mu, \text{diag}(\sigma))$$

K optimalizaci modelu se využívá tzv. *evidence lower bound* metoda (ELBO), která je dána vztahem:

$$\begin{aligned}\log p_\theta(x) &= \mathop{\mathbb{E}}_{q_\phi(z|x)}\left[\log p_\theta(x)\right] \\ &= \mathop{\mathbb{E}}_{q_\phi(z|x)}\left[\log \frac{p_\theta(x,z)}{p_\theta(z|x)}\right] \\ &= \mathop{\mathbb{E}}_{q_\phi(z|x)}\left[\log \frac{p_\theta(x,z)q_\phi(z|x)}{p_\theta(z|x)q_\phi(z|x)}\right] \\ &= \mathop{\mathbb{E}}_{q_\phi(z|x)}\left[\log \frac{p_\theta(x,z)}{q_\phi(z|x)}\right] + \mathop{\mathbb{E}}_{q_\phi(z|x)}\left[\log \frac{q_\phi(z|x)}{p_\theta(z|x)}\right]\end{aligned}$$

kde prvním sčítancem je výše zmíněná ELBO a druhým Kullback-Leiber (KL) divergence [28]. KL divergence udává poměr mezi dvěma distribucemi pravděpodobností [29]. Z definice musí být KL divergence nezáporná a bude rovna nule právě tehdy, když platí:

$$q_\phi(z|x) = p_\theta(z|x)$$

Čím je tedy inferenční model podobnější distribuci $p_\theta(z|x)$, tím se bude blížit hodnota ELBO logaritmické distribuci modelu $p_\theta(x)$ a nabývat nejvyšších možných hodnot.

Pro VAE to znamená výpočet a maximalizaci ELBO během trénování modelu. Toho docílí tzv. reparametrizační trik [27], který je dán následným vztahem. Nechť je dána náhodná veličina $\tilde{z} \sim q_\phi(z|x)$, reparametrizace je možná za použití diferencovatelné transformace $g_\phi(\epsilon, x)$ a náhodné veličiny[3] ϵ:

$$\tilde{z} = g_\phi(\epsilon, x); \epsilon \sim p(\epsilon)$$

Tato reparametrizace spolu s Monte Carlo odhadem vede k výpočtu gradientu, na základě kterého dochází k optimalizaci ELBO [27]. Pro implementaci to znamená, že výstupem enkodéru nebude latentní vektor, jako u autoenkodérů, ale střední hodnota a rozptyl pro každou latentní veličinu, na základě kterých se sestrojí latentní prostor.

Dále je důležité zadefinovat účelovou funkci pro VAE. Ta je složena ze dvou částí – rekonstrukční účelové funkce a podobnostní účelové funkce [30]. Rekonstrukční účelová funkce se liší podle použitých dat. Jako podobnostní funkce se v praxi využívá KL divergence popsaná vztahem:

$$\mathrm{KL}(\mathrm{G}(\mu_\theta(X), \sigma_\theta(X)), \mathrm{X}(0,1)) = \frac{1}{2}(\mu_\theta(X)^2 + \sigma_\theta(X) - \log(\sigma_\theta^2(X)) - 1)$$

kde μ_θ je střední hodnotou inferenčního modelu a σ_θ^2 je jeho rozptylem.

[3] V různých zdrojích je také nazývána šumem.

2.1.1 TVAE

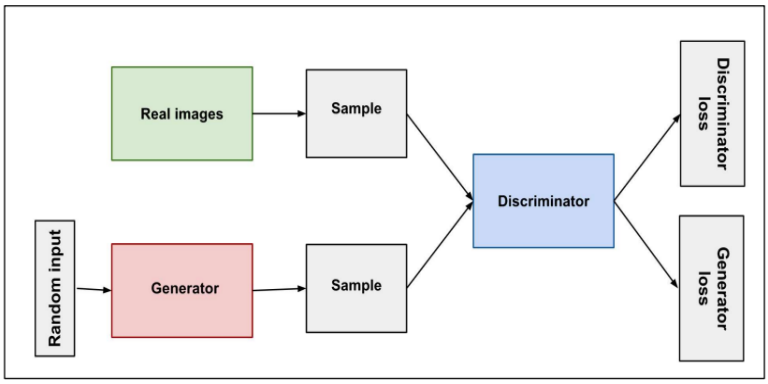
Důležitou modifikací VAE je tzv. *Tabular VAE* (TVAE), se kterým přišli Xu a spol [31]. Výstupem enkodéru $p_\theta(r_j|z_j)$ je společné rozdělení pravděpodobnosti $2N_c + N_d$ veličin odpovídajících $2N_c + N_d$ veličiny r_j, kde N_c a N_d odpovídají spojitým (N_c) či diskrétním (N_d) sloupcům [31]. Dále autoři použili jako aktivační funkci pro spojité veličiny hyperbolický tangens a pro diskrétní funkci softmax. Zbytek TVAE odpovídá původnímu návrhu VAE. Testování proběhlo společně s modelem CTGAN, kterému se bude věnovat část následující sekce.

Implementace TVAE je součástí knihovny synthetic data vault (sdv).

2.2 Generative adversarial networks

Zatímco Diederik Kingma a Max Welling pracovali na VAE, Ian Goodfellow, Jean Pouget-Abadie a spol pracovali na novém typu hlubokého generativního modelu tzv. *Generative adversarial networks* (GANs) [16]. Jejich cílem bylo vytvoření generativního modelu, který nebude využívat mnoho aproximací a výpočtů pravděpodobnostního gradientu. Takový model by měl být schopný generovat data s cílenou distribucí za nižší výpočetní náročnosti [16].

GANs se skládají ze dvou modelů – generátor a diskriminátor. Cílem diskriminátoru je rozeznat původní data od vygenerovaných, zatímco se generátor snaží vygenerovat nová data co nejpodobnější původním. Tento proces znázorňuje obrázek 2.2[4]. Aby byl generátor schopný generovat data odpovídající reálným datům, musí se naučit distribuci p_g na datech x.



Obrázek 2.2: Architektura GANs [2]

Nechť z jsou náhodné veličiny latentního prostoru a p_z jejich distribuce. Poté lze značit prostor dat jako $G(z; \Theta_g)$, kde G je diferencovatelná funkce reprezentovaná vícevrstvým perceptronem s parametry Θ_g. Dále Goodfellow

[4] *Real Images* reprezentuje reálná data.

a spol [16] definují druhý vícevrstvý perceptron $D(x; \Theta_d)$. Výstupem tohoto perceptronu je hodnota, která udává, zda se jedná o vygenerovaná data (tedy distribuce p_g), nebo o původní data. Pro zhodnocení trénovacího procesu je dále potřeba cenová funkce $V(G, D)$. Celý proces učení je tedy dán vztahem:

$$\min_D \max_G V(G, D)$$

Cílem učení je maximalizace rozeznání vygenerovaných od původních dat perceptronem D a zároveň trénink G tak, aby došlo k minimalizaci $\log(1 - D(G(z)))$. Tedy:

$$\min_D \max_G V(D, G) = \mathop{\mathbb{E}}_{x \sim p_{data(x)}} [\log D(x)] + \mathop{\mathbb{E}}_{z \sim p_z(z)} [\log(1 - D(G(z)))]$$

Dále Goodfellow a spol [16] dokázali, že pro fixní generátor existuje unikátní optimální diskriminátor D^* daný vztahem:

$$D^*(x) = \frac{p_{data}(x)}{p_{data}(x) + p_g(x)}$$

Za předpokladu generátoru, pro který platí $p_g(x) = p_{data}(x)$, získáme úspěšnost diskriminátoru 50%. Jinými slovy, diskriminátor není schopen rozeznat původní data od vygenervaných, a je tedy dosažena maximální konfůze [32]. Takový generátor vznikne při dosažení Nashova equilibria během trénování modelu.

I přes důkazy existence optimálních generátorů je trénování GANs náročné a mnohdy bez úprav trénovacího cyklu nemožné [16, 33, 34]. To může být způsobeno například příliš rychlou konvergencí diskriminátoru k nule, což vede k neschopnosti generátoru aktualizat gradient [34].

2.2.1 DCGAN

S první významnou modifikací – *Deep Convolutional GANs* (DCGAN) – přišli Alec Radford a Luke Metz [35]. Oproti původnímu modelu nejsou skryté vrstvy DCGAN plně propojené. Dále dochází k tzv. *Batch Normalization* – normalizaci vstupu do každé jednotky tak, aby měl vstup střední hodnotu 0 a rozptyl roven 1 [35]. Nakonec model využívá aktivační funkce LeakyReLU[5] pro všechny vrstvy diskriminátoru a pro generátor využívá aktivační funkce ReLU[6] pro všechny vrstvy kromě výstupní, kde využívá tanh.

Radford a Metz trénovali model na 3 obrázkových datasetech (LSUN, IMAGENET-1K a FACES DATASET). Pro testování zvolili klasifikaci CIFAR-10 s využitím DCGAN pro rozpoznávání vzorů. Podle experimentů [35] měl DCGAN větší úspěšnost než všechny přístupy založené na metodě K-means, ale nižší než *Exemplar CNN* [36]. To, že DCGAN nebyl nikdy trénovaný na CIFAR-10, a přesto tato data úspěšně klasifikoval, ukazuje robustnost modelu.

[5] $\phi(x, a) = \max(a, x)$
[6] $\phi(x) = \max(0, x)$

2.2.2 WGAN

Další zajímavou modifikací je Wasserstein GAN (WGAN), který navrhli Arjovsky a spol [37]. Cílem WGAN je opět stabilizace učení modelu a eliminace jeho kolapsu. Toho Arjovsky a spol docílili změnou výpočtu vzdálenosti při učení modelu. Místo Jensen-Shannonovy nebo Kullback-Leiberovy vzdálenosti autoři použili Wasserstein-1[7] vzdálenost. Ta je podle autorů [37] dána vztahem:

$$W(P_r, P_g) = \inf_{\gamma \in \Pi(P_r, P_g)} \mathbb{E}_{(x,y)\sim\gamma}[||x - y||]$$

kde $\Pi(P_r, P_g)$ je společným rozdělením $\gamma(x, y)$, jehož marginálními pravděpodobnostmi jsou postupně P_r a P_g. Rozdělení $\gamma(x, y)$ tedy říká, jaké kroky je třeba provést, aby se z distribuce P_r stala distribuce P_g. Poté bude $W(P_r, P_g)$ udávat cenu této transformace [37].

Za předpokladu, že vygenerovaný prostor je spojitý, bude poté i spojitá funkce výpočtu Wassersteinovy vzdálenosti. Zároveň bude tato funkce téměř vždy diferencovatelná. To stejné ale nelze tvrdit o Jensen-Shannonově nebo Kullback-Leiberově vzdálenosti [37], které byly použity v původním návrhu GANs. Použití Wassersteinovy vzdálenosti při optimalizaci má za důsledek spolehlivější učení diskriminátoru. Z testů, které provedli Arjovsky a spol, vyplývá, že WGAN je robustnější než standardní GAN a na rozdíl od DCGAN není třeba provádět *Batch Normalization*, což vede ke zrychlení učení modelu. Důležité je také zdůraznit, že při žádném z testů nedošlo ke kolapsu WGAN [37].

2.2.3 CTGAN

V roce 2019 Xu a spol [31] představili *Conditional Tabular GANs* (CTGAN). Jedná se o první modifikaci GANs, která se zaměřuje na generování tabulkových dat. Protože tabulková data mohou nabývat diskrétních i spojitých hodnot, bylo potřeba vytvořit flexibilnější model. Z toho důvodu, Xu a spol přišli se dvěma přístupy zpracování různých typů dat. Pro spojité veličiny vytvořili tzv. *mode-specific normalization*, aby odstranili problém nenormální distribuce [31]. Tato normalizace probíhá ve 3 krocích. Nejprve se pro každý sloupec C_i vytvoří smíšené normální rozdělení a vypočítá střední hodnota pro každou část směsi. Poté dojde k výpočtu pravděpodobnosti z jednotlivých částí směsi pro každou proměnnou $c_{i,j}$ z C_i. Jednotlivé hustoty pravděpodobnosti jsou pak dány vztahem:

$$\rho_k = \mu_k \mathcal{N}(c_{i,j}; \eta_k \phi_k)$$

kde η_k, μ_k a ρ_k jsou popořadě k-tá střední hodnota, váha její části směsi a příslušný rozptyl [31]. Nakonec dojde k normalizaci na základě jedné z distribucí.

[7] Také nazývána Earth-Moverova vzdálenost.

Dalším problémem je generování diskrétních veličin. To není možné pro GANs podle původního návrhu [16]. CTGAN tuto problematiku řeší za použití podmíněného vektoru[8], křížové entropie jako ztrátové funkce a trénování na bázi vzorkování[9].

Testování CTGAN probíhalo na simulovaných datech i na reálných datasetech. Pro porovnání Xu a spol využili Bayesovské sítě (PrivBN, CLBN), *Tabular VAE* (TVAE) a další modifikace GANs (MedGAN, VeeGAN, TableGAN) [31]. Simulovaná data Xu a spol vytvořili za použití orákul postavených na 4 známých Bayesovských sítích – *alarm*, *child*, *asia* a *insurance*. Použití orákul umožnilo jednodušší výpočet pravděpodobnostní *fitness* funkce pro ohodnocení dat. Jako reálné datasety použili autoři dataset CREDIT z Kaggle, ADULT, CENSUS, COVERTYPE, INTRUSION a NEWS z *UCI machine learning repository* a binarizovaný MNIST v podobě 28×28 a 12×12 pixelů. Vyhodnocení probíhalo na základě úspěšnosti modelu v klasifikačních a regresních úlohách.

Na simulovaných datech měl CTGAN nižší úspěšnost než Bayesovské sítě, ale dosáhl lepších výsledků než ostatní generativní modely. Na reálných datasetech se CTGAN i TVAE předvedly jako lepší řešení než Bayesovské sítě i ostatní modifikace GANs. Přestože TVAE dosáhl v mnoha případech lepších výsledků než CTGAN, nic nenaznačuje, že by se jednalo o lepší model, který by se měl použít v každém případě [31].

Implementace CTGAN je dostupná v knihovně SDV stejně jako implementace TVAE.

[8] Zřetězení kategorií zakódovaných použitím *one-hot encoding*.
[9] *Training-by-sampling*.

Kapitola 3

Implementace

V předchozí kapitole jsem představil teoretický základ pro variační autoenkodéry (VAE) a *Generative adversarial networks* (GANs). Zde se budu věnovat implementaci obou modelů. Pro implementaci jsem zvolil jazyk Python, protože má mnoho dostupných knihoven pro strojové učení a tvorbu neuronových sítí, například SCIKIT-LEARN (strojové učení), PYTORCH nebo TENSORFLOW (tvorba neuronových sítí). Dalším důvodem, proč jsem zvolil Python je podpora projektem Jupyter, který zjednodušuje tvorbu a testování modelů strojového učení.

Pro implementaci obou modelů jsem zvolil knihovnu PYTORCH. Tato knihovna obsahuje, mimo jiné, moduly NN a OPTIM. Modul NN slouží jako základní blok pro tvorbu neuronových sítí a grafů a v modulu OPTIM jsou implementované jednotlivé optimalizační algoritmy pro neuronové sítě (např. ADAM nebo SGD).

Implementace VAE, GANs, Jupyter notebooků i uživatelských skriptů jsou dostupné v přílohách, na přiloženém přenosném médiu a online na adrese `https://github.com/JkbRnc/Data_Generators`.

3.1 Variační autoenkodér

Základem VAE (kód B.1) je enkodér a dekodér. Z toho důvodu jsem nejprve pro oba modely implementoval jednoduché třídy, které dědí vlastnosti z NN.MODULE. Pro enkodér to znamenalo implementovat metody:

- ___INIT___(*input_dim*, *latent_dim*, *hidden_params*)
- FORWARD(x)

V konstruktoru ___INIT___ dochází k inicializaci neuronové sítě, kde velikost vstupní vrstvy odpovídá parametru *input_dim*, velikost skrytých vrstev odpovídá jednotlivým prvkům listu *hidden_params* a velikost výstupní vrstvy

odpovídá *latent_dim*. Dále konstruktor vytváří 2 další vrstvy o velikosti *latent_dim*, které slouží k výpočtu střední hodnoty a rozptylu. Metoda FORWARD pak vrací střední hodnotu a rozptyl výstupu neuronové sítě při vstupním parametru x. Tato metoda je definována v NN.MODULE a pro funkčnost modelu je nutné ji přepsat.

Třída pro dekodér opět obsahuje konstruktor a metodu FORWARD. Konstruktor vytvoří neuronovou síť podle vstupních parametrů, zatímco metoda FORWARD dekóduje vstup z latentního prostoru. Struktura obou sítí je plně propojená a využívá aktivační funkce ReLU. Navíc po každé vrstvě dojde k náhodnému vynulování 30% vstupních parametrů za použití NN.DROPOUT, což snižuje pravděpodobnost přeučení modelu.

Dále jsem implementoval třídu pro samotný VAE. Ta obsahuje, kromě pomocných metod, následující metody:

- ___INIT___(*latent_dim*, *e_params*, *d_params*, *num_epochs*, *batch_size*, *lr*)
  - *latent_dim* – dimenze latentního prostoru, automaticky nastavena na hodnotu 32
  - *e_params* – list obsahující počet uzlů pro jednotlivé skryté vrstvy enkodéru, automaticky nastaven na list [256, 256]
  - *d_params* – list obsahující počet uzlů pro jednotlivé skryté vrstvy dekodéru, automaticky nastaven na list [256, 256]
  - *num_epochs* – počet epoch při učení modelu, pokud nezměněn, je parametr nastaven na hodnotu 300
  - *batch_size* – velikost jednotlivých dávek dat, na kterých se model učí, automaticky nastaven na hodnotu 512
  - *lr* – parametr ovlivňující rychlost učení modelu, pokud neupraven, nastaven na 10^{-3}
- FIT(*data*, *categorical_columns*)
- SAMPLE(*quantity*)

Konstruktor si ukládá předané parametry a dále vytváří instance transformátorů pro kategorická a numerická data. Cílem těchto transformátorů je normalizace a úprava dat před zpracováním neuronovou sítí. Numerická data jsem normalizoval za použití STANDARDSCALER z knihovny SKLEARN. STANDARDSCALER upraví data tak, aby odpovídala normálnímu rozdělení. Pro kategorická data jsem použil ONEHOTENCODER. Ten z kategorických dat vytvoří tzv. kód 1 z n.

Následně jsem implementoval metodu FIT. Vstupními parametry jsou data, na kterých se model bude učit a list kategorických sloupců. Na základě parametru *categorical_columns* se vstupní data rozdělí na numerická a kategorická. Numerická data jsou poté transformována za použití STANDARDSCALER

a kategorická s pomocí ONEHOTENCODER. Poté dojde k inicializaci enkodéru a dekodéru, po které následuje učení obou modelů. K tomu slouží pomocná metoda ___TRAIN popsaná v algoritmu 1. Pro optimalizaci VAE jsem využil optimalizační algoritmus ADAM z knihovny OPTIM. Aby byl model schopný kvalitní rekonstrukce, která správně pokrývá rozdělení původních dat, zvolil jsem jako účelovou funkci součet *mean squared error* a Kullback-Leiber divergenci (viz algoritmus 2).

Algoritmus 1 Trénování VAE

for all Epochs **do**
  for batch in data **do**
    $mean, \log(variance) \leftarrow$ enkodér(batch)
    $z \leftarrow$ reparametrizace (standardizace) podle $mean, \log(variance)$
    $reconstruction \leftarrow$ dekodér(z)
    výpočet odchylky *reconstruction* od původních dat (viz algoritmus 2)
    přepočítání vah na základě odchylky v obou modelech
  end for
end for

Algoritmus 2 Účelová funkce

Require: $original, reconstruction, mean, \log(variance)$
$MSE = \mathbb{E}[(reconstruction - original)^2]$
$KLD = -0,5 * (1 + \log(variance) - mean^2 - e^{\log(variance)})$
return $MSE + KLD$

Nakonec jsem definoval metodu SAMPLE s parametrem *quantity*, který udává počet vygenerovaných řádků dat. Nejprve dojde ke generování náhodných dat v latentní dimenzi podle normálního rozdělení. Tato data následně dekodér transformuje do normalizované podoby. Nakonec dojde k inverzní transformaci za použití STANDARDSCALER pro numerická data, nebo ONEHOTENCODER pro kategorická data.

Pro využití modelu stačí nejprve inicializovat instanci třídy VAE a případně upravit parametry modelu. Poté stačí zavolat metodu FIT s dvěma parametry, a to zvolená data a list názvů kategorických sloupců. Nová data lze vygenerovat za použití metody SAMPLE, která vrací vygenerovaná data v podobě NUMPY ARRAY.

3.2 GANs

GANs (viz kód B.2) jsou opět tvořeny dvěma modely, a to diskriminátorem a generátorem. V obou případech jednotlivé modely dědí vlastnosti z třídy NN.MODULE z knihovny PYTORCH. Jak pro diskriminátor, tak pro generátor jsem implementoval kromě konstruktoru i povinnou metodu FORWARD.

Konstruktor diskriminátoru s parametry *input_dim* a *hidden_layers_params* vytvoří neuronovou síť se vstupní vrstvou o velikosti *input_dim* a výstupní vrstvou s 1 umělým neuronem. Velikosti jednotlivých skrytých vrstev odpovídají postupně jednotlivým prvkům listu *hidden_layers_params*. Jako aktivační funkci jednotlivých skrytých vrstev jsem zvolil LeakyReLU s parametrem 0,2. Dále mezi jednotlivými vrstvami dojde k aplikaci metody DROPOUT, která náhodně vynuluje různé elementy vektoru. To sníží pravděpodobnost, že během učení dojde k přeučení diskriminátoru. Metoda FORWARD vrací výstup neuronové sítě, na který je aplikována sigmoida. Vlastnosti sigmoidy zaručují, že výstup bude mezi hodnotami -1 a 1 a umožní mi později využít binární křížovou entropii jako účelovou funkci.

Na rozdíl od konstruktoru diskriminátoru má konstruktor generátoru navíc parametr *output_dim*, který určuje velikost výstupní vrstvy. Metoda generátoru FORWARD pouze vrací výstup neuronové sítě, přičemž na výstup není aplikována žádná další funkce.

Následně jsem implementoval třídu pro GANs. Ta obsahuje tyto metody:

- ___INIT___(*latent_dim*, *d_params*, *g_params*, *num_epochs*, *batch_size*, *loss_function*, *lr*, *k*)
  - *latent_dim* – velikost latentní dimenze, automaticky nastavena na hodnotu 64
  - *d_params* – list, ve kterém jeho prvky určují velikost jednotlivých skrytých vrstev diskriminátoru, automaticky nastavený na [256, 256]
  - *g_params* – list, ve kterém jeho prvky určují velikost jednotlivých skrytých vrstev generátoru, automaticky nastavený na [256, 256]
  - *num_epochs* – parametr určující počet epoch při učení modelu, automaticky nastaven na hodnotu 300
  - *batch_size* – velikost jednotlivých dávek dat při učení modelu, automaticky nastaven na hodnotu 512
  - *loss_function* – účelová funkce modelu, automaticky nastavena na binární křížovou entropii
  - *lr* – parametr určující rychlost učení modelu, automaticky nastaven na hodnotu 0,002
  - *k* – parametr určující počet iterací při učení diskriminátoru na 1 dávce dat, automaticky nastaven na hodnotu 5

- FIT(*data*, *categorical_columns*)
- SAMPLE(*quantity*)

Konstruktor pouze uloží předané parametry a vytvoří instance transformátorů pro kategorická a spojitá data. Stejně jako u VAE jsem využil modely z knihovny SKLEARN, a to STANDARDSCALER pro spojitá data a ONEHOTENCODER pro kategorická data.

Metoda FIT nejprve rozdělí data na kategorická a číselná podle parametru *categorical_data*. Poté postupně dojde k úpravě dat za použití jednotlivých transformátorů a k inicializaci diskriminátoru a generátoru. Parametry obou modelů jsou dány vlastnostmi dat. Následuje zavolání pomocné metody ___TRAIN, ve které proběhne samotné učení GANs, které je popsáno v algoritmu 3.

Algoritmus 3 Trénování GANs

Require: $k, epochs, data$
  for all *epochs* **do**
    for *batch* v datech **do**
      for all k **do**
        $x_d \leftarrow$ náhodně vygenerovaná data podle normálního rozdělení
        $y_d \leftarrow$ generátor(x_d)
        $z_d \leftarrow y_d + batch$
        diskriminátor(z_d)
        vyhodnocení úspěšnosti diskriminátoru účelovou funkcí
        přepočítání vah diskriminátoru na základě výstupu účelové funkce
      end for
      $x_g \leftarrow$ náhodně vygenerovaná data podle normálního rozdělení
      $y_g \leftarrow$ generátor(x_g)
      $z_g \leftarrow$ diskriminátor(y_g)
      vyhodnocení kvality dat y_g podle z_g účelovou funkcí
      přepočítání vah generátoru na základě výstupu účelové funkce
    end for
  end for

Učení modelu lze rozdělit na dvě fáze. V první fázi dojde k trénování diskriminátoru. Nejprve dojde k vygenerování náhodných dat podle normálního rozdělení. Z těchto dat následně generátor vytvoří data podobná původnímu datasetu. Vygenerovaná data spolu s původními daty tvoří vstup do diskriminátoru, který má za úkol rozeznat originální data od syntetických dat. Následuje vyhodnocení úspěšnosti diskriminátoru za použití účelové funkce, a poté úprava vah modelu. Tato fáze se opakuje podle parametru k uloženého v konstruktoru. Opakované učení diskriminátoru vede k jeho stabilizaci pro daný generátor. Druhá fáze obsahuje trénink generátoru, který probíhá obdobně jako

trénování diskriminátoru. Nejprve se vygenerují náhodné vektory s normálním rozdělením, ze kterých generátor vytvoří syntetická data (y_g, viz algoritmus 3). Tato data jsou předána diskriminátoru, který je ohodnotí (z_g, v algoritmu 3). Nakonec dojde k vyhodnocení kvality dat y_g podle z_g zvolenou účelovou funkcí. Obě fáze jsou opakovány pro jednotlivé dávky dat a jednotlivé epochy.

Protože diskriminátor provádí binární klasifikaci na původních i vygenerovaných datech, zvolil jsem jako účelovou funkci binární křížovou entropii (funkce BCELOSS z knihovny PYTORCH). Pro optimalizaci generátoru i diskriminátoru jsem vybral optimalizační algoritmus ADAM pro jeho rychlou konvergenci.

Nakonec jsem implementoval metodu SAMPLE. Tato metoda má jeden parametr – *quantity* – určující množství vygenerovaných dat. Nejprve dojde k vygenerování náhodných vektorů podle normální distribuce. Ty slouží jako vstup pro generátor, který z nich vytvoří výsledná normalizovaná data. Následuje rozdělení podle typu dat a transformace příslušnými transformátory. Metoda nakonec vrátí výsledná data v podobě NUMPY ARRAY.

Pro vygenerování dat za použití GANs je třeba nejprve inicializovat instanci třídy GAN. Následné volání metody FIT naučí model na zvolených datech. Jako druhý parametr je třeba také uvést list názvů kategorických sloupců. Pokud uživatel tento parametr nespecifikuje, přepokládá se, že žádný takový sloupec neexistuje. K vygenerování nových dat slouží metoda SAMPLE. GANs lze dále modifikovat úpravou předávaných parametrů konstruktoru.

3.3 Implementace experimentů a využití

Pro jednodušší využití obou modelů jsem vytvořil skript pro generování dat. Pro jeho použití je třeba nejprve nainstalovat všechny potřebné knihovny (setup.py - kód B.3) a mít nainstalovaný python 3.10.4 nebo novější. Poté stačí spustit skript s příslušnými argumenty:

```
python generate_script.py [DATA_PATH] [MODEL_PATH]
[SAMPLE_SIZE] [CATEGORICAL_COLUMNS] [MODEL_TYPE]
```

První dva argumenty příkazu jsou povinné a udávají postupně cestu ke vstupním datům a cestu k uloženému modelu. Cesta ke vstupním datům může být v podobě jak absolutní cesty, tak v podobě url. Je potřeba, aby tato data byla ve formátu csv a oddělená čárkou. Parametr [MODEL_PATH] udává cestu k již existujícímu modelu, nebo název souboru, do kterého se má model po naučení uložit. Je tedy možné opakovaně využít 1 model při více procesech generování dat. Zbývající argumenty udávají postupně počet vygenerovaných řádků dat, tento argument je automaticky nastavený na hodnotu 100, list kategorických sloupců[10], předběžně nastavený na prázdný list, a typ modelu[11].

[10] Názvy je třeba zadávat oddělené čárkou a bez mezer (například X1,X2,X3).
[11] Možnosti argumentu jsou VAE nebo GANs.

Pokud není typ modelu specifikován, je automaticky nastaven na VAE.

Spuštění generujícího skriptu (kód B.4) nejprve zjistí, zda model existuje, nebo zda je potřeba jej vytvořit. Pokud existuje, tak ho ze specifikovaného souboru načte. Pokud model neexistuje, inicializuje podle argumentu [MODEL_TYPE] instanci VAE, případně GANs. Následuje učení modelu na vstupních datech. Jakmile je model natrénovaný nebo načtený, následuje generování příslušného počtu řádků dat. Tato data budou následně uložena do souboru *generated_samples.csv*. Vygenerovaná data mají formát csv a jednotlivé položky jsou odděleny čárkou.

Pro zjednodušení testování jsem implementoval několik funkcí (kód B.5). První z nich je funkce *distance*, která vypočítá L2 vzdálenost (Euklidovská vzdálenost) na 1 prvek[12] mezi dvěma maticemi. Dále jsem vytvořil funkce *evaluate_randomforest* a *evaluate_knn*. Obě funkce mají parametry, kterými se jim předávají originální data, vygenerovaná data, cílený sloupec a informace, zda mají provést regresi, či klasifikaci. Funkce *evaluate_randomforest* a *evaluate_knn* jsou popsány v algoritmu 4.

Algoritmus 4 Vyhodnocovací algoritmus

Require: $classification$
  rozdělení originálních dat na trénovací a testovací množiny
  if $classification$ == True **then**
    vytvoření příslušného klasifikátoru A
    vytvoření příslušného klasifikátoru B
    naučení klasifikátoru A na vygenerovaných datech
    naučení klasifikátoru B na originálních trénovacích datech
    $x \leftarrow$ přesnost klasifikátoru A při predikci původních dat
    $y \leftarrow$ přesnost klasifikátoru B při predikci testovacích dat
    return (x, y)
  end if
  vytvoření příslušného regresoru A
  vytvoření příslušného regresoru B
  naučení regresoru A na vygenerovaných datech
  naučení regresoru B na originálních trénovacích datech
  $x \leftarrow$ MSE při predikci původních dat regresorem A
  $y \leftarrow$ MSE při predikci testovacích dat regresorem B
  return (x, y)

Tyto funkce vrací 2 hodnoty, a to úspěšnost/střední kvadratickou odchylku (MSE) při naučení predikčního modelu na syntetických datech a testování na originálních datech a úspěšnost/MSE při naučení modelu na části původních dat a testování na zbytku původních dat. První hodnota nám pomáhá určit kvalitu vygenerovaných dat, zatímco druhá nám ukazuje možnosti datasetu.

[12] $m_e = \frac{1}{m * n} \sqrt{\sum_{i=1}^{n} \sum_{j=1}^{m} (a_{i,j} - b_{i,j})^2}$

Nakonec jsem implementoval funkci *create_statistics*. Tato funkce naučí předaný model na originálních datech. Následně vygeneruje dané množství dat a vyhodnotí jejich úspěšnost za použití funkcí *evaluate_randomforest* a *evaluate_knn*. Tento proces generování a měření úspěšnosti se opakuje podle předaného parametru k, a tím vytvoří statistiku úspěšnosti modelu. Účelem funkce *create_statistics* je zjednodušení pozorování úspěšnosti modelu a eliminace extrémů.

Kapitola 4

Experimenty

V této kapitole otestuji kvalitu vygenerovaných dat. Pro testování jsem zvolil následující datasety:

- Real estate valuation dataset[13] [38] (dále jen real estate dataset),
- dry bean dataset[14] [39],
- breast cancer coimbra dataset[15] [40] (dále jen breast cancer dataset),
- prediction of average localization error in wsns dataset[16] [41](dále jen wsns dataset).

V prvních fázích experimentů jsem pouze hodnotil kvalitu dat z pohledu závislosti jednotlivých sloupců. K tomu jsem využíval korelační matice a L2 (Euklidovu) vzdálenost mezi nimi[17]. Následně jsem vytvořil jednoduché klasifikátory a regresory za použití algoritmů *random forest* a k-nejbližších sousedů (KNN). Tyto modely jsem poté naučil na vygenerovaných datech a měřil jejich úspěšnost v dané úloze na původních datech. Výsledky mých variačních autoenkodérů (VAE) a *Generative adversarial networks* (GANs) jsem porovnal s výsledky dvou modelů z knihovny synthetic data vault, a to s *Conditional tabular generative adversarial networks* (CTGAN) a *Tabular variational autoencoder* (TVAE).

[13] Dostupné ke dni 7.5.2022 z `https://archive.ics.uci.edu/ml/datasets/Real+estate+valuation+data+set`.

[14] Dosupné ke dni 7.5.2022 z `https://archive.ics.uci.edu/ml/datasets/Dry+Bean+Dataset`.

[15] Dostupné ke dni 7.5.2022 z `https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Coimbra`.

[16] Dostupné ke dni 7.5.2022 z `https://www.kaggle.com/datasets/abhilashdata/prediction-of-average-localization-error-in-wsns`.

[17] $m_e = \sqrt{\sum_{i=1}^{n} (a_i - b_i)^2}$

4.1 Real estate dataset

real estate dataset obsahuje informace týkající se faktorů ovlivňujících cenu nemovitostí v Nové Tchaj-pej, Taiwan. Dataset má celkem 8 sloupců:

- No – pořadí záznamu
- X1 – datum transakce v podobě desetinného čísla (např. 2013,500 značí červen 2013)
- X2 – věk nemovitosti v počtu let
- X3 – vzdálenost od nejbližší zastávky hromadné dopravy v metrech
- X4 – počet samoobslužných prodejen v dochozí vzdálenosti
- X5 – zeměpisná šířka
- X6 – zeměpisná délka
- Y – cena domu v přepočtu na 1 Ping (1 Ping = 3,3 m^2)

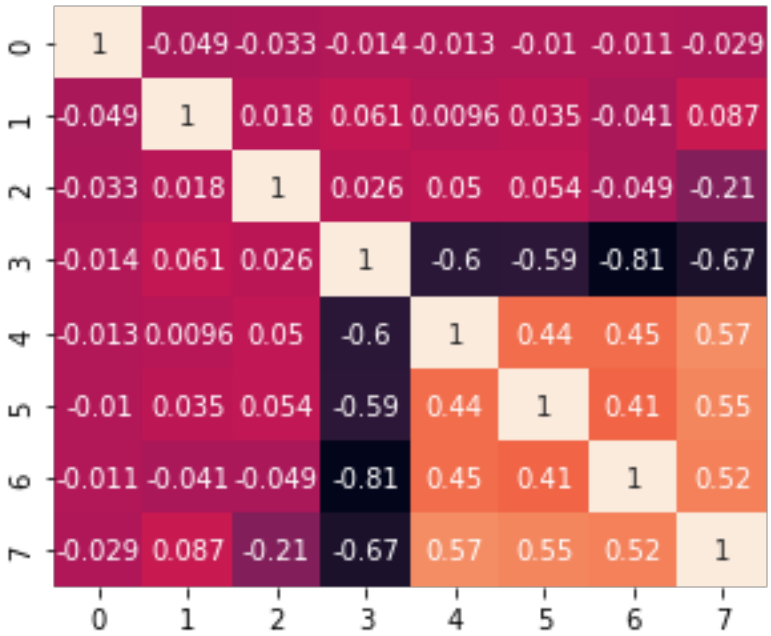
Nejprve jsem vytvořil instance tříd VAE a GANs. Oba modely jsem následně naučil na tomto datasetu a vygeneroval 10 000 řádků nových dat. Následně jsem vygeneroval z jednotlivých dat korelační matice[18]. Na obrázku 4.1 se nachází korelační matice původního datasetu, zatímco na obrázcích 4.3 a 4.2 lze vidět postupně korelační matice dat vygenerovaných VAE a GANs.

Následně jsem vyhodnocoval kvalitu dat podle průměrné L2 vzdálenosti korelačních matic na 1 prvek. V tabulce 4.1 tuto metriku značí poslední sloupec. Z této tabulky je možné vidět, že GANs spolu s modelem TVAE dosáhly velmi podobných výsledků. Zárovň korelační matice vygenerovaných dat velmi připomínají korelační matice dat původních. Vedle toho VAE a CTGAN dosáhly o něco horších výsledků. Přesto se stále jedná pouze o nevýznamnou nesrovnalost. Podobnost korelačních matic dat vygenerovaných GANs a původních dat je zřejmá i z pohledu na obrázky 4.1 a 4.2. Při porovnání obrázků 4.1 a 4.3 lze vidět výše zmíněné statistické rozdíly ve vlastnostech dat.

Tabulka 4.1: Výsledky testů – real estate dataset

Model	*random forest*	KNN	L2 vzdálenost korelačních matic
VAE	80,54	86,51	0,0170
GANs	61,07	83,62	0,0034
TVAE	79,97	93,57	0,0073
CTGAN	86,17	130,25	0,0208
Originální data	57,34	85,39	0

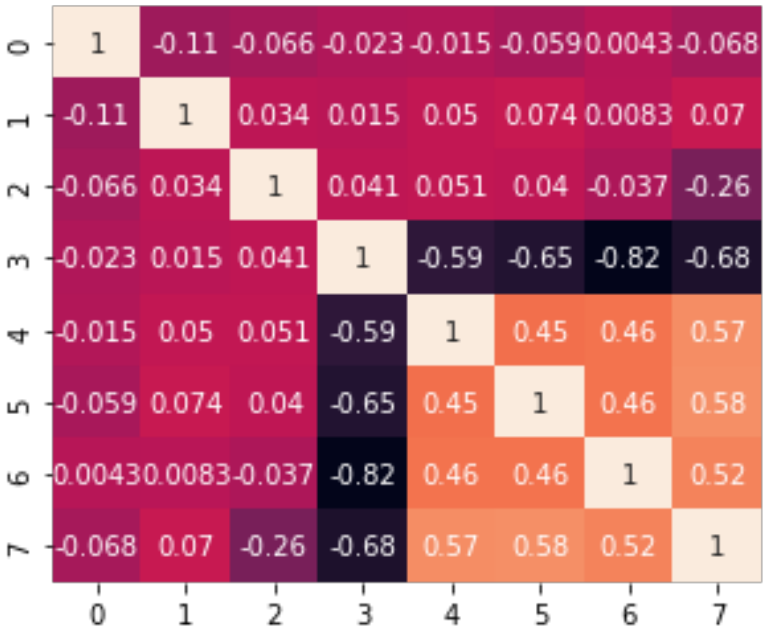
[18] Pro grafické zobrazení jsem použil knihovnu seaborn a funkci *heatmap*.



Obrázek 4.1: Korelační matice originálního REAL ESTATE DATASETU

Dalším způsobem, jakým jsem měřil kvalitu dat byla úspěšnost při regresi. Nejprve jsem vygeneroval data za použití jednotlivých generátorů. Následně jsem vytvořil regresory *random forest* a KNN, které jsem naučil na vygenerovaných datech a změřil jejich úspěšnost při regresi původních dat. Úspěšnost jsem měřil pomocí střední kvadratické chyby (MSE). Tento proces jsem opakoval stokrát a jako výslednou hodnotu jsem zvolil střední hodnotu této statistiky (funkce *create_statistics*). Pro porovnání jsem naučil jednotlivé regresory na dvou třetinách původních dat a na zbylé třetině je otestoval. Takto jsem opět vytvořil novou statistiku a zapsal do tabulky 4.1 její střední hodnotu.

Nejúspěšnějším modelem tohoto testu je jednoznačne GANs. Při použití *Random forest* se chyba dat vygenerovaných GANs liší minimálně od chyby predikce na originálním datasetu. U KNN regresoru vygenerovaná data GANs dokonce dosahují mírně lepších výsledků než originální data. To je způsobeno velikostí původního datasetu. Protože původní dataset má pouze okolo 400 řádků, během predikce se projeví každá extrémní hodnota v trénovací množině dat. Toto se u dat vygenerovaných GANs nestává, protože diskriminátor s vyšší pravděpodobností přijme standardní hodnoty za pravé, a proto je generátor bude generovat ve větším množství. To má za důsledek přesnější predikci u standardních hodnot.



Obrázek 4.2: Korelační matice vygenerovaného real estate datasetu GANs

Obrázek 4.3: Korelační matice vygenerovaného real estate datasetu VAE

4.2 Dry bean dataset

DRY BEAN DATASET je dataset o více než 13 000 záznamech, který obsahuje informace o 7 druzích suchých fazolí [39]. Celkem obsahuje 16 sloupců, ze kterých 15 popisuje fyzické vlastnosti fazolí (plocha, obvod, ...) a 1 značí druh fazolí.

Při testování vygenerovaných dat jsem postupoval obdobně jako u REAL ESTATE DATASETU. Nejprve jsem vytvořil korelační matice, které jsem následně porovnával. Jak z hodnot L2 metriky v tabulce 4.2, tak z příslušných obrázků (obrázky C.1, C.2, C.3 v přílohách, které postupně odpovídají korelačním maticím originálních dat, dat vygenerovaných GANs a dat vygenerovaných VAE) je zřejmé, že oba generátory jsou schopné generovat data s velmi podobnými závislostmi. Tento závěr potvrzují i výsledky druhého testu.

Druhým testem je klasifikační úloha, která strukturálně odpovídá regresní úloze u předešlého datasetu. Vytvořil jsem opět klasifikátory *random forest* a KNN, které jsem naučil na vygenerovaných datech. Jejich kvalitu jsem následně měřil na základě úspěšnosti predikce původních dat. Klasifikátory naučené na datech vygenerovaných VAE dosáhly nejvyšší úspěšnosti jak za použití klasifikátoru *random forest*, tak klasifikátoru KNN (viz tabulka 4.2). V obou případech se úspěšnost při učení na vygenerovaných datech lišila od úspěšnosti při učení na originálních datech do 3%. Klasifikátor *random forest* dokonce dosáhl úspěšnosti 90%. Úspěšnost dat vygenerovaných GANs zaostávala jak za daty vygenerovanými VAE, tak za daty syntetizovanými modely z knihovny SDV. Kromě toho se zde projevila další nevýhoda GANs. Protože DRY BEAN DATASET obsahuje mnoho řádků dat a velké množství sloupců (a další vytvořené kódováním 1 z n), trénování GANs zabralo výrazně větší množství času, než trénování VAE. Tento problém šlo pozorovat i u modelu CTGAN.

Tabulka 4.2: Výsledky testů – DRY BEAN DATASET

Model	*random forest*	KNN	L2 vzdálenost korelačních matic
VAE	90,69%	69,3%	0,0058
GANs	75,44%	49,22%	0,0079
TVAE	89,44%	53,02%	0,0070
CTGAN	90,02%	55,67%	0,0060
Originální data	92,27%	72,18%	0

4.3 Breast cancer dataset

BREAST CANCER DATASET je dataset vytvořený pozorováním 64 pacientů a 52 zdravých kontrolních jedinců v Coimbře v Portugalsku. Obsahuje 10 kvantitativních znaků (například věk nebo BMI) a 1 binární znak indikující, zda se

jednalo o pacienta nebo o kontrolního jedince. BREAST CANCER DATASET je dostupný z článku [40].

Protože se jedná o dataset pro klasifikaci, provedl jsem obdobné testy jako u DRY BEAN DATASETU. Při porovnávání vzdálenosti korelačních matic mé modely značně předčily jak TVAE, tak CTGAN (viz tabulka 4.3). V klasifikační úloze za použití *random forest* data vygenerovaná oběma modely přesáhla hranici 80% a předčila tak i klasifikátor naučený na původních datech. To je pravděpodobně způsobeno malým počtem záznamů, což zvyšuje váhu extrémních hodnot při učení modelu.

Tabulka 4.3: Výsledky testů – BREAST CANCER DATASET

Model	*random forest*	KNN	L2 vzdálenost korelačních matic
VAE	80,42%	73,4%	0,007
GANs	81,57%	75,29%	0,0085
TVAE	76,89%	68,63%	0,0159
CTGAN	56,24%	52,33%	0,0252
Originální data	71,1%	51,52%	0

4.4 WSNs dataset

Posledním datasetem, na kterém jsem prováděl experimenty je WSNS DATASET. Jedná se o dataset obsahující informace o lokalizační chybě bezdrátových senzorových sítí. Pro jednotlivé testy jsem využil pouze 5 z původních 6 sloupců, zbyly tedy 4 prediktory a odhadovaná hodnota – proměna odezvy. Protože predikovaná hodnota je spojitá, jedná se o regresní úlohu stejně jako u REAL ESTATE DATASETU.

Pokud porovnáme obrázek C.7 matice originálního datasetu spolu s obrázky C.9 a C.8 (v přílohách) uvidíme, že rozdíly mezi nimi jsou zanedbatelné. To odpovídá i příslušným hodnotám v tabulce 4.4. Co se týče chyby při regresi, MSE je v tomto případě nejmenší z testovaných datasetů.

Tabulka 4.4: Výsledky testů – WSNS DATASET

Model	*random forest*	KNN	L2 vzdálenost korelačních matic
VAE	0,05	0,051	0,011
GANs	0,0416	0,053	0,01
TVAE	0,111	0,106	0,03
CTGAN	0,1493	0,185	0,038
Originální data	0,039	0,056	0

4.5 Shrnutí experimentů

Z výše uvedených experimentů je zřejmé, že vytvořené generátory jsou schopné generovat data se stejnými závislostmi, jako mají původní data. To je pozorovatelné jak z vizuální podoby korelačních matic, tak z L2 vzdálenosti mezi nimi. Klasifikační a regresní testy dále ukázaly, že lze použít modely naučené na vygenerovaných datech bez nutnosti dalších úprav na data původní.

Z výsledků experimentů nelze jednoznačně určit úspěšnější implementaci modelu. Data vytvořená GANs byla opakovaně nejúspěšnějšími, ale vyskytovaly se i případy, kdy tato data zdaleka neodpovídala původním datům. Oproti GANs se z provedených experimentů VAE jeví jako stabilnější model. Další nevýhodou GANs je doba učení. Na více rozměrném datasetu (např. DRY BEAN DATASET) je doba učení mnohokrát delší než doba učení VAE. Totéž bylo možné pozorovat i na modelu CTGAN.

Závěr

Hlavním cílem práce bylo vytvoření generátoru, který je schopný generovat tabulková data na základě vstupních dat. Záměrem bylo, aby nově vygenerovaná data reflektovala statistické vlastnosti původních dat. Dalšími cíli bylo vytvoření přehledu způsobů generování dat a představení teoretického podkladu zvolených technik.

Nejprve jsem shrnul různé metody pro generování dat, ze kterých jsem na základě jejich vlastností vybral variační autoenkodéry a *Generative adversarial networks*. Oba modely jsem podrobně popsal a představil jejich teoretické podklady. V praktické části jsem se věnoval implementaci zvolených modelů.

Výsledkem práce jsou 2 datové generátory, které jsou schopné generování numerických i kategorických hodnot na základě vstupu. Jednotlivé generátory jsem otestoval na následujících datasetech: REAL ESTATE VALUATION, DRY BEAN, BREAST CANCER COIMBRA a PREDICTION OF AVERAGE LOCALIZATION ERROR IN WSNS. Následně jsem vyhodnotil kvalitu vygenerování dat za použití modelů pro klasifikaci a regresi. Jmenovitě se jednalo o metody *Random forest* a k nejbližších sousedů. Kromě klasifikačních a regresních testů jsem porovnal korelační matice vygenerovaných dat s původními a měřil L2 vzdálenost pro jednotlivé prvky matic. Z provedených experimentů je zřejmé, že vygenerovaná dala reflektují statistické vlastnosti původních dat a modely naučené na syntetických datech lze použít na originální data bez nutnosti *transfer learning*.

S rostoucím výkonem počítačů, přibývajícím množství algoritmů strojového učení a naléhavostí využití dat lze předpokládat, že v následujících letech dojde ke vzniku nových algoritmů pro generování dat. Z toho důvodu je vhodné se k tématu této práce později vrátit a rozšířit ji o nové modely.

Literatura

[1] Dertat, A.: Applied Deep Learning - Part 3: Autoencoders. 2017, [Online], [cit. 25.4.2022]. Dostupné z: `https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798`

[2] Kuo, J. C.: AI Classical Music Composer — Bi-LSTM & CNN-GAN. 2021, [Online], [cit. 25.4.2022]. Dostupné z: `https://medium.com/analytics-vidhya/ai-classical-music-composer-63d983ee5fc0`

[3] Popić, S.; Pavković, B.; Velikić, I.; aj.: Data Generators: a short survey of techniques and use cases with focus on testing. In *2019 IEEE 9th International Conference on Consumer Electronics (ICCE-Berlin)*, 2019, s. 189–194.

[4] Thomas, D. B.; Luk, W.; Leong, P. H. W.; aj.: Gaussian random number generators. *ACM Computing Surveys*, ročník 39, č. 4, 2007: s. 11–es.

[5] Doudali, T. D.; Konstantinou, I.; Koziris, N.: Spaten: a Spatio-temporal and Textual Big Data Generator. In *2017 IEEE International Conference on Big Data (BIGDATA)*, 2017, s. 3416–3421.

[6] Ruscio, J.; Kaczetow, W.: Simulating multivariate nonnormal data using an iterative algorithm. *Multivariate Behav Res*, ročník 43, č. 3, 2008: s. 355–381.

[7] Fleishman, A.: A method for simulating nonnormal distributions. *Psychometrika*, ročník 43, č. 4, 1978: s. 521–532.

[8] Navara, M.: *Pravděpodobnost a matematická statistika*, ročník 1. Praha: Skriptum FEL ČVUT, 2007, 23 s.

[9] Vale, C. D.; Maurelli, V. A.: Simulating multivariate nonnormal distributions. *Psychometrika*, ročník 48, č. 3, 1983: s. 465–471.

[10] Foldnes, N.; Olsson, U. H.: A Simple Simulation Technique for Nonnormal Data with Prespecified Skewness, Kurtosis, and Covariance Matrix. *Multivariate Behavioral Research*, ročník 51, č. 2–3, 2016: s. 207–219.

[11] Buscema, M.: Genetic doping algorithm (GenD): theory and applications. *Expert Systems*, ročník 21, č. 2, 2004: s. 63–79.

[12] Meraviglia, C.; Massini, G.; Croce, D.; aj.: GenD an evolutionary system for resampling in survey research. *Quality & Quantity*, ročník 40, č. 5, 2006: s. 825–859.

[13] Friedman, J. H.: An Overview of Predictive Learning and Function Approximation. *From Statistics to Neural Networks*, ročník 136, 1994: s. 1–61.

[14] Robnik-Šikonja, M.: Data Generators for Learning Systems Based on RBF. *IEEE Transactions on Neural Networks and Learning Systems*, ročník 27, č. 5, 2016: s. 926–938.

[15] Billings, S. A.; Zheng, G.: Radial basis function network using genetic algorithms. *Neural Networks*, ročník 8, č. 6, 1995: s. 877–890.

[16] Goodfellow, I.; Pouget-Abadie, J.; Mirza, M.; aj.: Generative adversal nets. In *Advances in Neural Information Processing Systems*, Curran Associates, Inc., 2014, s. 2672–2680.

[17] Pan, Z.; Yu, W.; Yi, X.; aj.: Recent Progress on Generative Adversarial Networks (GANs): A Survey. *IEEE access*, ročník 7, 2019: s. 36322–36333.

[18] Bellman, R.: Dynamic Programming. *Science*, ročník 153, č. 3731, 1966: s. 34–37.

[19] Wang, W.; Huang, Y.; Wang, Y.; aj.: Generalized Autoencoder: A Neural Network Framework for Dimensionality Reduction. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, 2014, s. 490–497.

[20] Vahdat, A.; Kautz, J.: NVAE: A Deep Hierarchical Variational Autoencoder. In *Advances in Neural Information Processing Systems*, ročník 33, editace H. Larochelle; M. Ranzato; R. Hadsell; M. F. Balcan; H. Lin, Curran Associates, Inc., 2020, s. 19667–19679.

[21] Reiter, E.; Dale, R.: Building applied natural language generation systems. *Natural Language Engineering*, ročník 3, č. 1, 1997: s. 57–87.

[22] Robertson, A. M.; Willett, P.: Applications of n-grams in textual information systems. *Journal of Documentation*, ročník 54, č. 1, 1998: s. 48–67.

[23] Sidorov, G.; Velasquez, F.; Stamatatos, E.; aj.: Syntatic N-grams as machine learning features for natural language processing. *Expert Systems with Applications*, ročník 41, č. 3, 2014: s. 853–860.

[24] Knight, K.; Hatzivassiloglou, V.: Two-Level, Many-Paths Generation. In *Proceedings of the 33rd Annual Meeting of the Association for Computational Linguistics (ACL-95)*, MA, 1995, s. 252–260.

[25] Church, K. W.; Gale, W. A.: A comparison of the enhanced Good-Turing and deleted estimation methods for estimating probabilities of English bigrams. *Computer Speech & Language*, ročník 5, č. 1, 1991: s. 19–54.

[26] Ratnaparkhi, A.: Trainable Methods for Surface Natural Language Generation. In *Proceedings of the 1st Annual North American Association of Computational Linguistics*, NAACL, 2000, s. 194–201.

[27] Kingma, D.; Welling, M.: Auto-Encoding Variational Bayes. In *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 12 2014.

[28] Kingma, D. P.; Welling, M.: An Introduction to Variational Autoencoders. *Foundations and Trends® in Machine Learning*, ročník 12, č. 4, 2019: s. 307–392, ISSN 1935-8237.

[29] Kullback, S.; Leibler, R. A.: On Information and Sufficiency. *The Annals of Mathematical Statistics*, ročník 22, č. 1, 1951: s. 79 – 86.

[30] Asperti, A.; Trentin, M.: Balancing Reconstruction Error and Kullback-Leibler Divergence in Variational Autoencoders. *IEEE Access*, ročník 8, 2020: s. 199440–199448.

[31] Xu, L.; Skoularidou, M.; Cuesta-Infante, A.; aj.: Modeling Tabular data using Conditional GAN. In *Advances in Neural Information Processing Systems*, ročník 32, editace H. Wallach; H. Larochelle; A. Beygelzimer; F. d'Alché-Buc; E. Fox; R. Garnett, Curran Associates, Inc., 2019, [Online], [cit. 29.4.2022]. Dostupné z: `https://proceedings.neurips.cc/paper/2019/file/254ed7d2de3b23ab10936522dd547b78-Paper.pdf`

[32] Creswell, A.; White, T.; Dumoulin, V.; aj.: Generative Adversarial Networks: An Overview. *IEEE Signal Processing Magazine*, ročník 35, č. 1, 2018: s. 53–65.

[33] Salimans, T.; Goodfellow, I.; Zaremba, W.; aj.: Improved Techniques for Training GANs. In *Advances in Neural Information Processing Systems*, ročník 29, editace D. Lee; M. Sugiyama; U. Luxburg; I. Guyon; R. Garnett, Curran Associates, Inc., 2016, s. 2226–2234.

[34] Arjovsky, M.; Bottou, L.: Towards Principled Methods for Training Generative Adversarial Networks. 2017, doi:10.48550/ARXIV.1701.04862, [Online], [cit. 28.4.2022]. Dostupné z: https://arxiv.org/abs/1701.04862

[35] Radford, A.; Metz, L.; Chintala, S.: Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. 2015, doi:10.48550/ARXIV.1511.06434, [Online], [cit. 29.4.2022]. Dostupné z: https://arxiv.org/abs/1511.06434

[36] Dosovitskiy, A.; Fischer, P.; Springenberg, J. T.; aj.: Discriminative Unsupervised Feature Learning with Exemplar Convolutional Neural Networks. 2014, doi:10.48550/ARXIV.1406.6909, [Online], [cit. 29.4.2022]. Dostupné z: https://arxiv.org/abs/1406.6909

[37] Arjovsky, M.; Chintala, S.; Bottou, L.: Wasserstein Generative Adversarial Networks. In *Proceedings of the 34th International Conference on Machine Learning*, *Proceedings of Machine Learning Research*, ročník 70, editace D. Precup; Y. W. Teh, PMLR, 06–11 Aug 2017, s. 214–223, [Online], [cit. 29.4.2022]. Dostupné z: https://proceedings.mlr.press/v70/arjovsky17a.html

[38] Yeh, I.-C.; Hsu, T.-K.: Building Real Estate Valuation Models with Comparative Approach through Case-Based Reasoning. *Appl. Soft Comput.*, ročník 65, č. C, apr 2018: str. 260–271, ISSN 1568-4946, doi:10.1016/j.asoc.2018.01.029, [Online], [cit. 30.4.2022]. Dostupné z: https://doi.org/10.1016/j.asoc.2018.01.029

[39] Koklu, M.; Ozkan, I. A.: Multiclass classification of dry beans using computer vision and machine learning techniques. *Computers and Electronics in Agriculture*, ročník 174, 2020: str. 105507, ISSN 0168-1699, doi:https://doi.org/10.1016/j.compag.2020.105507, [Online], [cit. 29.4.]. Dostupné z: https://www.sciencedirect.com/science/article/pii/S0168169919311573

[40] Patrício, M.; Pereira, J.; Crisóstomo, J.; aj.: Using resistin, glucose, age and BMI to predict the presence of breast cancer - BMC cancer. Jan 2018, [Online],[cit. 30.4.2022. Dostupné z: https://bmccancer.biomedcentral.com/articles/10.1186/s12885-017-3877-1

[41] Singh, A.; Kotiyal, V.; Sharma, S.; aj.: A Machine Learning Approach to Predict the Average Localization Error With Applications to Wireless Sensor Networks. *IEEE Access*, ročník 8, 11 2020: s. 208253 – 208263, doi:10.1109/ACCESS.2020.3038645, [Online], [cit. 30.4.2022].

Příloha **A**

Seznam použitých zkratek

VM Vale a Maurelli
IG nezávislý generátor (*independant generator*)
GenD *Genetic doping algorithm*
GA genetický algoritmus
RBF *Radial basis function*
VAE variační autoenkodér
GANs *Generative adversarial networks*
NLG generátory pro přirozený jazyk (*Natural language generators*)
ELBO *evidence lower bound*
KL Kullback-Leiber
TVAE *Tabular variational autoencoder*
DCGAN *Deep convolutional generative adversarial networks*
WGAN *Wasserstein generative adversarial networks*
CTGAN *Conditional tabular generative adversarial networks*
KNN k nejbližších sousedů (*k-nearest neighbors*)
MSE střední kvadratická chyba (*mean squared error*)

Příloha B

Zdrojové kódy

Kód B.1: vae.py

```
import torch
from torch import nn
from torch.nn import functional as F

import pandas as pd

import numpy as np

from sklearn.preprocessing import StandardScaler, OneHotEncoder

class Encoder(nn.Module):
  def __init__(
    self, input_dim, latent_dim = 32, hidden_params = [256, 256]
    ):
      super(Encoder, self).__init__()
      hidden_layers = []
      input = input_dim
      for layer in hidden_params:
        hidden_layers += [
                          nn.Linear(input, layer),
                          nn.Dropout(0.3),
                          nn.ReLU()

        ]
        input = layer

      self.__model = nn.Sequential(*hidden_layers)
      self.__mean = nn.Linear(input, latent_dim)
      self.__logvar = nn.Linear(input, latent_dim)

  def forward(self, x):
```

```
    hidden = self.__model(x)
    return self.__mean(hidden), self.__logvar(hidden)

class Decoder(nn.Module):
  def __init__(
    self, output_dim, input_dim = 32, hidden_params = [256, 256]
    ):
      super(Decoder, self).__init__()
      hidden_layers = []
      input = input_dim
      for layer in hidden_params:
        hidden_layers += [
                          nn.Linear(input, layer),
                          nn.Dropout(0.3),
                          nn.ReLU()
        ]
        input = layer

      hidden_layers += [nn.Linear(input, output_dim)]
      self.__model = nn.Sequential(*hidden_layers)

  def forward(self, x):
    return self.__model(x)

class VAE():
  def __init__(
    self, latent_dim = 32, e_params = [256, 256],
    d_params = [256, 256],
    num_epochs = 300, batch_size = 512, lr = 1e-3
    ):
    self.__encoder = None
    self.__decoder = None
    self.__continous_scaler = StandardScaler()
    self.__categorical_scaler = OneHotEncoder()
    self.latent_dim = latent_dim
    self.e_params = e_params
    self.d_params = d_params
    self.num_epochs = num_epochs
    self.batch_size = batch_size
    self.lr = lr

  def fit(self, data, categorical_columns = []):
    continous, categorical = self.__split(data, categorical_columns)
    self.__categorical_columns = categorical_columns
    self.__continous_columns = list(continous.columns)
    self.columns = self.__continous_columns
    + self.__categorical_columns

    self.__continous_scaler.fit(continous)
```

```
        self.__categorical_scaler.fit(categorical)
        continous, categorical = self.__transform(continous, categorical)
        d1 = pd.DataFrame(continous, columns=self.__continous_columns)
        data = d1.join(
            pd.DataFrame(
              categorical,
              columns=self.__categorical_scaler.get_feature_names_out()
            )
        )

        self.data_dim = data.shape[1]
        self.__encoder = Encoder(
          self.data_dim, self.latent_dim, self.e_params
        )
        self.__decoder = Decoder(
          self.data_dim, self.latent_dim, self.d_params
        )

        self.__train(data)

    def __transform(self, continous, categorical):
        ctn = self.__continous_scaler.transform(continous)
        ctg = self.__categorical_scaler.transform(categorical)
        return ctn, ctg.toarray().astype(np.float64)

    def __split(self, data, categorical_columns):
        categorical = data[categorical_columns]
        continous = data.drop(categorical_columns, axis=1)
        return continous, categorical

    def __train_loader(self, data):
        d = torch.tensor(data.values)
        d = d.type(dtype=torch.float32)

        train_loader = torch.utils.data.DataLoader(
            d, batch_size=self.batch_size, shuffle=True
        )
        return train_loader

    def __train(self, data):
        optimizer = torch.optim.Adam(
            list(self.__encoder.parameters()) +
            list(self.__decoder.parameters()
            ),
            lr = self.lr,
            weight_decay=1e-6)
        self.training = True
        data_loader = self.__train_loader(data)
```

```
    for epoch in range(self.num_epochs):
      for i, batch in enumerate(data_loader):
        optimizer.zero_grad()
        mu, logvar = self.__encoder(batch)
        z = self.__reparametrize(mu, logvar)
        reconstruction = self.__decoder(z)

        loss = self.__loss_function(batch, reconstruction, mu, logvar)
        loss.backward()
        optimizer.step()

        if epoch % 10 == 0 and i == 0:
          print(f"Epoch: {epoch}, loss: {loss}")

  def __reparametrize(self, mu, logvar):
    if self.training:
      std = torch.exp(0.5 * logvar)
      eps = torch.randn_like(std)
      return eps * std + mu
    return mu

  def __loss_function(self, data, reconstruction, mu, logvar):
    MSE = F.mse_loss(
      input=reconstruction, target=data, reduction='sum'
    )
    KLD = -0.5 * torch.sum(
      1 + logvar - mu.pow(2) - logvar.exp()
    )
    return MSE + KLD

  def sample(self, quantity = 1000):
    self.training = False
    with torch.no_grad():
      latent_samples = torch.randn(quantity, self.latent_dim)
      generated_samples = self.__decoder(latent_samples)
      df = pd.DataFrame(
        generated_samples.detach(),
        columns = list(self.__continous_columns) + list(
          self.__categorical_scaler.get_feature_names_out())
      )
      continous, categorical = self.__split(
        df, self.__categorical_scaler.get_feature_names_out()
      )

      if self.__continous_columns:
        scaled_continous = self.__continous_scaler.inverse_transform(
          continous
        )
        if self.__categorical_columns:
```

```
        scaled_categorical=self.__categorical_scaler.inverse_transform
        (
          categorical
        )
        return np.concatenate((scaled_continous, scaled_categorical),
        axis=1)
      return scaled_continous
    scaled_categorical = self.__categorical_scaler.inverse_transform(
      categorical
    )
    return scaled_categorical
```

Kód B.2: gans.py

```
import torch
from torch import nn
from torch.nn import functional as F

import pandas as pd

import numpy as np
from sklearn.preprocessing import StandardScaler, OneHotEncoder

class Discriminator(nn.Module):
  def __init__(self, input_dim, hidden_layers_params):
    super(Discriminator, self).__init__()
    layers = []
    input = input_dim
    for layer_param in hidden_layers_params:
      layers += [
        nn.Linear(input, layer_param),
        nn.LeakyReLU(0.2),
        nn.Dropout(0.5)
      ]
      input = layer_param
    layers += [nn.Linear(input, 1)]
    self.model = nn.Sequential(*layers)

  def forward(self, x):
    output = self.model(x)
    return torch.sigmoid(output)

class Generator(nn.Module):
  def __init__(self, input_dim, output_dim, hidden_layers_params):
    super(Generator, self).__init__()
    layers = []
    input = input_dim
    for layer_param in hidden_layers_params:
      layers += [
        nn.Linear(input, layer_param),
        nn.ReLU()
        ]
      input = layer_param
    layers += [nn.Linear(input, output_dim)]
    self.model = nn.Sequential(*layers)

  def forward(self, x):
    output = self.model(x)
    return output

class GAN():
  def __init__(self, latent_dim = 64, d_params=[256, 256],
```

```
                g_params=[256, 256], num_epochs = 300,
                batch_size = 512, loss_function = nn.BCELoss(),
                lr = 0.002, k=5
                ):
        self.__discriminator = None
        self.__generator = None
        self.__continous_scaler = StandardScaler()
        self.__categorical_scaler = OneHotEncoder()
        self.d_params = d_params
        self.g_params = g_params
        self.latent_dim = latent_dim
        self.num_epochs = num_epochs
        self.batch_size = batch_size
        self.loss_function = loss_function
        self.lr = lr
        self.k = k

    def fit(self, data, categorical_columns = []):
        continous, categorical = self.__split(data, categorical_columns)
        self.__categorical_columns = categorical_columns
        self.__continous_columns = list(continous.columns)

        self.columns = self.__continous_columns
        + self.__categorical_columns

        self.__continous_scaler.fit(continous)
        self.__categorical_scaler.fit(categorical)
        continous, categorical = self.__transform(continous, categorical)
        data = pd.DataFrame(
          continous, columns=self.__continous_columns
          ).join(
            pd.DataFrame(categorical,
            columns=self.__categorical_scaler.get_feature_names_out()
            )
        )

        self.data_dim = data.shape[1]
        self.__discriminator = Discriminator(
          self.data_dim, self.d_params
        )
        self.__generator = Generator(
          self.latent_dim, self.data_dim, self.g_params
        )

        self.__train(data, k=self.k)

    def __transform(self, continous, categorical):
        ctn = self.__continous_scaler.transform(continous)
        ctg = self.__categorical_scaler.transform(categorical)
```

```
        return ctn, ctg.toarray().astype(np.float64)

    def __split(self, data, categorical_columns):
        categorical = data[categorical_columns]
        continous = data.drop(categorical_columns, axis=1)
        return continous, categorical

    def __train_loader(self, data):
        d = torch.tensor(data.values)
        d = d.type(dtype=torch.float32)
        d_length = d.size(dim=0)
        d_labels = torch.zeros(d_length)
        train_set = [
                    (d[i], d_labels[i]) for i in range(d_length)
        ]
        train_loader = torch.utils.data.DataLoader(
            train_set, batch_size=self.batch_size, shuffle=True
        )
        return train_loader

    def __train(self, data, k = 5, l = 1):
      optimizer_discriminator = torch.optim.Adam(
        self.__discriminator.parameters(), lr=self.lr,
        betas=(0.5, 0.9), weight_decay=1e-6
      )
      optimizer_generator     = torch.optim.Adam(
        self.__generator.parameters(), lr=self.lr,
        betas=(0.5, 0.9), weight_decay=1e-6
      )
      data_loader = self.__train_loader(data)

      for epoch in range(self.num_epochs):
        for n, (real_samples, _) in enumerate(data_loader):
          batch_size = real_samples.size(dim=0)
          real_samples_labels = torch.ones((batch_size, 1))

          for i in range(k):
            latent_space_samples = torch.randn(
              (batch_size, self.latent_dim), dtype=torch.float
            )

            generated_samples = self.__generator(latent_space_samples)
            generated_samples_labels = torch.zeros((batch_size, 1))

            all_samples = torch.cat(
                (real_samples, generated_samples)
            )
            all_samples_labels = torch.cat(
                (real_samples_labels, generated_samples_labels)
```

```
            )

            # discriminator training
            self.__discriminator.zero_grad()
            output_discriminator = self.__discriminator(all_samples)
            loss_discriminator = self.loss_function(
                output_discriminator, all_samples_labels
            )
            loss_discriminator.backward()
            optimizer_discriminator.step()

        # data for generator training
        for i in range(l):
            latent_space_samples = torch.randn(
              (batch_size, self.latent_dim)
            )

            # generator training
            self.__generator.zero_grad()
            generated_samples = self.__generator(latent_space_samples)
            output_discriminator_generated = self.__discriminator(
              generated_samples
            )
            loss_generator = self.loss_function(
                output_discriminator_generated, real_samples_labels
            )
            loss_generator.backward()
            optimizer_generator.step()

        # Show loss
        if epoch % 10 == 0 and n == 0:
            print(
              f"Epoch: {epoch} Loss Discriminator: {loss_discriminator}"
            )
            print(
              f"Epoch: {epoch} Loss Generator: {loss_generator}"
            )

    def sample(self, quantity = 1000):
        latent_samples = torch.randn(quantity, self.latent_dim)
        generated_samples = self.__generator(latent_samples)
        df = pd.DataFrame(
          generated_samples.detach(),
          columns = list(self.__continous_columns)
          + list(self.__categorical_scaler.get_feature_names_out())
        )
        continous, categorical = self.__split(
          df, self.__categorical_scaler.get_feature_names_out()
        )
```

```
if self.__continous_columns:
  scaled_continous = self.__continous_scaler.inverse_transform(
    continous
  )
  if self.__categorical_columns:
    scaled_categorical = self.__categorical_scaler.inverse_transform(
      categorical
    )
    return np.concatenate(
      (scaled_continous, scaled_categorical), axis=1
    )
  return scaled_continous
scaled_categorical = self.__categorical_scaler.inverse_transform(
  categorical
)
return scaled_categorical
```

Kód B.3: setup.py

```
from xml.etree.ElementInclude import include
from setuptools import setup, find_packages

setup(
    name="TabularDataGenerator",
    version='1.0.0',
    python_requires='>=3.10.4',
    packages=find_packages(
        include=['Data_Generators', 'Data_Generators.*']
        ),
    install_requires=[
        'pandas',
        'numpy',
        'scikit-learn',
        'torch>=1.8.0,<2'
    ]
)
```

Kód B.4: generate_script.py

```
from random import sample
from Data_Generators.generators.gans import GAN
from Data_Generators.generators.vae import VAE
from os import path
import torch
import pandas as pd
import sys

def train(model_type, data, categorical_columns):
  if model_type == 'vae':
    model = VAE()
  else:
    model = GAN()
  df = pd.read_csv(data, sep=',')
  model.fit(df, categorical_columns=categorical_columns)
  return model

def generate(
  data, PATH, sample_size=100, categorical_columns=[],
  model_type='vae'
  ):
  if not path.exists(PATH):
    model = train(
      model_type=model_type, data=data,
      categorical_columns=categorical_columns
    )
    torch.save(model, PATH)
  else:
    model = torch.load(PATH)
  samples = model.sample(sample_size)
  df = pd.DataFrame(samples, columns=model.columns)
  return df

def set_args(
  sample_size=100, categorical_columns="", model_type='vae'
  ):
    categorical_columns = list(categorical_columns.split(','))
    sample_size = int(sample_size)
    return sample_size, categorical_columns, model_type

def main():
    args = sys.argv
    if len(args) < 3:
        print('Insufficient amount of arguments.')
        return
    data_path=args[1]
    size = len(args)
    match size:
```

```
      case 4:
        sample_size, categorical_columns, model_type = set_args(
          args[3]
        )
      case 5:
        sample_size, categorical_columns, model_type = set_args(
          args[3], args[4]
        )
      case 6:
        sample_size, categorical_columns, model_type = set_args(
          args[3], args[4], args[5]
        )
      case _:
        sample_size, categorical_columns, model_type = set_args()
    df = generate(
      data=data_path, PATH=args[2], sample_size=sample_size,
      categorical_columns=categorical_columns, model_type=model_type
    )
    df.to_csv('generated_samples.csv')


if __name__ == "__main__":
    main()
```

Kód B.5: eval.py

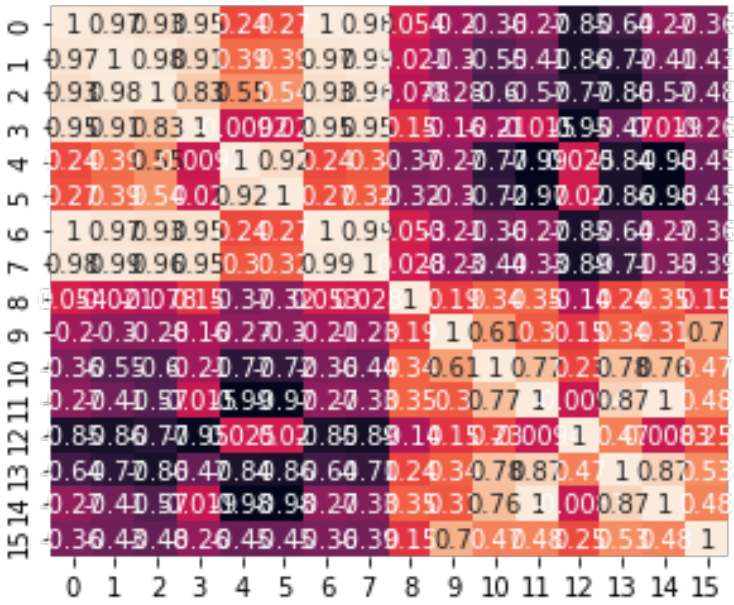
```
import pandas as pd
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import RandomForestRegressor
from sklearn.neighbors import KNeighborsClassifier
from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

def evaluate_randomforest(
  original, generated, target_column, classification=True
  ):
  train_data = generated.drop([target_column], axis=1).to_numpy()
  train_values = generated[target_column].to_numpy()
  test_data = original.drop([target_column],axis=1).to_numpy()
  test_values = original[target_column].to_numpy()

  x_train, x_test, y_train, y_test = train_test_split(
    test_data, test_values
  )

  if classification:
    model = RandomForestClassifier()
    test_model = RandomForestClassifier()
  else:
    model = RandomForestRegressor()
    test_model = RandomForestRegressor()

  model.fit(train_data, train_values)
  test_model.fit(x_train, y_train)

  if classification:
    model_score = model.score(test_data, test_values)
    test_score = test_model.score(x_test, y_test)
  else:
    model_score = mean_squared_error(
        test_values, model.predict(test_data)
    )
    test_score = mean_squared_error(
        y_test, test_model.predict(x_test)
    )
  return model_score, test_score


def evaluate_knn(
  original, generated, target_column, classification=True
  ):
  train_data = generated.drop([target_column], axis=1).to_numpy()
```

```
  train_values = generated[target_column].to_numpy()
  test_data = original.drop([target_column],axis=1).to_numpy()
  test_values = original[target_column].to_numpy()

  x_train, x_test, y_train, y_test = train_test_split(
    test_data, test_values
  )

  if classification:
    model = KNeighborsClassifier()
    test_model = KNeighborsClassifier()
  else:
    model = KNeighborsRegressor()
    test_model = KNeighborsRegressor()

  model.fit(train_data, train_values)
  test_model.fit(x_train, y_train)

  if classification:
    model_score = model.score(test_data, test_values)
    test_score = test_model.score(x_test, y_test)
  else:
    model_score = mean_squared_error(
        test_values, model.predict(test_data)
    )
    test_score = mean_squared_error(
        y_test, test_model.predict(x_test)
    )
  return model_score, test_score


def create_statistics(
  data, target_column, model, categorical_columns = [],
  k = 100, classification = False, sample_size = 10000,
  method='randomforest'
  ):
  model.fit(data, categorical_columns)
  model_score = []
  test_score = []
  for _ in range(k):
    samples = model.sample(sample_size)
    samples_df = pd.DataFrame(samples, columns=model.columns)
    if method == 'randomforest':
      x, y = evaluate_randomforest(
        data, samples_df, target_column=target_column,
        classification=classification
      )
    else:
      x, y = evaluate_knn(
```

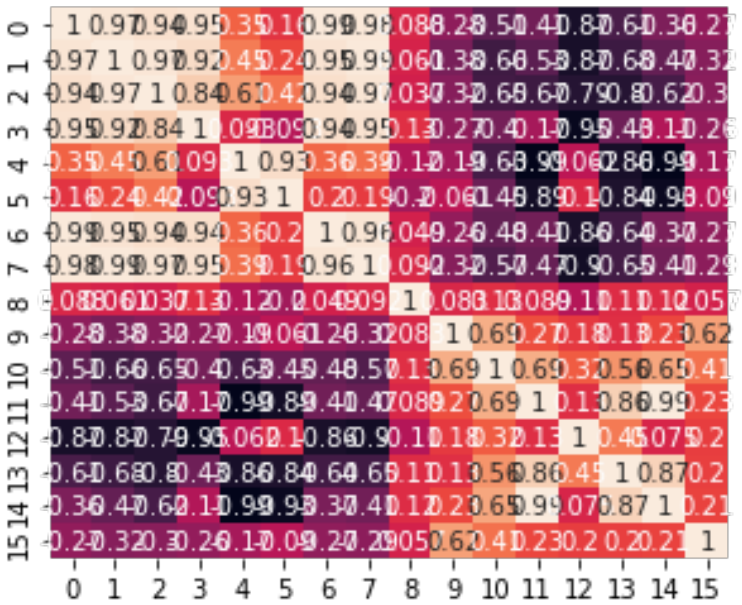
```
            data, samples_df, target_column=target_column,
            classification=classification
        )
      model_score.append(x)
      test_score.append(y)
  return np.array(model_score), np.array(test_score)


def distance (og, new):
  m = og.to_numpy() - new.to_numpy()
  return np.linalg.norm(m) / m.shape[0]**2
```

Příloha C

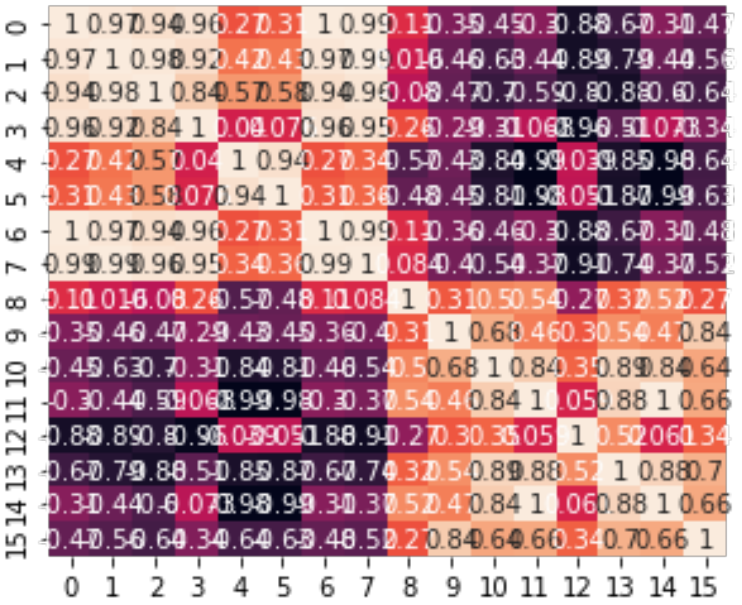
Korelační matice



Obrázek C.1: Korelační matice originálu DRY BEAN DATASETU



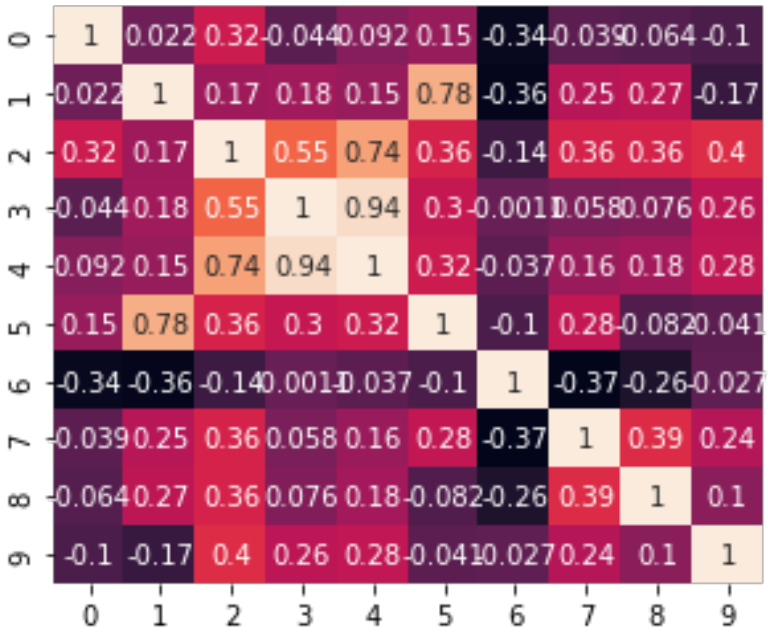
Obrázek C.2: Korelační matice vygenerovaného dry bean datasetu GANs



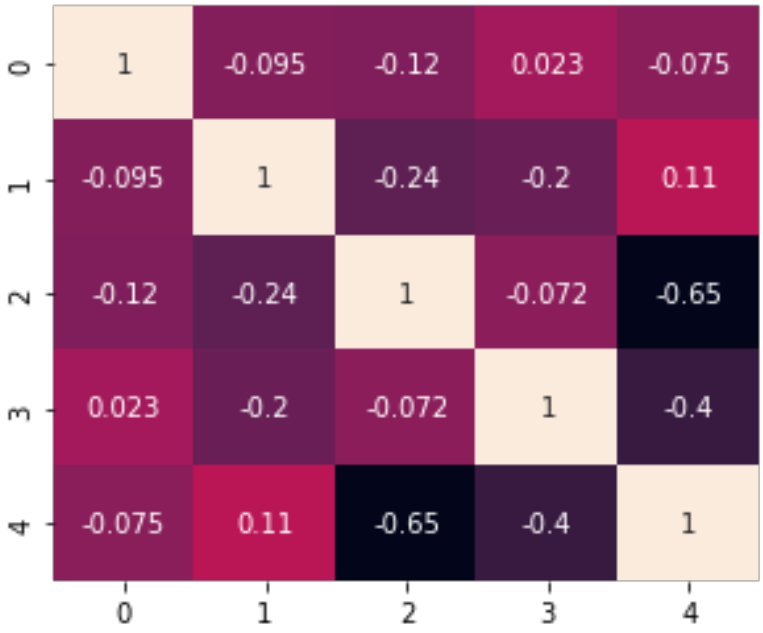
Obrázek C.3: Korelační matice vygenerovaného dry bean datasetu VAE

Obrázek C.4: Korelační matice originálu BREAST CANCER DATASETU

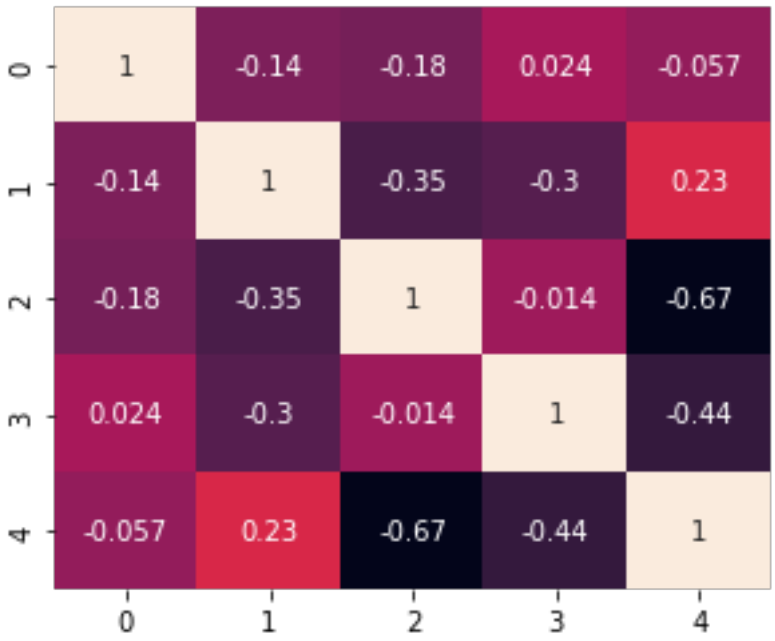
Obrázek C.5: Korelační matice vygenerovaného BREAST CANCER DATASETU GANs



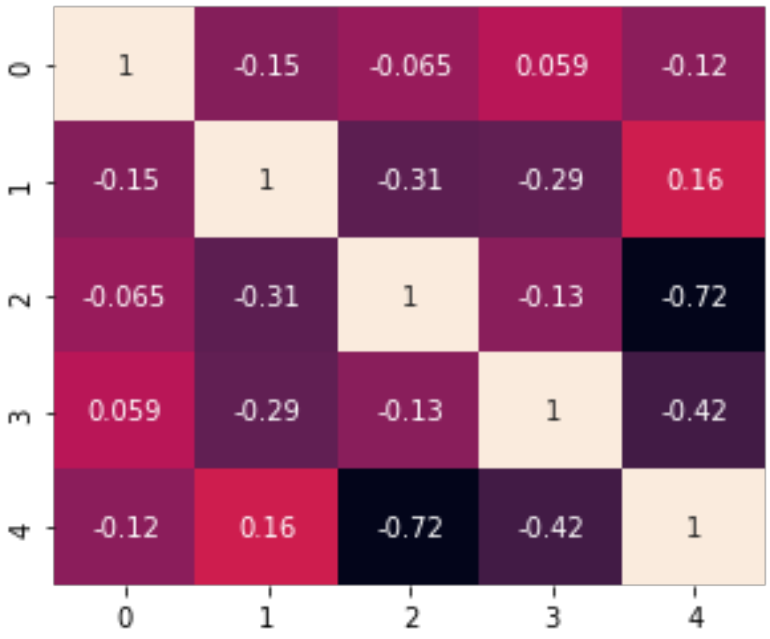
Obrázek C.6: Korelační matice vygenerovaného BREAST CANCER DATASETU VAE



Obrázek C.7: Korelační matice originálu WSNS DATASETU



Obrázek C.8: Korelační matice vygenerovaného WSNS DATASETU GANs



Obrázek C.9: Korelační matice vygenerovaného WSNS DATASETU VAE