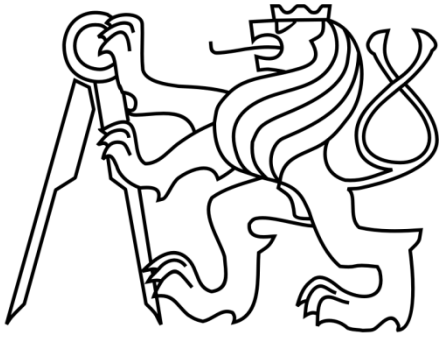
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

FAKULTA STROJNÍ
ÚSTAV MECHANIKY, BIOMECHANIKY A MECHATRONIKY



Diplomová práce

Použití moderních technologií v predikci plánů výroby a jejich vyhodnocení pomocí BI nástrojů

Praha, 2019 **Bc. Daniel Malachov**

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Malachov** Jméno: **Daniel** Osobní číslo: **410056**

Fakulta/ústav: **Fakulta strojní**

Zadávající katedra/ústav: **Ústav mechaniky, biomechaniky a mechatroniky**

Studijní program: **Průmysl 4.0**

Studijní obor: **bez oboru**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Metody predikce plánů výroby a jejich vyhodnocení pomocí BI nástrojů

Název diplomové práce anglicky:

Methods of prediction of production plans and their evaluation using BI tools

Pokyny pro vypracování:

Úlohou diplomové práce je predikovat výrobní data, konkrétně počet vyrobených kusů a časový fond výroby a navhrnout koncepci predikčního systému pro obecné použití. Hlavním úkolem je predikovat výrobní prostoje za účelem jejich následné minimalizace. V teoretické části definujte problémy a dílčí úlohy, proveďte rešerši možných postupů a analýzu možných řešení. V praktické části, popište data a vybrané metody a jejich nastavení, implementujte a vyhodnoťte. Na základě poznatků z teoretické i experimentální části a ze znalosti reálného prostředí, navhrněte koncepci systému pro doporučování správného modelu predikce a rozhraní pro snadný import dat a pokud možno co nejrychlejší praktické nasazení. Rozsah práce min. 50 stran + přílohy

Seznam doporučené literatury:

[1] Fogel, David B, Derong Liu a James M Keller. Fundamentals of Computational Intelligence: Neural Networks, Fuzzy Systems, and Evolutionary Computation. Hoboken, NJ, USA: John Wiley & Sons, Inc., 2016. ISBN 978-1-119-21440-3. doi:10.1002/9781119214403 (e-book https://ebookcentral.proquest.com/lib/cvut/detail.action?docID=4592115)
[2] BUKOVSKY, Ivo a Noriyasu HOMMA. An Approach to Stable Gradient-Descent Adaptation of Higher Order Neural Units. IEEE Transactions on Neural Networks and Learning Systems [online]. 2016, 1–13. ISSN 2162-237X, 2162-2388. Dostupné z: doi:10.1109/TNNLS.2016.2572310

Jméno a pracoviště vedoucí(ho) diplomové práce:

doc. Ing. Ivo Bukovský, Ph.D., U12110.3

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **29.04.2019** Termín odevzdání diplomové práce: **16.08.2019**

Platnost zadání diplomové práce: _____

doc. Ing. Ivo Bukovský, Ph.D. podpis vedoucí(ho) práce	prof. Ing. Milan Růžička, CSc. podpis vedoucí(ho) ústavu/katedry	prof. Ing. Michael Valášek, DrSc. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

30.4.2019 _____

Datum převzetí zadání Podpis studenta

Anotační list

Jméno autora:	Daniel Malachov
Název diplomové práce:	Použití moderních technologií v predikci plánů výroby a jejich vyhodnocení pomocí BI nástrojů
Anglický název:	Time predictions of production data and visualisation with BI tools
Akademický rok:	2018/2019
Obor studia:	Průmysl 4.0 (Mechatronika)
Ústav/odbor:	Ústav mechaniky, biomechaniky a mechatroniky
Vedoucí bakalářské práce:	doc. Ing. Ivo Bukovský, Ph.D.
Bibliografické údaje:	Počet stran: 82 Počet obrázků: 30 Počet příloh: 1
Klíčová slova:	Predikce, časová řada, business inteligence, pokročilé plánování, ARIMA, machine learning, neuronová síť, automatická volba parametrů, automatická příprava dat
Keywords:	Prediction, time series, business inteligence, advanced planing, ARIMA, machine learning, neural nets, automatic paremeters optimalisation, automatic data preprocessing

Anotace: Práce se zabývá vývinem softwaru predikujícího výrobní data, především časový fond výroby a počet vyrobených kusů. Software musí fungovat naprosto samostatně bez zásahu člověka. Výstupem je zápis predikovaných hodnot do databáze a interaktivní graf výsledků. Program si sám očistí data, vybere nejlepší z 20 vytvořených modelů a sám si najde pro daná data vhodné parametry výpočtu. Software má možnost debug módu, kde je možné vypsat veškerá upozornění a také výsledky i grafy jednotlivých modelů v průběhu výpočtu.

Abstract: This thesis describe software for making production data predictions, especially downtimes. Software is fully automated and it's goal is to write predictions into the database and create fully interactive graph of results. It include automatic data preprocessing, it automatically choose the best of the 20 made models and find the optimal model's parameters for given data. Software has debug mode, where all warnings are displayed and results of all the models are displayed during computing.

Prohlášení

Prohlašuji, že tuto bakalářskou práci jsem vypracoval samostatně. Použitou literaturu a všechny další zdroje uvádím dle příslušných norem. Beru na vědomí, že tato práce je na příslušném místě zveřejněna a je volně k nahlédnutí.

V Praze, dne…… Podpis……

Obsah

Úvod

Pojmy jako průmysl 4, umělá inteligence, nebo machine learning se v poslední době těší relativní mediální oblibě. Všech těchto pojmů se týká i celá následující práce. Zatímco automatizace v oblasti monotónní manuální práce dosáhla již značné úrovně, tato práce se věnuje automatizaci práce kancelářské. Dat je dnes nepřeberné množství a proto je rozumné vytvářet nástroje pro jejich pohodlné používání. Automatizací mám na mysli nikoliv nahrazení kancelářských pozic, ale naopak jejich nástroj pro co největší efektivnost tam, kde potřebujeme z dat vyzískat informace, které mají sloužit pro důležitá firemní rozhodnutí.

Ohledně umělé inteligence mají mnozí lidé velká očekávání. AI marketingová oddělení svými líbivými přísliby získaly mnoho pozornosti. Odměnou může být přísun finančních i lidských zdrojů, trestem může být jejich rychlý odsun v případě nenaplnění očekávání. Tato práce je ukázkou toho, jak konkrétně může vypadat pokus automatizovat část toku dat a ulehčit a zkvalitnit práci dalších lidí. Cílem je vytvořit software schopný predikce, čili předpovídání hodnot v budoucí době. Software sám si musí očistit a přizpůsobit vstupní data, dále zjistit jaký model je vhodný pro výpočet a také si pro ten model najít optimální parametry. Mám trochu pochybnosti o tom, zda algoritmy použité v této práci jsou skutečně inteligentními, ale jsem si jist, že se dají inteligentně použít tak, aby nám lidem spolehlivě sloužily.

Práce je rozdělena na dvě části. První je teoretická, do které spadá podrobnější definice problému, rešerše možných postupů a analýza nejvhodnějších řešení. Druhá část od kapitoly 6, je část praktická, kde bude několik přístupů implementováno, popsáno, a ohodnoceno. Celý kód je součástí přílohy.

1. Definice problematiky a cíle práce

Úlohou této diplomové práce je predikovat výrobní data, konkrétně počet vyrobených kusů a časový fond výroby. Prvním a nejdůležitějším úkolem je predikovat výrobní prostoje za účelem jejich následné minimalizace. O tom, proč je to důležité a jaké výhody to může firmám přinést jsou následující kapitoly.

1.1. Business inteligence

BI je metodika používaná především pro rychlá a správná rozhodnutí. Rozhodnutí jako kdy spustit reklamní kampaň, kdy rozšířit sortiment, kdy vyřadit zastaralé produkty, kdy se pokusit podnikat na novém trhu a kde, to jsou rozhodnutí, která vyžadují krom intuice a znalosti dané problematiky i reálná data. Dat je v dnešní době nadbytek a s tím vzniká i potřeba nových nástrojů pro jejich analýzu. Informační systémy (ERP, MES a podobně) jsou systémy, které nám umožňují mít všechna tato data v přehledné formě. BI je potom nástrojem daných informačních systémů, který má za úkol z dat vytěžit informace. Společným jmenovatelem změn v tomto oboru je především vzhled a propojenost. Propojeností je myšleno jakési oživení dat, přechod od statických tabulek a reportů, k tabulkám dynamickým. Součástí této práce je tedy nejen vytvoření modelu schopného predikce, ale i vyobrazení predikovaných dat v moderních BI nástrojích, pro co největší přehlednost a srozumitelnost. Součástí bude také vizualizace rozdílů mezi jednotlivými modely. Tato práce má pomoci hledat odpovědi na otázky jako kolik kusů a za jak dlouho se vyrobí? Stihnu splnit požadovanou zakázku v termínu? Vystačí velikost skladů, pokud vypadne klíčový odběratel? Jak dlouho? Cílem této práce je přetvořit data v informace, které budou sloužit manažerům v pokročilém plánování.

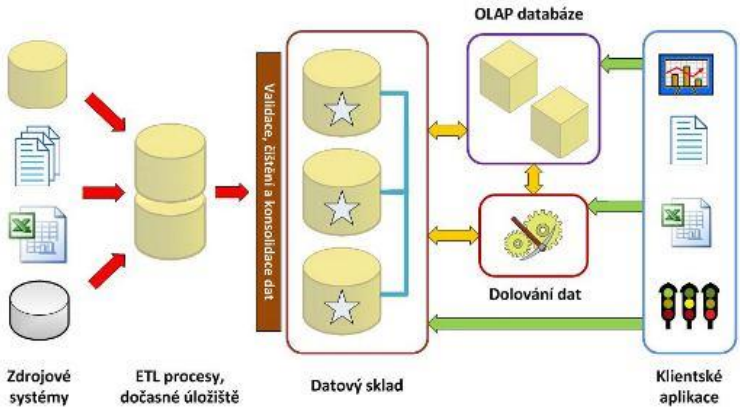
1.2. Tok dat

Vstupní data použitá v této práci máme v SQL Serveru a každý den jsou aktualizovaná. Sběr dat probíhá na jednotlivých linkách. To, jak jsou data importována a dále upravována pro finální vizualizaci nazveme datovým tokem. Data z jednotlivých linek putují do tzv. datawarehousu, jinak také datového skladu. Datový sklad má několik vrstev. První je dočasná databáze nazývaná *stage*, do které importujeme všechna data. Tento mezikrok je nutný především z důvodu co nejmenšího zatížení zdrojů. Způsob, jakým do *stage* data importujeme nazveme ETL (Export, transfer, load) procesem, který také jinak nazýváme datovou pumpou. Další vrstvu datového skladu můžeme nazvat *dbo*. V této vrstvě máme data již vyčištěná, upravená a setříděná. Duplicity jsou zde výhodou. Tabulky dat si rozdělíme na tzv. měřítka a tzv. dimenze. Měřítkem jsou hodnoty důležité pro byznys, jako například počet vyrobených kusů, prostoje, náklady. Dimenze jsou data, která nám pomáhají na měřítka pohlížet z různých úhlů. Například počtu

vyrobených kusů dává rámec to, kdy byly vyrobeny, v jaké hale, na jakém stroji, jakým operátorem. Více o teorii datových skladů je možné nalézt například v [1].

Vhodným nástrojem pro analýzu dat je OLAP (Online Analytical Processing) datová kostka, která je formou databáze optimalizované pro rychlé čtení. Pro finální vizualizaci následně můžeme použít software jako například Tableau, Qlik, nebo Power BI od Microsoftu, který bude využíván v této práci.

Cestu datového toku ilustruje například následující obrázek.



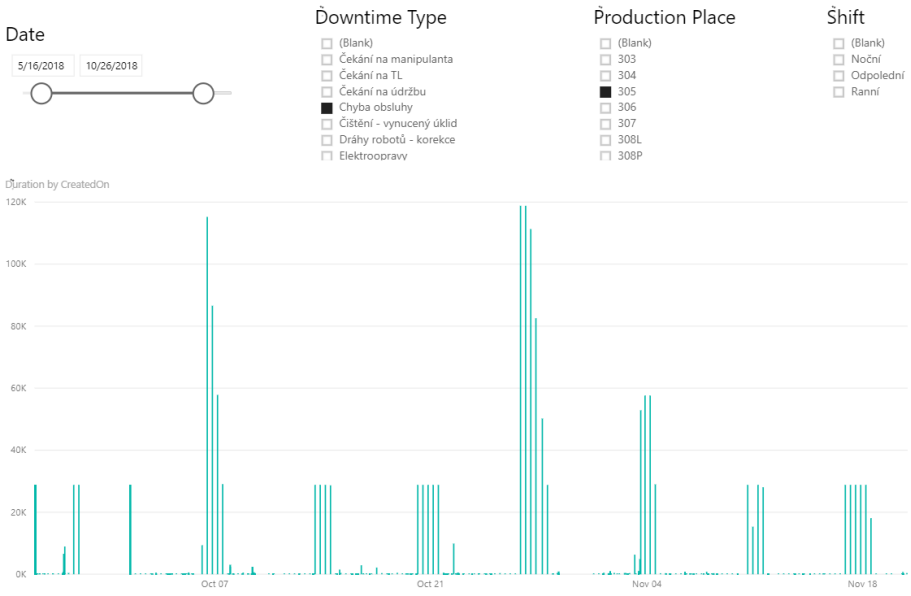
Obrázek 1 - Tok dat firmou - [2]

1.3. Vizualizace vstupních dat pomocí BI nástrojů

Pro kvalitní analýzu je ideální data vhodným způsobem vizualizovat. Dnešním trendem jsou živé grafy, které potřebný počet tabulek a grafů redukují na minimum. Navíc data jsou přehlednější a rychleji k nalezení. Pro počítačovou predikci je nejdříve potřeba definovat jaké prostoje chceme predikovat, na jakém zařízení. Pokud máme v úmyslu zavádět prediktivní údržbu, predikovat prostoje je jednou z možných cest. Snažíme se minimalizovat právě ty prostoje, které způsobují největší komplikace. Vytvořil jsem proto graf, který je praktickou ukázkou toho, jak mohou být prostoje setříděné podle délky trvání, podle typu prostoje, zařízení, pracoviště či směny.

Obrázek 2 - Porovnání prostojů podle délky trvání

Na následujícím grafu se pak můžeme zaměřit na konkrétní data v čase. Nástěnka je interaktivní, takže můžeme analyzovat jednotlivé typy prostojů, které jsme podle grafu výše uznali za kritické. Stačí daný typ prostoje, zařízení a pracoviště, jejichž detaily chceme vědět zaškrtnout v polích výběru nahoře. Nástěnka je součástí přílohy, a to ve formátu *pbix*.



Obrázek 3 - Vyobrazení prostojů na jednotlivých linkách v čase

1.4. Vstupní proměnné do modelu

Velkou otázkou je, co všechno do predikce zahrnout. Na první pohled čím více, tím lépe, ovšem nemusí to platit za všech okolností. V nejoptimálnějším scénáři zahrneme informace o tom, jaké produkty se mají vyrábět a na jakých linkách, jaký je plán výroby, takt strojů, kolik zaměstnanců stroje obsluhuje a kdo konkrétně, kdo bude na dozoru výroby, kdy budou plánované odstávky strojů, zda se blíží nějaký státní svátek. Množství vlivů, které konečnou výrobu ovlivňují je nezměrné, z praktického hlediska však míň může znamenat víc. Problémem nadmíry vstupních parametrů je především ten, že všechny parametry je nutné mít v databázi v budoucím čase. Nutit manažery ručně doplňovat tabulky může přinést více problémů než užitku, proto je lepší spolehnout se na data, která již v nějakém interním informačním systému firma má, a z kterých se tato data dají importovat. Ze sběru dat máme o každém kusu informaci jaký operátor byl u stroje přihlášen, což ovšem nemusí odpovídat realitě toho, kolik lidí se u linky pohybuje. Jedním parametrem je tedy daný operátor, druhým počet zaměstnanců na dané lince. Dalším parametrem, který máme vždy k dispozici jsou časové údaje. To, jestli se blíží svátky, zda je léto, nebo zima. Asi nejdůležitějším údajem jsou však uzavřené zakázky. Tento druh informací mají firmy běžně v ERP systému a lze je snadno importovat. Obsahují nejdůležitější informaci – co je potřeba vyrobit a v jakém množství. Výroba různých produktů trvá různou dobu. Je důležité, aby kupříkladu model pro predikci výroby linky na výlisky z plastu používal jako trénovací i testovací množinu právě data, která se k danému výrobku vztahují a žádná jiná. První a velmi důležitou úlohou, je tedy správný výběr dat pro model. Bylo by nerozumné snažit se predikovat počet vyrobených kusů, nebo trvání prostojů, v celém výrobním závodu najednou. Prvním úkolem bude vždy vyselektovat data z jednoho konkrétního zařízení, vyrábějícího jeden konkrétní produkt.

Vstupní data mohou být dvojího původu. První cestou je import dat z informačního systému. Ten má nejdůležitější informace, totiž nasmlouvané zakázky, a tedy i informace o tom, co se bude vyrábět a kdy. Dále by bylo nutné automatizované rozdělení výroby jednotlivých produktů mezi jednotlivé linky a následný automatizovaný výběr vstupních dat do modelu na základě předchozí znalosti procesu. Někdy už tyto informace firma má. Plánování provádí firmy různě pomocí rozličných nástrojů. Je potřeba zjistit, zda je možná data ze softwaru pro plánování importovat, nebo ne. Pokud bude predikce například řešena nějakou formou regrese, tedy pokud bude námi predikovaná hodnota modelována jako funkce vstupních parametrů, potřebujeme znát ony plánované parametry, které protože se ještě nestaly, nemohou pocházet ze sběru dat.

Pokud firma nemá konkrétní plánovací software, z něhož mohou být data importována, druhou cestou je použití tzv. write-backů. Je to forma, jak jednoduše doplnit data do databáze. Může pro to být použit na míru vytvořený software s grafickým rozhraním (pokud by od uživatele tohoto

softwaru pro predikci byl zájem i pro další modul BI řešení pro pokročilé plánování). Nevýhodou je potom nutnost tyto informace ručně doplňovat. Bez nich predikce nebude možná, nejde tedy o automatizované řešení.

1.5. Výběr, úprava a validace dat

Data nesmí obsahovat chybné ani neznámé hodnoty. Jelikož je tento projekt součástí celkového BI řešení, lze konstatovat, že data jsou z podstatné části předzpracovaná i validovaná. Přesto však budou vytvořeny dodatečné kontroly. To, jaká zvolíme vstupní data, má přinejmenším stejný dopad na výsledek, jako to, jaký model pro predikce zvolíme. Obecně navrhuji dva přístupy. Jeden bude predikovat hodnoty jako funkci několika proměnných (jako může být datum, velikost objednávek, počet zaměstnanců a podobně). Druhý způsob bude využívat pouze minulých stavů sledované hodnoty a její dynamiky. Predikční systémy můžeme rozdělit do několika typů. SISO, MISO, MIMO podle toho, jestli obsahují jeden a nebo více vstupů a výstupů. Praktická část bude popisovat jak metody pro vstup jednorozměrný, tak i vícerozměrný.

Jednou z velmi důležitých otázek je, jak dlouhý vzorek dat použít. Čím větší trénovací množinu jsme schopni poskytnout, tím přesnější by měl být model, úvaha platí jen pro stacionární děje. Lze očekávat, že ve výrobě dochází k neustálým změnám ať už z pohledu personálního, z pohledu rozmístění strojů nebo například z pohledu probíhajících marketingových kampaní. Predikce mají proto vycházet především z těch nejaktuálnějších informací. Tento rozpor může být řešen například spočítáním statistických charakteristik jako je průměr a směrodatná odchylka jednotlivých výsečí a podle nich následné zařazení, nebo nezařazení starších dat. Velmi důležitým parametrem je granularita dat. Data ze sběru dat máme vždy až na nejjemnější rozdělení – jeden vyrobený kus. Vyvstává tedy před námi otázka, zda data před vstupem modelu agregovat. Pokud ano, tak další otázkou je, do jaké úrovně, zda po minutách, nebo hodinách. S tím souvisí otázka, zda udělat pro predikci dnů, týdnů i měsíců jeden model, anebo udělat pro každou kategorii model jiný s jinou výchozí granularitou.

Velmi důležité je zamyslet se nad tím, jaké informace jsou pro predikci nejdůležitější. Chceme předvídat propady, kritické hodnoty, to, jak se bude výroba měnit ze dne na den, nebo nás spíše zajímá průměr za nějaké období? Nabízí se, že budeme predikovat právě průměr a směrodatnou odchylku, a odhadovat jejich vývoj v čase. Podle nich potom budeme predikovat finální hodnoty. Další možností je naopak tyto hodnoty zanedbat a soustředit se pouze na predikování tvaru křivky, onen tvar následně proložit průměrem. Cílová informace může mít binární podobu – roste/neroste. S rozhodnutím kdy jakou strategii použít, nám může pomoci úvaha, že pro velmi krátké predikce se průměr ani směrodatná odchylka nezmění. Čím delší však predikce má být, tím méně nás zajímají rozdíly mezi jednotlivými dny, nebo drobné propady. Pro dlouhodobější předpovědi nás bude z pohledu výroby zajímat spíše kolik se toho za určité období vyrobí. Pokud bude mít

naopak plánovač za úkol zjistit, zda bude mít do tří dnů pokrytou zakázku, bude ho zajímat vývoj hodina od hodiny.

2. Možné algoritmy

Algoritmy rozdělíme na statistické a machine learningové. Teoretická část popisuje pouze princip fungování některých z nich. To jak byly následně implementovány a přizpůsobeny k obrazu mému je následně popsáno v části praktické.

2.1. Statistické metody

- Arima
- ETS
- GARCH
- Theta
- Bayesovská statistika
- Holt-Winters
- EWMA
- GAM-based models

2.2. Machine learningové metody

- Multi-Layer Perceptron (MLP)
- Bayesian Neural Network (BNN)
- Radial Basis Functions (RBF)
- Generalized Regression Neural Networks (GRNN)
- K-Nearest Neighbor regression (KNN)
- CART regression trees (CART)
- Support Vector Regression (SVR)
- Gaussian Processes (GP)
- LSTM
- Neural Networks Autoregression (NNAR)
- RNN (Recurrent Neural Network)
- LSTM (Long Short-Term Memory)
- GRU (Gated Recurrent Unit)

2.3. Validace predikcí, hodnocení modelu

Máme-li modely schopné předpovědí, potřebujeme nástroje, jak jednotlivé modely mezi sebou porovnat. Je hned několik hodnot, které nám s tím mohou pomoci. Jelikož se v praxi hojně používají akronymy, ponechám i anglickou terminologii.

- Mean Error (ME)
- Mean squared error (MSE)
- Root Mean Squared Error (RMSE)
- Mean Absolute Error (MAE)
- Mean Percentage Error (MPE)
- Mean Absolute Percentage Error (MAPE)
- Mean Absolute Scaled Error (MASE)
- Symmetric Mean Absolute Percentage Error (sMAPE)

Co se predikcí týče, tak velmi používanou je RMSE. Jde o odmocninu z MSE, tedy o střední hodnotu z odchylky, nebo také jinak řečeno reziduí, čili rozdílu predikované hodnoty a skutečnosti. Např. podle [3].

$$RMSE = \sqrt{\frac{1}{n}\sum_{j=i}^{n}\left(y_j - \hat{y}_j\right)^2} \qquad MAE = \frac{1}{n}\sum_{j=1}^{n}\left|\left(y_j - \hat{y}_j\right)\right| \tag{2.1}$$

Chceme-li mít měřítko nezávislé na hodnotě, čili měřítko nějakým způsobem normalizované, k dispozici máme MAPE, SMAPE a MASE. Tím, že u těchto metrik nezáleží na konkrétních hodnotách, můžeme porovnávat modely napříč různými datasety. Symetrická verze SMAPE bere při normalizaci v potaz i predikovanou hodnotu. MASE potom porovnává chybu vůči jiné predikci, konkrétně s takzvanou naivní predikcí, čímž se odlišuje od ostatních, díky čemuž může být někdy vhodnější. Jejich jednotlivé srovnání, včetně shrnutí jednotlivých nedostatků je například na [4], odkud s drobnými úpravami pocházejí následující definice.

$$MAPE = \frac{1}{n}\sum_{j=1}^{n}\left|\frac{\left(y_j - \hat{y}_j\right)}{y_j}\right| \tag{2.2}$$

$$SMAPE = \frac{2}{n}\sum_{j=1}^{n}\frac{\left|\left(y_j - \hat{y}_j\right)\right|}{|y_j| + |\hat{y}_j|} \tag{2.3}$$

$$MASE = \frac{1}{n}\sum_{j=1}^{n} \frac{|(y_j - \hat{y}_j)|}{\frac{1}{n-1}\sum_{k=2}^{n} |y_j - y_{j-1}|} \tag{2.4}$$

Celkové množství chyby není jediným měřítkem. Další důležitou informací je pro nás charakteristika této chyby. Pro optimální model by měla mít chyba charakteristiky bílého šumu, tedy nulový průměr, konstantní rozptyl, přibližně normální rozdělení a nulovou korelaci s hledanou funkcí, kterou si lze ověřit například pomocí Ljung-box testu. Ljung-box test definujeme např. z [5]:

$$Q = n(n+2)\sum_{k=1}^{h} \frac{\hat{\rho}_k^2}{n-k} \tag{2.5}$$

Kde n je počet vzorků, $\hat{\rho}_k$ je autokorelace v lagu k, h je číslo již testovaných lagů.

2.4. Automatizovaná volba modelu

Dnes existuje nepřeberné množství modelů. Pro rozhodnutí, jaký model zvolit potřebujeme daný model nějak ohodnotit. Hledítkem pro nás není pouze velikost chyby, ale také komplexnost modelu, jinak řečeno počet proměnných. Existují určitá kritéria, která nám s tím pomohou. Mezi nejznámější patří AIC - Akaike Information Criteria, HQC - Hannan-Quinn Criterion, nebo SBC - Schwarz-Bayesian Criterion, někdy také značeno jako BIC. Obě definice převzaty z [6].

$$AIC = n - ln(\frac{MSE}{n}) + 2p \tag{2.6}$$

$$SBC = n - ln(\frac{MSE}{n}) + p - ln(n) \tag{2.7}$$

3. Porovnání jednotlivých algoritmů

Nyní, když máme nástroje, jak jednotlivé predikce ohodnotit, můžeme jednotlivé metody navzájem porovnat. Obecně lze říci, že pro různá data jsou optimální různé modely. Pokud se chceme rozhodnout, který model je pro nás tím vhodným, velkým pomocníkem nám může být [7], které nabízí porovnání většiny běžně používaných metod.

Method	Average errors				Rank across all methods			
	sMAPE(%)	MdRAE	MASE	AR	sMAPE(%)	MdRAE	MASE	AR
Theta	14.89	0.88	1.13	17.8	2	3	1	2
Illies	**15.18**	**0.84**	**1.25**	**18.4**	**3**	**2**	**11**	**4**
ForecastPro	15.44	0.89	1.17	18.2	4	4	3	3
DES	15.90	0.94	1.17	18.9	5	14	3	6
Comb S-H-D	15.93	0.09	1.21	18.8	6	5	7	5
Autobox	15.95	0.93	1.18	19.2	7	11	5	7
Flores	**16.31**	**0.93**	**1.20**	**19.3**	**8**	**11**	**6**	**8**
SES	16.42	0.96	1.21	19.6	9	16	7	12
Chen	**16.55**	**0.94**	**1.34**	**19.5**	**11**	**14**	**18**	**9**
D'yakonov	**16.57**	**0.91**	**1.26**	**20.0**	**12**	7	**12**	**15**
AANN	**16.81**	**0.91**	**1.21**	**19.5**	**13**	7	7	**9**
Kamel	**16.92**	**0.90**	**1.28**	**19.6**	**14**	**5**	**13**	**12**

NNs are defined with bold numbers.

https://doi.org/10.1371/journal.pone.0194889.t002

Obrázek 4 - Porovnání různých metod pro predikce - [7]

Následující tabulka ukazuje, jak se chyba predikované hodnoty může vyvíjet s počtem predikovaných kroků.

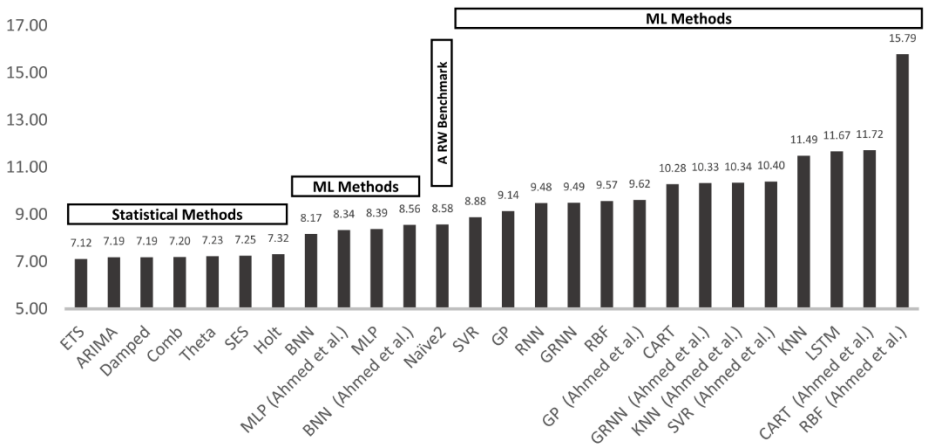
Method	Forecasting horizon										Average of forecasting horizons					
	1	2	3	4	5	6	8	12	15	18	1–4	1–6	1–8	1–12	1–15	1–18
Theta	8.4	9.6	11.3	12.5	13.2	14.0	12.0	13.2	16.2	18.2	10.4	11.5	11.6	12.0	12.4	13.0
Damped	8.8	10.0	12.0	13.5	13.7	14.3	12.5	13.9	17.5	18.9	11.1	12.0	12.1	12.4	13.0	13.6
Box-Jenkins	9.2	10.4	12.2	13.9	14.0	14.8	13.0	14.1	17.8	19.3	11.4	12.4	12.5	12.8	13.4	14.0
AANN	**9.0**	**10.4**	**11.8**	**13.8**	**13.8**	**15.5**	**13.4**	**14.6**	**17.3**	**19.6**	**11.2**	**12.4**	**12.6**	**13.0**	**13.5**	**14.1**
Single	9.5	10.6	12.7	14.1	14.3	15.0	13.3	14.5	18.3	19.4	11.7	12.7	12.8	13.1	13.7	14.4
Holt	9.0	10.4	12.8	14.5	15.1	15.8	13.9	14.8	18.8	20.2	11.7	12.9	13.1	13.4	14.0	14.6
Naive 2	10.5	11.3	13.6	15.1	15.1	15.9	14.5	16.0	19.3	20.7	12.6	13.6	13.8	14.2	14.8	15.5

NNs are defined with bold numbers.

https://doi.org/10.1371/journal.pone.0194889.t001

Obrázek 5 - Nárůst chyby podle počtu kroků - [7]

Poslední tabulka ukazuje výsledky v grafu, na kterém je hezky vidět porovnání statistických a ML metod.



Obrázek 6 - Porovnání jednotlivých metod v grafu - [7]

Z obrázku vyplývá, že statistické metody jasně dominují. Je však potřeba upozornit, že velmi záleží, na jakém modelu testy probíhaly. ML má své nezastupitelné uplatnění a to především tam, kde ostatní metody selhávají. ML může být nejlepším řešením právě tam, kde máme nekompletní, nebo "divoká", nelineární data, tam kde se sezónnost, či trend nahodile mění. Relativnost porovnávání jednotlivých metod může potvrdit například [8], kde jako nejlepší model vyšel SVR, nebo srovnání [9], kde dosáhly nejlepších výsledků genetické algoritmy, nebo také [10], kde vyhrál RBF. Ze získaných poznatků lze vyvodit, kdy volit machine learningové metody a kdy naopak spoléhat na statistiku.

3.1. Analýza časových řad

Používá se například ve statistice, ekonomii, meteorologii, či ve zpracování signálů.

Jaké parametry jsou pro nás důležité a které můžeme analyzovat?

- Trend – Rostoucí, klesající, bez trendu
- Sezónnost – Hledání periodicky opakujících se vzorů
- Vzory – Vzory, které se však opakují neperiodicky

Můžeme rozdělit na analýzu v časové a na analýzu ve frekvenční doméně.

V časové doméně můžeme použít například autokorelaci k hledání určitých opakujících se vzorů. Ve frekvenční zase spektrální analýzu, která zase může poskytnou informace o periodickém chování.

Modely můžeme podle toho, jakým způsobem data predikují, rozdělit na jednokrokové modely a modely vícekrokové. Jednokrokové modely predikují vždy jen nejbližší hodnotu a poté se stejný postup opakuje na nová data, proto se jí někdy říká rekurzivní. Vícekrokové metody se někdy nazývají jako přímé. Existují i hybridní modely nazývané *DirRec*, které oba přístupy kombinují.

Mezi velmi používané modely patří například *nearest-neighbor*. Dalším, optimalizovanějším modelem je *lazy learning*, viz např. [11]. Pracuje na principu Rozděl a panuj. Rozděluje nelineární a komplexní modely na snáz analyzovatelné lineární části. Postup algoritmu podle [12] je takový, že setřídíme vstupní vektory dle euklidovské vzdálenosti, poté určíme ideální počet sousedů a pro dané okno vypočítáme pomocí lineárního modelu predikovanou hodnotu. Model vyžaduje volbu několika vstupních parametrů. Počet sousedů, jádro funkce a vzdálenostní metriku. Metodou, jak optimalizovat volbu počtu sousedů může být *Leave-one-out* cross validace, jejíž principy můžeme uplatnit i v mnoha jiných ML metodách.

3.1.1. AR, ARIMA a ARCH

Nejznámější a nejpoužívanější metodou pro analýzu časových řad je metoda Box-Jenkinsova, jejíž kroky můžeme nazvat identifikace, odhad parametrů, ověřování modelu. Metoda zahrnuje principy autoregresivního modelu (AR) a plovoucího průměru (MA). Kombinací dostaneme ARMA model (z anglického termínu autoregressive moving average). Jeho další variantou je ARIMA (i ve významu integrovaný). Obecnějším modelem je potom ARFIMA model (zde ve významu Fractionally integrated, tedy částečně integrovaný). Mezi další modely patří například nelineární ARCH. Akronym znamená autoregressive conditional heteroskedasticity. Heteroskedasticita znamená to, že rozptyl, není konstantní a je závislý na nějakém parametru. ARCH můžeme zjednodušeně popsat jako autoregresivní model nad rozptylem. Více o ARCH např. na [13]. Opět existuje několik variant daného modelu – GARCH, TARCH, EGARCH. V poslední době také nabírají na popularitě waveletové metody.

AR, MA a ARCH se tedy mohou doplňovat. Nejdříve popíšeme autoregresi. Použijeme-li terminologie z [14], tak autoregresivní funkci řádu p můžeme definovat takto:

$$y_t = \mathrm{b}_1 \mathrm{y}_{\mathrm{t}-1} + \mathrm{b}_1 \mathrm{y}_{\mathrm{t}-2} + \cdots + \mathrm{b}_\mathrm{p} \mathrm{y}_{\mathrm{t}-\mathrm{p}} + \varepsilon_t \tag{3.1}$$

y je hodnota, *b* jsou parametry modelu a ε je bílý šum.

Plovoucí průměr MA je podle [14]:

$$y_t = \varepsilon_t + w_1 \varepsilon_{t-1} + w_2 \varepsilon_{t-2} + \cdots + w_q \varepsilon_{t-q} \tag{3.2}$$

kde w jsou parametry modelu a ε je bílý šum.

3.2. Regresní modely

Na regresi můžeme pohlížet jako na odhadování hodnoty náhodné veličiny. Můžeme buď prokládat body křivkou a nebo rozdělit stejně jako v případě data na vektor vstupů a výstupů tím způsobem, že budeme každou hodnotu výstupní modelovat jako obvykle lineární kombinaci n hodnot předchozích. S drobnou úpravou pochází následující vzorec z [15].

$$y_i = \beta_0 + \beta_1 x_{i1} + \cdots + \beta_p \mathrm{x}_{ip} + \varepsilon_i = \boldsymbol{X}_i^T \boldsymbol{\beta} + \varepsilon_i \tag{3.3}$$

Vše co rovnici vidíme je identické, jako vztahy popsané v kapitole o AR. Regresní funkce však mohou nabývat mnoha různých podob. Jedním ze způsobů je tzv. regularizace, čili omezení některých parametrů, čímž můžeme dosáhnout například toho, že ty parametry, které nemají příliš dobrý dopad na minimalizaci chybové funkce budou při výpočtu ignorovány. Existuje mnoho variant mimo jiné ridge regrese, lasso regrese, elastic net regrese a mnoho dalších.

3.2.1. Ridge regrese

Tento typ regrese svým způsobem penalizuje hodnoty parametrů. Právě míru penalizace určuje parametr λ, jehož formulace lze vidět v následujících rovnicích, které pocházejí z těchto materiálů [16], kde je problematika podrobně vysvětlena.

$$L_{ridge}(\hat{\beta}) = \sum_{i=1}^{n}(y_i - x_i^T\hat{\beta})^2 + \lambda\sum_{j=1}^{m}\hat{\beta}_j^2 = ||\boldsymbol{y} - \boldsymbol{X}\widehat{\boldsymbol{\beta}}||^2 + \lambda||\widehat{\boldsymbol{\beta}}||^2 \quad (3.4)$$

$$\hat{\beta}_{ridge} = (\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I})^{-1}(\boldsymbol{X}^T\boldsymbol{Y}) \quad (3.5)$$

3.3. Machine learning a Neuronové sítě

Cílem této práce není obecný popis neuronových sítí, přesto je však nutné uvést základní použitou terminologii. To, jak jsou jednotlivé neurony, popřípadě jejich vrstvy organizované a navzájem propojené budeme nazývat topologií. Síť můžeme dále rozlišit na cyklickou a acyklickou (dopřednou). Daná cyklicita může být různých úrovní. Výstup neuronu může být sám svým vstupem, nebo například výstup neuronu může být vstupem do všech ostatních neuronů. To, jak jsou jednotlivé vstupy a výstupy provázány můžeme nazvat stavovým prostorem sítě. Dalším rozdělením může být rozdělení na synchronní a asynchronní modely podle toho, zda je aktualizace hodnot řízena centrálně, či ne. Funkční strukturou nazvu výběr metod, algoritmů a hodnot, které budou ovlivňovat správnost výsledné sítě.

Shrneme-li tyto poznatky, budou pro nás klíčové informace o

- Organizační struktuře (topologické)
- Vstupní inicializaci hodnot
- Funkční struktuře (volba modelu neuronu, výpočetního algoritmu, volba parametrů…)

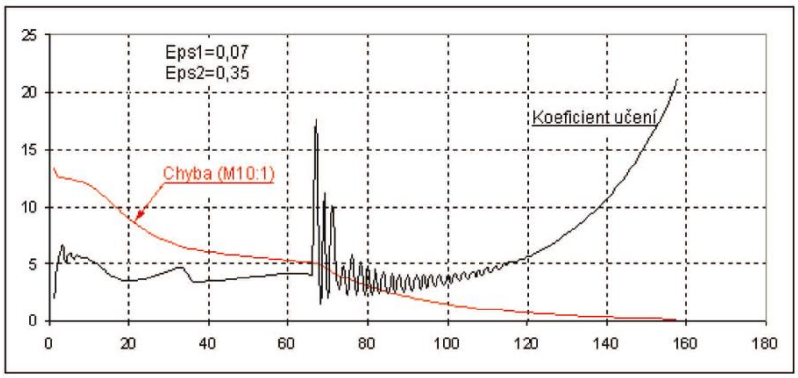
Tento souhrn parametrů budu nazývat architekturou dané sítě.

Jedním z nejdůležitějších prvků ML je rodina algoritmů backpropagation. Mějme nějakou kriteriální funkci, která nese informaci a správnosti modelu. Dále máme váhy tras mezi neurony, hodnoty aktivací jednotlivých neuronů a jejich bias. Chceme-li změnit váhu tak, aby se zmenšila chyba předpovědi od reality, pomocí derivace chyby přes váhu zjistíme, jak jedna veličina ovlivňuje druhou. Jelikož můžeme na síť pohlížet jako na řadu vnořených funkcí, můžeme použít řetězové derivační pravidlo, anglicky chain rule. Derivace složené funkce je derivací vnější funkce, krát derivace vnitřní funkce. Tímto způsobem můžeme postupovat od poslední vrstvy vždy až k neuronu, jehož optimální váhu chceme znát. Protože postupujeme od výsledné vrstvy, nese rodina algoritmů tento název. Gradient descent (dále někdy jen GD) je potom metoda, v které se snažíme minimalizovat chybu vynásobením všech gradientů získaných pomocí zpětné

propagace zápornou konstantou někdy nazývanou koeficientem učení sítě. Výstupní neuron následně počítáme pomocí propagace dopředné.

3.3.1. Učící krok a jeho optimalizace

Tento koeficient se někdy značí jako λ a může mít v průběhu výpočtu konstantní hodnotu, nebo se může s průběhem měnit. Jak se může λ měnit je popsáno například v české publikaci [17]. Vliv na výslednou hodnotu můžeme vidět na obrázku z dané publikace, kde je výpočet lambdy jiný pro počáteční (pro klidný a stabilní průběh) a jiný pro finální výpočet výstupu neuronové sítě (dynamičtější, vhodný pro snížení počtu nutných kroků).

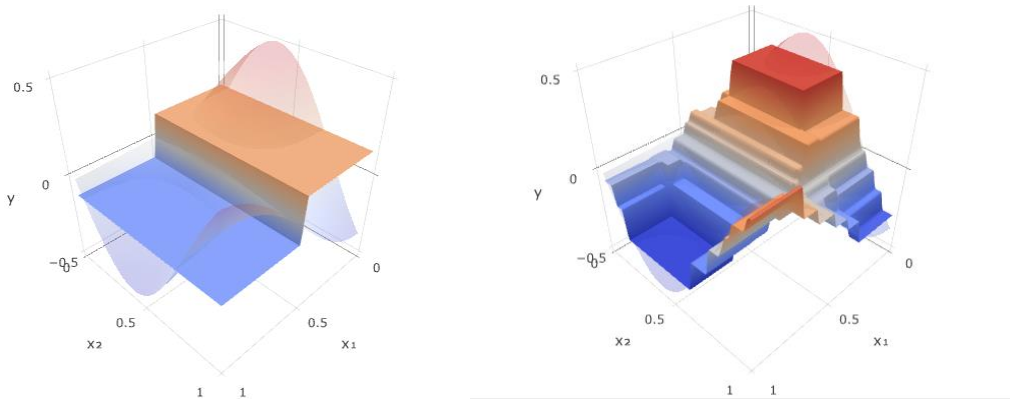


Obrázek 7 - Rozdílný výpočet λ na procesu učení neuronové sítě modelující rozhodování o poskytnutí úvěru [17]

3.3.1.1. Rmprop, adagrad, adam

Gradient descent nemusí být vždy optimálním řešením. Jeho vylepšená verze nese název stochastický gradient descent, který neustále mění svůj krok a tím se rychleji dostává k cíli a snáze překonává lokální minima. Nabývá několika podob. Jednou z nich je Adagrad, kde učící krok je optimalizován pro každý krok zvlášť. Novější podobou Adagradu je RMSProp. Mezi další přístupy patří použití tzv. Momentu, nebo Nesterova akcelerovaného gradientu.

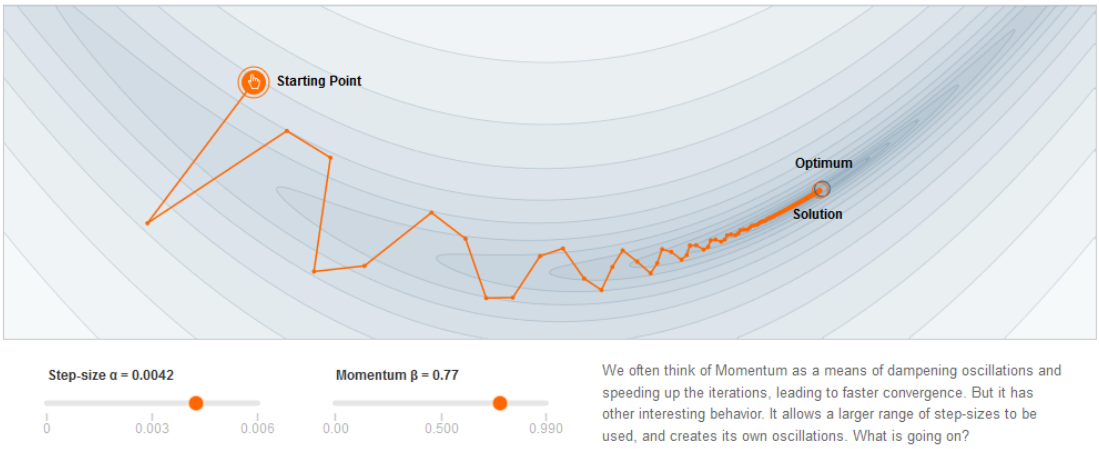
Současný trend v této oblasti je tzv. gradient boosting. Pokračuje dál ve vývoji výše uvedených metod a to různými způsoby. Hojně používanou metodou je použití rozhodovacích stromů, které jsou původně známy z klasifikačních problémů. To jak můžeme používat kombinaci rozhodovacích stromů k regresi a jaký vliv má právě počet takových vytvořených stromů je vidět na následujícím obrázku pocházejícím z interaktivní aplikace [18], kde lze nalézt i finální funkce vytvořené ze 100 takových rozhodovacích stromů.



Obrázek 8 – Vliv počtu rozhodovacích stromů (vlevo 2, vpravo 6) na správnost regrese - [18]

Popíšu podrobnější postup metody RMSprop. Nejdříve rprop, z které metoda vychází. Zjistíme si znaménka posledních dvou gradientů. Pokud jsou shodné, znamená to, že jsme se nedostali přes ideální řešení a tím pádem můžeme zvětšit učící krok. Pokud jsou rozdílná, znamená to logicky, že krok naopak musíme zmenšit. RMSprop potom řeší batchové učení, kdy nelze spočítat pouhé jedno znaménko. Uchováváme proto informaci o klouzavém průměru kvadrátu gradientů přes jednotlivé váhy, kterým následně dělíme krok.

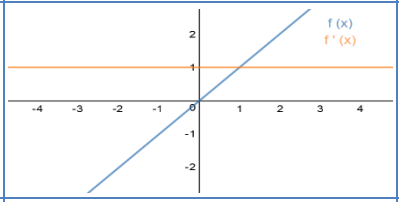
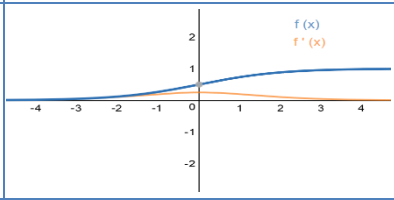
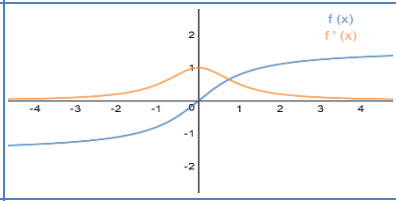
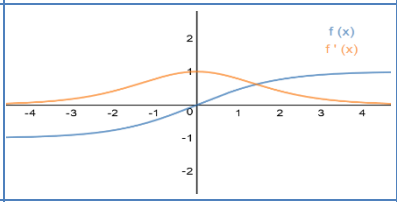
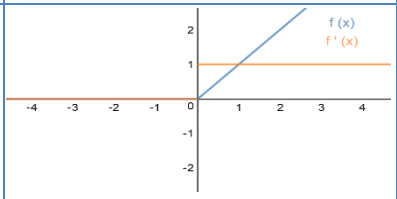
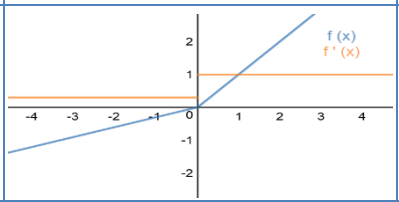
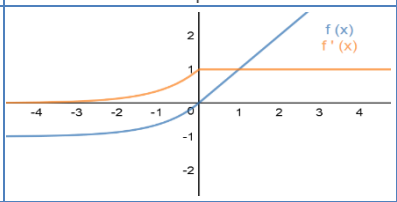
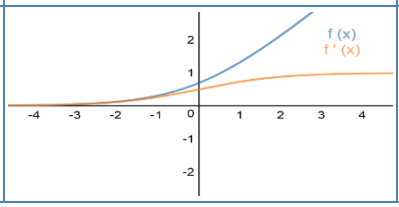
Jaký vliv může mít volba učícího kroku na potřebný počet iterací, konkrétně například u metody používající moment, je nádherně a obsáhle znázorněno ve webové aplikaci [19], kde lze vidět počet nutných kroků v závislosti na volitelných parametrech, které si uživatel může zvolit, viz následující obrázek. Na závěr uvedu metodu Adam (Adaptive Moment Estimation), která často vychází nejlépe. Pěkné shrnutí mnoha dalších různých optimalizačních principů je na [20].



Obrázek 9 – Vliv volby kroku a momentu na průběh výpočtu modelu – [19]

3.3.2. Aktivační funkce

Aktivační, nebo taky somatická funkce, je zobrazením funkce synaptické do funkce, která bude pro daný specifický problém vhodnějším nositelem informace. Například pro klasifikaci je pro nás ideální výstupní hodnota 1, nebo 0. Hodnota výstupního neuronu tedy musí transformovat vstupy pomocí vah do této škály. Hodnota mezi těmito krajními hodnotami nám poté přímo udává pravděpodobnost výsledného zařazení. Jednou z takových nejběžnějších aktivačních funkcí je sigmoida. V případu regrese nemusí mít výstupní vrstva aktivační funkci žádnou, u vrstev skrytých, se zase může ukázat jako výhodnější aktivační funkce pohybující se od -1 do 1. Je to logické, pokud se totiž pokusíme dát procesu učení neuronové sítě nějaký význam, tak zatímco u klasifikace nám vysoká hodnota aktivace některého prvku může naznačovat výskyt prvku a zařazení do konkrétní třídy, u regrese můžeme na výsledné hodnoty nahlížet jako funkci vstupní proměnné. Změna určitého vstupu tedy může mít za následek nejen růst, ale i pokles našeho výstupu. Mezi další používané aktivační funkce patří sigmoida, tanh, arctan, usměrněná funkce (rectifier), nebo její hladká podoba Softplus. Další, zdaleka ne však poslední funkcí je Softmax. Více o těchto aktivačních funkcích například na [21] a [22] odkud pochází následující definice v tabulce. Grafy byly vytvořeny v aplikaci Desmos.

Název	Graf	Definice	Derivace
Binární krok		$f(x) = 0 \quad x \in (-\infty, 0)$ $f(x) = 1 \quad x \in (-\infty, 0)$	$f'(x) = 0$
Identická aktivační funkce		$f(x) = x$	$f'(x) = 1$
Sigmoida		$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x) \cdot (1 - f(x))$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2 + 1}$
TanH		$f(x) = \frac{2}{1 + e^{-x}} - 1$	$f'(x) = 1 - f(x)^2$
ReLU (Rectifier)		$f(x) = 0 \quad x \in (-\infty, 0)$ $f(x) = x \quad x \in (-\infty, 0)$	$f'(x) = 0 \quad x \in (-\infty, 0)$ $f'(x) = 1 \quad x \in (-\infty, 0)$
Leaky ReLU		$f(x) = mx \;\; x \in (-\infty, 0)$ $f(x) = x \quad x \in (-\infty, 0)$	$f'(x) = 0 \quad x \in (-\infty, 0)$ $f'(x) = 1 \quad x \in (-\infty, 0)$
ELU (Exponential Linear Unit)		$f(x) = n(e^x - 1), x \in (-\infty, 0)$ $f(x) = x \quad x \in (-\infty, 0)$	$f'(x) = f(x) + nx \in (-\infty, 0)$ $f'(x) = 1 \quad x \in (-\infty, 0)$
Softplus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1 + e^{-x}}$

Poněkud stranou stojí Softmax [21].

$$f(x)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \qquad \sum_{j=1}^{J} f(x)_j = 1 \tag{3.6}$$

Kde *j* je index výstupu neuronu a *z* je vektor vstupů. Ve zkratce aplikujeme exponenciální funkci na každý vstup a normalizujeme ho součtem všech těchto exponenciel.

Zatímco sigmoida a softmax se hodí spíše pro klasifikační funkce, pro regresi se více využívá TanH, nebo ReLU. Obzvláště u hlubokých a rekurentních sítí může mít ReLU výhodu v tom, že je více odolná vůči tzv. mizejícímu gradientu o čem se lze více dočíst na [23].

3.3.3. Krokové vs. Dávkové zpracování

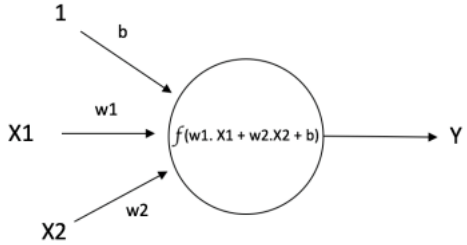
Některé metody mohou predikovat hodnoty krok po kroku, některé fungují dávkově. Z algoritmů, které zpracovávají data dávkově uvedu LB (Levenberg–Marquardt) algoritmus, již zmíněný RProp (který může vykazovat větší stabilitu), popřípadě konjugovaný gradient (který je zase velmi rychlý). Existuje také metoda nesoucí název mini-dávkové zpracování, které se nyní těší asi největší oblibě.

3.3.4. Použité modely

Následuje přehled několika metod, které budou použity. Zde jen teroretický základ a uvedení podrobnější literatury. Implementace a optimalizace bude rozebrána v praktické části, kde budou ukázány výsledky nad konkrétními reálnými daty.

3.3.4.1. Autoregresivní lineární neuronová jednotka – LNU

Lineární proto, že výstup je lineární kombinací vstupů. Autoregresivní proto, že vstupem do neuron jsou jeho předchozí stavy. Tělo neuronu je popsáno na následujícím obrázku.



Obrázek 10 - Lineární neuron - [24]

Autokorelační funkci můžeme zapsat jako:

$$\hat{y}(k) = w_0 + w_1 \cdot x_1(k) + w_2 \cdot x_2(k) \tag{3.7}$$

kde w jsou váhy neuronu, *x* jsou jeho vstupy. Neuron lze zapsat ve vektorové podobě.

$$\hat{y}(k) = [w_0 \;\; w_1 \;\; w_2] \cdot x \tag{3.8}$$

Zvolená chybová funkce je tedy:

$$e = (\hat{y} - y) \tag{3.9}$$

Kriteriální funkcí může být polovina druhé mocnina naší chyby, takže derivací naší kriteriální funkce, je funkce chybová. Postupujeme v řetězovém pravidlu, až k jednotlivé váze, přičemž využijeme toho, že derivace výstupu neuronu přes váhu pro lineární případ je jeho vstup.

$$\frac{\partial \text{chyba}}{\partial w} = (y - \hat{y}) \cdot \frac{\partial (y - \hat{y})}{\partial w} = e \cdot \boldsymbol{x} \tag{3.10}$$

Hodnota váhy v následujícím kroku je v tomto případě:

$$w_{n+1} = w_n + e \cdot x \cdot \mu \tag{3.11}$$

3.3.4.2. Metoda nejprudšího pádu

K porozumění použitým metodám je nutné napsat vice o kvadratických formách a jejich minimalizaci. Mějme kvadratickou formu:

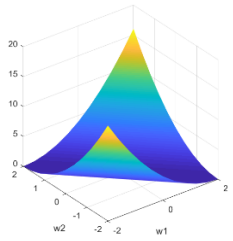
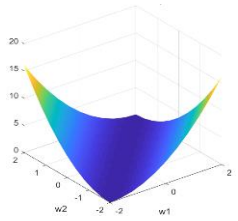
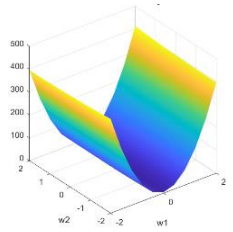
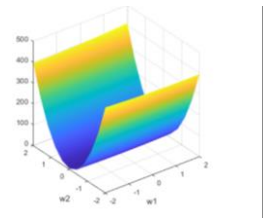
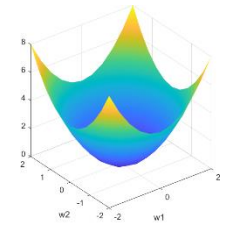
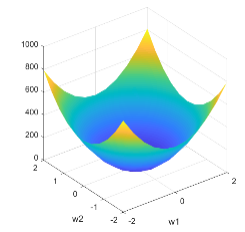
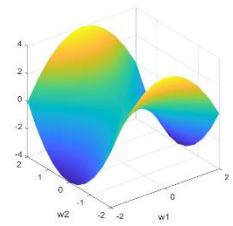
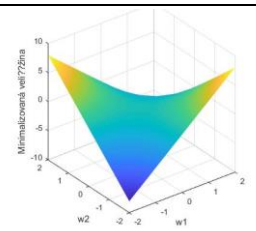
$$\varphi(x) = \frac{1}{2} x^T A x - x^T b \tag{3.12}$$

Ve funkci jedné proměnné minimalizujeme křivku tak, že položíme derivaci funkce rovnou nule a zjistíme, zda je druhá derivace kladná. Analogicky lze použít obdobný postup obecněji i pro vektorový popis. Hledáme-li minimum, můžeme tedy použít následující vztah.

$$Ax = b \tag{3.13}$$

Můžeme použít poznatků, že minimalizujeme-li funkci mající tvar kvadratické formy, musí být matice A symetrická a pozitivně definitní. Tento poznatek nám zaručuje, že funkce má lokální minimum. Kvadratickou formou pro nás může být například kritérium nejmenších čtverců, které se snažíme minimalizovat. V případě, že nelze nalézt minimum pomocí nulového gradientu explicitně, iterativní postup může být tím nejpohodlnějším řešením.

Pochopení kvadratických forem, je pro optimalizaci stěžejní, proto jsem se rozhodl jim věnovat více prostoru. Podle návodu [25] jsem vymodeloval několik kvadratických funkcí.

A	Graf	A	Graf
$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$		$\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}$	
$\begin{bmatrix} 100 & 1 \\ 1 & 1 \end{bmatrix}$		$\begin{bmatrix} 1 & 1 \\ 1 & 100 \end{bmatrix}$	
$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$		$\begin{bmatrix} 100 & 1 \\ 1 & 100 \end{bmatrix}$	
$\begin{bmatrix} -1 & -1 \\ 1 & 1 \end{bmatrix}$		$\begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix}$	

Metoda nejprudšího spádu má analogický postup s GD. Hledáme směr a hodnotu kroku, kterými se funkce bude iterativně blížit minimu. Jelikož jsou tyto hodnoty jiné než u GD, je nutné zvolit i jinou terminologii, viz [26].

$$x_{k+1} = x_k + \alpha_k r_k \tag{3.14}$$

Směr, kterým se z předchozího bodu vydáme, je rovný zápornému gradientu v daném místě a značíme ho r. Krok značíme α. Můžeme si pomoci úvahou, že směřujeme podél gradientu do té doby, dokud hodnotící funkce klesá. Hodnotu x_{k+1} dosadíme do původní kvadratické funkce za původní x. Nyní máme minimalizovanou funkci jako funkci jedné proměnné alfa. Položíme funkci rovnou nule, čímž opět hledáme minimum dané kvadratické funkce.

$$\frac{df(\alpha_k)}{d\alpha_k} = r_k^T A x_k + \alpha_k r_k^T A r_k - r_k^T b = 0 \tag{3.15}$$

Z toho lze již snadno vyvodit hodnotu optimálního kroku, která je jádrem tzv. steepest gradient descent metody. Oba vztahy opět podle [26].

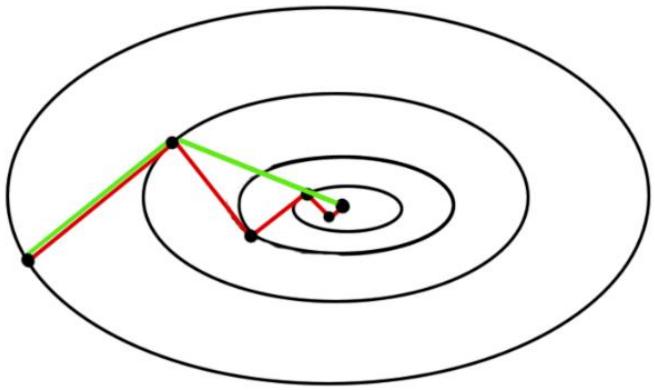
$$\alpha_k = \frac{r_k^T r_k}{r_k^T A r_k} \tag{3.16}$$

3.3.4.3. Sdružené gradienty

Nejprve co to znamená sdružený, neboli konjugovaný vektor k matici A.

$$u^T A v = 0 \tag{3.17}$$

Jedná se o vektory splňující výše uvedené podmínky zajišťující ortogonalitu vektorů. Namísto konstantního učícího kroku hledáme opět krok optimální, kromě toho ale měníme i směr. Jaký může být rozdíl mezi Gradient descent algoritmem a metodou konjugovaného gradientu (dále jen CG, z anglické terminologie), lze vidět na následujícím obrázku.



Obrázek 11 - Porovnání algoritmu Gradient descent s optimalizovaným krokem a metody konjugovaného gradient [27]

Zatímco v předchozí metodě optimalizujeme hodnotu učícího kroku α, v této metodě vybíráme i směr s. Podrobné odvození lze nalézt na [26].

$$\alpha_k = \frac{r_k^T r_k}{s_k^T A s_k} \qquad \beta_k = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k} \tag{3.18}$$

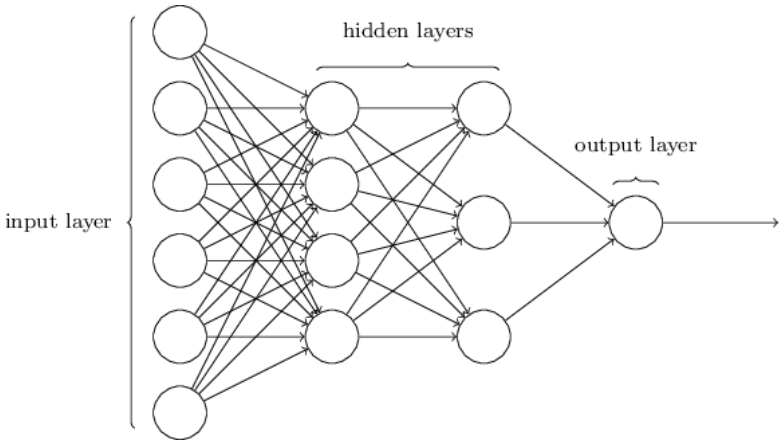
$$r_{k+1} = r_k - \alpha_k A s_k \qquad s_{k+1} = r_{k+1} + \beta_k s_k \tag{3.19}$$

$$x_{k+1} = x_k + \alpha_k s_k \tag{3.20}$$

3.3.5. Modely importované z knihoven

3.3.5.1. Multi layer perceptron

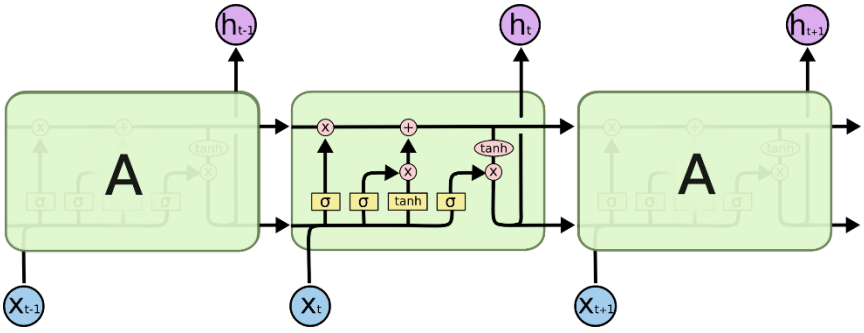
Pokud lineární neuronovou jednotku řadíme ve vrstvách za sebe, vznikne nám vícevrstvá síť. Výsledek záleží na počtu neuronů, počtu vrstev, celkové architektuře. Algoritmus výpočtu je gradient descent využívající k výpočtu vah zpětnou propagaci. Obojí bylo již výše popsáno. To jak může taková síť vypadat je dobře vidět na následujícím obrázku a jedná se o poměrně používané vyobrazení neuronových sítí.



Obrázek 12 – Multilayer perceptron - [28]

3.3.5.2. LSTM

Zkratka nese význam long short term memory model. Model využívá tzv. bran, které mají schopnost uchovávat, ale také vymazat určitou informaci buňky, což může být u rekurentních sítí obsahujících dlouhé sekvence velmi užitečné. LSTM buňka může vypadat takto. Obrázek pochází z [29], kde je také velmi pěkně zpracované vysvětlení toho, jak tyto sítě fungují.



4. Jazyk, prostředí a knihovny vhodné pro predikci

Co se jazyku týče, tak si můžeme vybrat mezi Pythonem a R. Zatímco R dominuje na poli datové analytiky, Python může nabídnout víc na poli machine learningu. Praktická část bude psána výhradně v Pythonu především kvůli jeho univerzálnosti a srozumitelnosti. Práce bude psána v IDE VS Code, jelikož nabízí nepřeberné množství funkcí, které mohou značně urychlit a zpříjemnit práci. Namátkou například Jupyter notebook pro rychlé testování kódu i vizualizaci, linting nebo deployování SQL příkazů rovnou na server.

4.1. Zen of Python

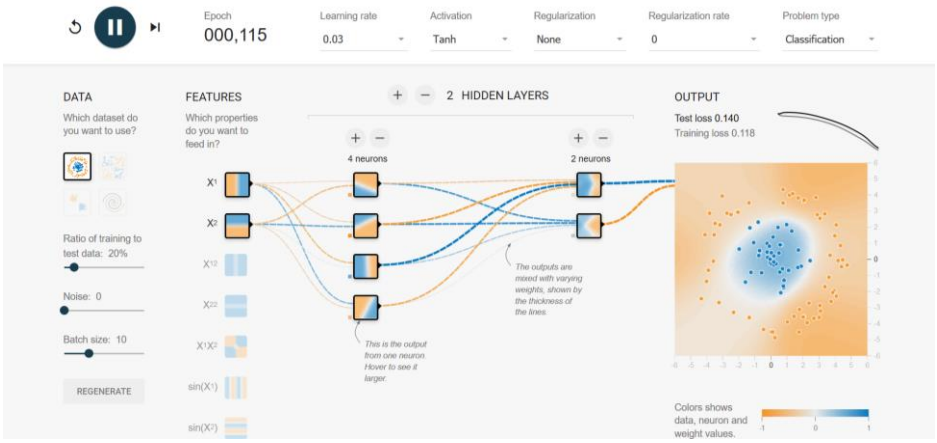
Python je relativně pomalý nekompilovaný jazyk bez oficiální podpory. Přesto python zaujal výhradní postavení v programátorské komunitě. Kdokoliv může přispět svým dílem a existuje ohromná komunita lidí, kterou bez jakýchkoliv pochybností nazývám subkulturou, která vám pomůže z nápadu udělat fungující knihovnu. Je to pravděpodobně atmosférou a celkovým pojetím jazyka, které je daleko uvolněnější než u jazyků konkurujících. Python není komerční projekt, je od lidí pro lidi a podle toho je i zpracován. Nenabízí výkon, ale nabízí srozumitelnost. Nenabízí zákaznický servis, ale poskytuje zástupy nadšenců/dobrovolníků. Podíváme-li se, kolik nám python nabízí modulů, dostaneme se téměř ke 200 000. Jednotlivých souborů jsou dokonce 2 000 000. Jednotlivé moduly potřebují ke svému životu moduly ostatní a jedná se tak o jakousi ohromnou síť vzájemných vztahů.

V době, kdy výpočetní výkon stále roste a kdy můžeme naše modely trénovat na grafické kartě či na cloudu, je pro nás srozumitelnost, rychlost a pohodlnost při práci tím nejdůležitějším. Python se člověk učí tím, že programuje, a to jde velice brzy. Python může být i pracovní styl, či přídavné jméno. Jednoduché shrnutí toho co python je, lze vykreslit pomocí příkazu *import this*, který vykreslí tzv. Zen of python, hlavní filozofii tohoto jazyka. Jedná se o 19 jednoduchých aforismů. Neznám jiný jazyk, který si může toto dovolit a nejen proto volím python jako jazyk pro tuto práci.

4.2. Knihovny

Z knihoven se kromě numpy a pandas nelze nezmínit o scikit-learn, která obsahuje nespočet již vytvořených modelů, jejichž implementace má vždy podobný syntax, čímž umožní použití velkého počtu modelů rychle a snadno. Další vhodnou knihovnou je statsmodels, která obsahuje celou sekci přímo pro analýzu časových řad, která obsahuje mimo jiné ARIMA, či ETS modely a jejich různé variance. Na práci s daty v SQL lze použít knihovnu Revoscalepy, která obsahuje též nepřeberné množství modelů pro predikce. Pro vykreslování dat, lze použít matplotlib,

seaborn, nebo pro interaktivní grafy knihovnu bokeh, či plotly. Mezi nejpoužívanější knihovny pro machine learning patří Keras, PyTorch, Caffe, nebo Tensorflow, v němž je také udělána názorná ukázka toho, jaký vliv může mít architektura sítě na konečný výsledek klasifikace, viz následující obrázek.



Obrázek 13 - Vliv vrstev a počet neuronů na výsledek – [30]

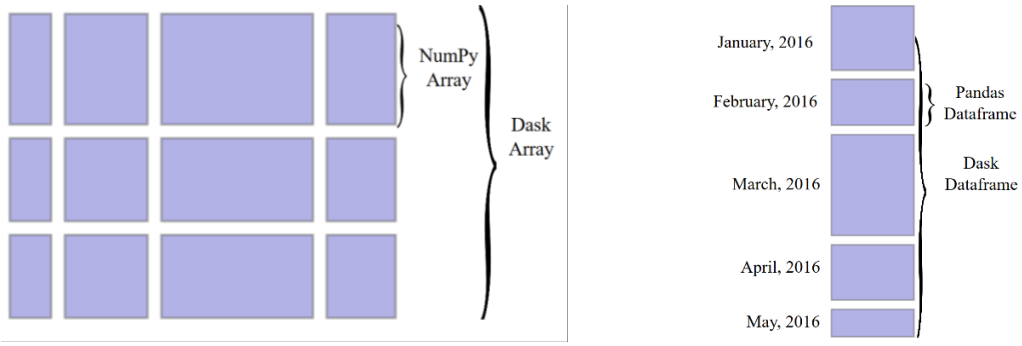
Výčet by mohl pokračovat mnohem déle a zahrnovat například knihovny XGBoost, či Light GBM obsahující metody pro tzv. gradient boosting. Využívány jsou i různé další knihovny které nejsou zaměřené přímo na predikce, ale usnadňují a zpříjemňují práci jako například PyODB a SQLAlchemy pro páci s SQL daty, pathlib pro práci s adresami nebo například prettytable pro vyobrazení výsledků.

4.3. Optimalizace výkonu - Numba, Dask, TPU

Pro optimalizaci výkonu se dá použít knihovna numba, která pomocí pouhého jednoho dekorátoru zrychlí výpočet až na úroveň jazyků jako je C či Fortran (dle slov uvedených na oficiálním webu). Jedná se o JIT (just in time compiler), který podstatnou část kódu částečně kompiluje (převádí do strojového kódu), čímž šetří čas pythonímu interpretru. Nutno podotknout, že kompilace probíhá tak, aby výpočet mohl využívat Cuda softwarovou architekturu od firmy Nvidia a náročné výpočty provádět na grafické kartě. Zatímco pro některý kód může mít knihovna špatný dopad, pro náš software je odhadován značný nárůst výkonu. Nárůst výkonu lze očekávat především u datového formátu knihovny numpy a u operací lineární algebry, kterých je v praktické části doslova nespočet. Více o knihovně numba se lze dovědět na jejich webu [31], nebo na github repozitáři [32] obsahujícím jak tutoriály, tak příklady využití. Aplikace je jednoduchá. Po importu knihovny před zvolenou funkci umístíme následující dekorátor.

```
@numba.njit()
def func():  # atd...
```

Další možností jak zlepšit výkon a to především u velkých datasetů, je použití knihovny Dask [33]. Dask slouží pro paralelizaci výpočtů. Paralelizace a threading byly pro pythonovou komunitu dlouho obskurním tématem. Takzvaný GIL (Global Interpreter Lock) jako nástroj pro správu paměti umožňuje kontrolu pythonního interpretru pouze jednomu vláknu. Důsledkem je to, že i když napíšu proces, který se má vykonat na 4 jádrech například pomocí knihovny threading, výpočet stejně nebude rychlejší než na jádru jednom. Dask by měl umět obejít omezení způsobené GIL a umožnit paralelismus. Výhodou je, že je knihovna implementována do známých a hojně používaných datových formátů jako je numpy *nd.array*, či pandas *dataframe*. Přestože rychlost výpočtu u numpy array pomocí Dasku nezlepšíme, obrovskou výhodou je možnost výpočtu nad daty, které by jinak paměť nezvládla. Knihovna umí spolupracovat také s knihovnou sklearn a můžeme tak importovat a natrénovat opravdu velké datasety.



Obrázek 14 - Vliv vrstev a počet neuronů na výsledek – [33]

Formáty je velmi snadné použít, zachovávají syntax původních formátů.

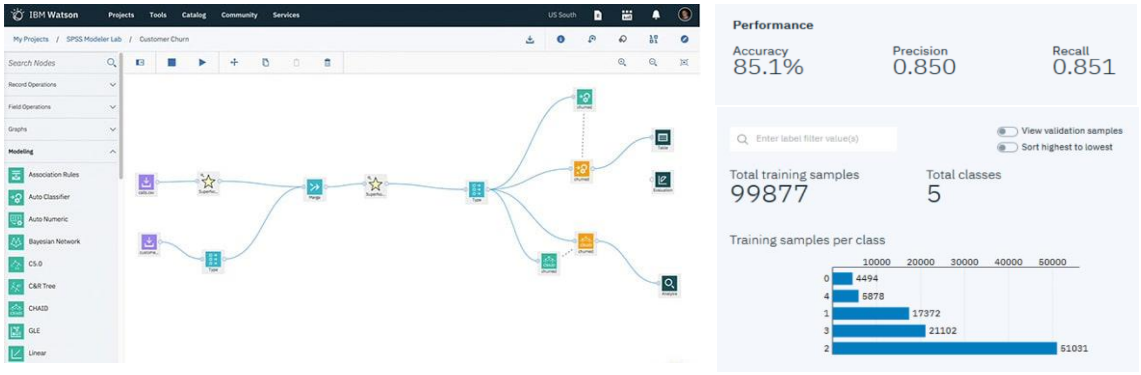
```
# Arrays implement the Numpy API
import dask.array as da
x = da.random.random(size=(10000, 10000),
                     chunks=(1000, 1000))
x + x.T - x.mean(axis=0)

# Dataframes implement the Pandas API
import dask.dataframe as dd
df = dd.read_csv('csv.csv')
df.groupby(df.account_id).balance.sum()

# Dask-ML implements the Scikit-Learn API
from dask_ml.linear_model import LogisticRegression
lr = LogisticRegression()
lr.fit(train, test)
```

Obzvláště výpočetně náročné modely jako LSTM je ideální počítat na GPU. Další možností je použít cloudové distribuované řešení (ideálně také na GPU) a jako další zajímavou možností se jeví použití tzv. TPU (tensor processor unit), procesorové jednotky navržené právě pro výpočty

machine learningových algoritmů. Jak zadarmo využít jednoho takového zdroje na google colab se lze dočíst na [34].

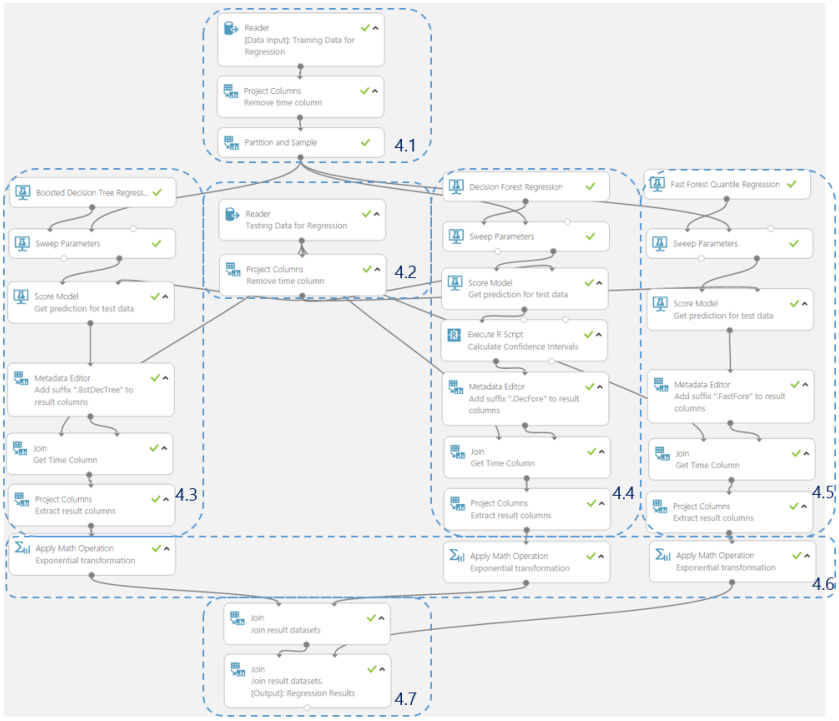
5. Rapid prototyping

Machine learning je již úspěšně implementován v mnoha různých oborech a přestože co problém, to jiné optimální řešení, podstatná část práce zůstává u většiny algoritmů společná. Vzniklo mnoho nástrojů, které se snaží vývojářům ulehčit práci a šetřit čas. Jedním z nástrojů, který se snaží práci zautomatizovat je Watson od IBM.



Obrázek 15 - Ukázka prostředí Watson [35]

Dalším prostředím je AzureML od Microsoftu.



Obrázek 16 – Ukázka grafického rozhraní AzureML [36]

Method	MAE	MPE	MAPE	MASE	RMSE
BstDecTree	89.70721	1.577143	22.281254	1.538018	143.404917
DecFore	79.996899	-0.50047	21.588716	1.440935	125.592576
FastFore	85.78435	-0.171968	20.97652	1.447306	143.623429
snaive	97.537255	-5.255961	24.762828	1.649059	158.943538
STL_ETS	83.359802	0.0456	21.73414	1.458174	134.234719
STLF_ARIMA	73.802817	-4.140493	19.824139	1.283583	122.836633

Obrázek 17 - Porovnání několika metod pomocí různých kritérií [36]

Tento výčet není zdaleka konečný, mezi další rozhraní s podobným účelem, můžeme zařadit H_2O.ai, Alpine data, Databricks, či Dataiku. Všechna uvedená prostředí mají společné velmi hezké grafické rozhraní, za kterým se však vždy skrývá velmi sofistikovaný kód. Namísto napsání stovek řádků kódu mnohdy stačí přetažení pár obrázkových bloků, které mohou mít za úkol například volbu výpočetního algoritmu, velikost trénovací množiny a typ křížové validace, nebo volbu hodnotícího kritéria.

Pro tuto práci jsem však žádné takové prostředí nevyužil. Mnohem přínosnějším pro mne bude udělat si modely co pokud možno nejvíc sám a to úplně od začátku. Přestože to stojí především zpočátku mnoho sil i energie, člověk není nakonec omezován vnějšími hranicemi programu a může si se svým výtvorem doslova hrát aniž by byl čímkoliv svazován.

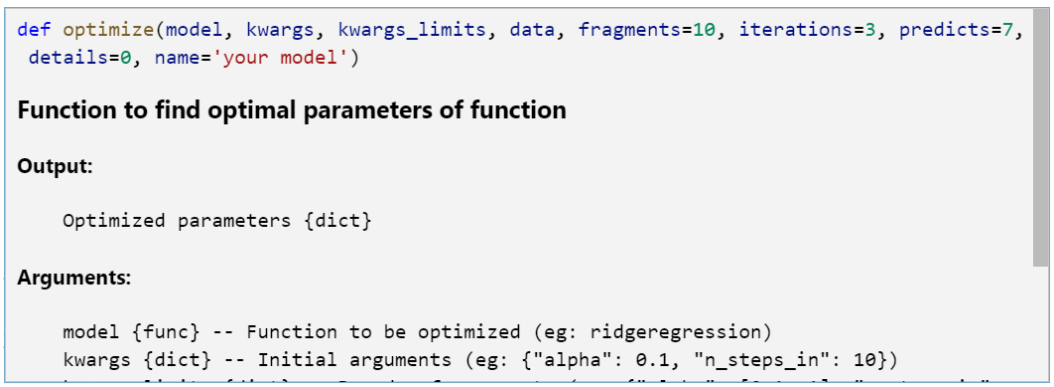
Praktická část

„NEJLEPŠÍ CESTA K PŘEDPOVĚDI BUDOUCNOSTI JE JEJÍ TVORBA.“ P. Drucker

Praktická část popisuje fungování celého systému od importu, analýzy a předpřípravy dat, přes implementování několika modelů popsaných v teoretické části, jejich ohodnocení, optimalizaci a vykreslení výsledků, popřípadě jejich zapsání do databáze. Praktická část obsahuje podstatné části kódu, celý kód je poté součástí přílohy. V kódu samotném je ke každé funkci psána dokumentace pomocí tzv. docstrings. Jelikož je kód součástí přílohy, dokumentace citována nebude, jelikož by značně prodlužovala citace a jelikož podstatné části budou vysvětleny přímo v práci. Dokumentaci lze formátovat podle několika stylů, například podle stylů reST, Numpy, Google nebo docBlockr. Následuje ukázka toho, jakým způsobem je dokumentace psána.

```
"""Function to find optimal parameters of function
======
Output:
------
    Optimized parameters {dict}

Arguments:
------
    model {func} -- Function to be optimized (eg: ridgeregression)
    kwargs {dict} -- Initial arguments (eg: {"alpha": 0.1, "n_steps_in":
                     10})
    kwargs_limits {dict} -- Bounds of arguments (eg: {"alpha": [0.1, 1],
                     "n_steps_in":[2, 30]})
    data {list, array, dataframe col} -- Data on which function is
                     optimized (eg: data1)
    fragments {int} -- Number of optimized intervals (default: 10)
    predicts {int} -- Number of predicted values (default: 7)
"""
```

Při najetí na funkci importovanou jinde poté můžeme v IDE vidět následující tooltip.



Obrázek 18 – Ukázka tooltipu vytvořené dokumentace

Formát psaní kódu je pokud možno v co největším souladu s doporučeními vydávanými pod názvem PEP – Python enhancement proposals, především podle dokumentu PEP 8.

Software funguje ve dvou režimech. V režimu běžném a v debug módu. Debug mód je pro testování modelů, ladění, hledání nejlepších výsledků a parametrů. Výstupem jsou velmi podrobná data ve formě tabulek a grafů včetně všech odchytnutých errorů. Jako testovací data si lze vložit I data ze sledovaného procesu z různých časových období. Druhým módem je mód pracovní, kdy predikujeme reálná data z nějakého procesu. Výstupem nejsou ani grafy ani podrobné tabulky, ale zápis do databáze. Grafické zpracování je součástí této práce, ale nikoliv tohoto softwaru. Finální vizualizace probíhá externě v softwaru Power BI.

Nejlepším zdrojem při psaní kódu byla dokumentace používaných knihoven obvykle přímo ta v kódu. Nejlepší literaturou pro následující řádky je dokumentace ke knihovně numpy [37], dokumentace knihovny pandas - [38] a také dokumentace pythonu samotného [39].

6. Struktura softwaru

Strukturu softwaru osvětlí nejlépe následující diagram. Ke každému ze souborů bude následovat konkrétnější popis.

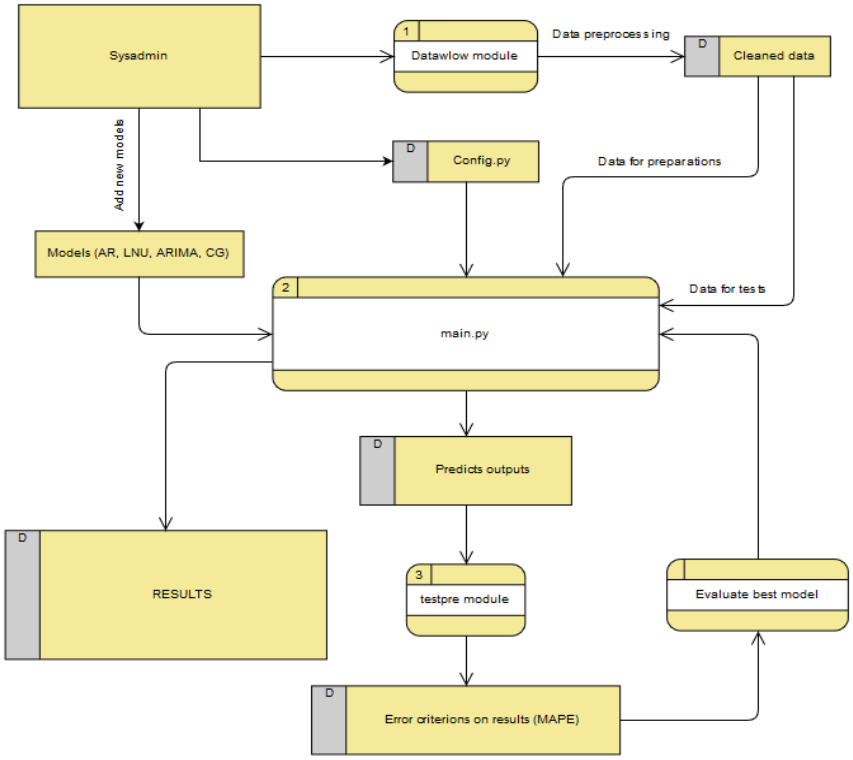
Struktura celého softwaru

```
- - Predikce
- - - - database
- - - - models
- - - - testdata
- - - - - - test1.csv
- - - - - - test2.txt
- - - - analyze.py
- - - - best_params.py
- - - - config.py
- - - - datatest.py
- - - - __init__.py
- - - - main.py
- - - - testpre
```

Struktura modulu models

```
- - models
- - - - __init__.py
- - - - ar
- - - - arima
- - - - autoreg_LNU
- - - - bayes_ridge_regression
- - - - bidirectional_lstm
- - - - congrad
- - - - elm
- - - - honu
- - - - lasso_regression
- - - - lstm
- - - - ridge_regression
- - - - sklearn_universal
```

Dataflow diagram potom může vypadat takto.



Obrázek 19 – Dataflow diagram – Vytvořeno v Visual paradigm

6.1. Requirements

Jak je obvyklé, tak requirements.txt obsahuje výpis všech potřebných knihoven včetně verzí, v kterých můžeme zaručit, že program zaručeně funguje. To může zrychlit nasazení u zákazníka.

```
arch==4.8.1
cufflinks==0.16
Keras==2.2.4
lightgbm==2.2.3
matplotlib==3.0.3
numpy==1.16.3
pandas==0.24.2
plotly==3.10.0
pmdarima==1.2.1
prettytable==0.7.2
pyodbc==4.0.26
scikit_learn==0.21.3
scipy==1.2.1
seaborn==0.9.0
sklearn_extensions==0.0.2
SQLAlchemy==1.3.6
statsmodels==0.9.0
tensorflow==1.13.1
tpot==0.10.2
xgboost==0.90
```

Chceme-li nainstalovat všechny knihovny najednou, postačí příkaz

```
pip install -r /path/to/requirements.txt
```

6.2. Config

Veškerá nastavení jsou dostupná v tomto souboru. Do jiného než tohoto souboru nedoporučuji uživatelům příliš zasahovat. Tento soubor je naopak velmi čitelný a i nezkušený uživatel se v něm rychle orientuje. Můžeme zde nastavit cestu ke skriptu (pokud neotvíráme celou složku přímo v IDE), můžeme zde nastavit, jaké všechny modely lze k predikci použít, jejich výchozí parametry a meze parametrů, mezi kterými budeme parametry optimalizovat. Dále lze nastavit počet predikovaných hodnot, délku dat z které budeme vycházet nebo zda-li budou data zapicklována a načítána z disku (lze uvažovat pouze pro testovací data). Důležité je správně nastavit vstupní data. Data je možné vkládat ve formátu CSV, či z databáze ve formátu SQL. U CSV stačí správně nastavit cestu k souboru a název predikovaného sloupce. U SQL je potřeba správně nastavit nejen název serveru a predikovaného sloupce, ale v případě, že bude použita jiná databáze, je potřeba definovat správný příkaz select, který vybere a agreguje potřebná data. Daný select se jako jediný nenastavuje v configu, ale v souboru database. Dalšími volitelnými parametry může být to, zda chceme data předem normalizovat, jestli ořezat outliery, zda chceme optimalizovat parametry a popřípadě časový limit a hloubku dané optimalizace, jestli budeme výpočetně náročné modely načítat z počítače, nebo zda je přetrénujem. Lze také nastavit jaké chceme detaily výsledků a jestli chceme vykreslit grafy. Většina parametrů lze nastavit snadno přiřazením 1 nebo 0 k proměnné daného názvu. Názvy jsou buď jednoznačné, nebo obsahují vysvětlující komentáře. Následuje ukázka toho, jak soubor vypadá.

```
server = '.'
database = 'Fk  '  # Název databáze
index_col = 'DateBK'  # Název sloupce s datumem
freqs = ['M', 'D', 'H']

csv_adress = r'C:\VSCODE\Diplomka\test_data\daily-minimum-temperatures.csv'  #
                            Adresa csv včetně názvu a přípony

cwd = r'C:\VSCOODE\Diplomka'  # CWD - absolutní adresa modulu predikcí

predicts = 7  # Počet predikovaných hodnot - defaultně 7
predicted_columns_names = ['SumNumber', 'SumDuration']  # Název sloupce jehož
                            hodnota má být predikována

evaluate_test_data = 1  # Jestli bude model hodnocen podle testovacích dat a
                            nebo pouze predikovaných dat
optimizeit = 0  # Najde optimální parametry modelů
optimizeit_details = 2  # 1 vypíše nejlepší parametry modelu, 2 vypíše každé
                            zlepšení a parametry
repeatit = 5  # repeatit je počet opakování z důvodu validace výsledků
compareit = 5  # S kolika modely bude výsledek srovnáván
confidence = 0.6  # Oblast nejistoty ve finálním grafu - vyšší hodnota znamená
                            užší oblast - maximum 1
```

```
remove_outliers = 1  # Odstraní hodnoty odlehlé od průměru. Hodnota uvádí
                           limit, nad který budou data smazána - jde o
                           násobek standardní směrodatné odchylky
criterion = 'mape'  # 'mape' or 'rmse'


plot = 1  # Pokud 1, vykreslí grafy výsledků nejlepší predikce
plotallmodels = 1  # Vykreslí všechny predikce všech modelů
piclkeit = 0  # Uloží testovací data na disk v serializované formě, čímž
                           zrychlí načítání - nutno vypnout na nulu, aby se
                           data nenačítala pokaždé
already_trained = 0  # Výpočetně náročné modely jako LSTM načíst z disku
saveit = 0
analyzeit = 0  # Analyzuje vstupní data - vypočíta autokorelaci
debug = 1  # Debug - vypíše podrobné výsledky všech predikcí
standardizeit = 0  # Standardizuje data od -1 do 1
normalizeit = 0  # Normalizuje data na směrodatnou odchylku 1 a průměr 0
```

Dále jsou definována testovací data nad kterými můžeme model hodnotit. Z důvodu častého použití byla data serializována a uložena na disk pomocí knihovny pickle.

```
dataallpickle = {
                    "Daily minimum temperatures": 'data0',
                    "Sin": 'data1',
                    "Sign": 'data2',
                    "Dynamic system": 'data3',
                    "Reálná data klapky": 'data4'
                }
```

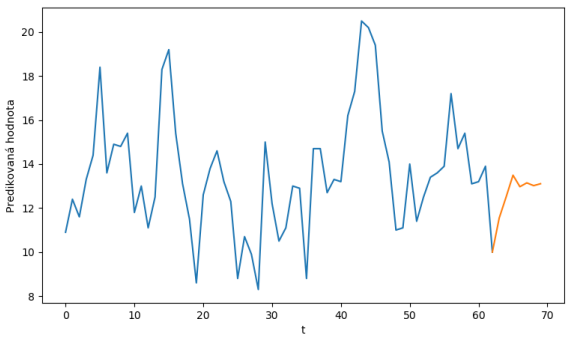
Důležité jsou 3 slovníky. První definuje které všechny modely chceme použít (ty které nechceme jednoduše zakomentujeme). Druhý slovník definuje inicializační parametry a třetí meze pro optimalizaci. Ukázka začátku všech tří slovníků je na následujících řádkách.

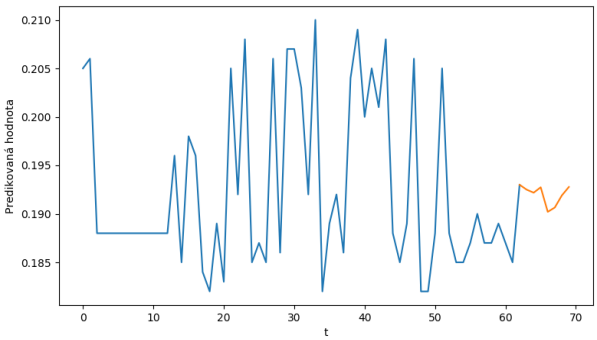
```
used_models = {

            "AR (Autoregression)": ar,
            "ARMA": arma,
            "ARIMA (Autoregression integrated moving average)": ar,
            "SARIMAX (Seasonal ARIMA)": sarima,

            "Autoregressive Linear neural unit": autoreg_LNU,
            "Linear neural unit with weigths predict": autoreg_LNU_withwpred,
            "Conjugate gradient": cg,

                                    . . .

                                      }

models_parameters = {

        "AR (Autoregression)": {"predicts": predicts, "plot": 0, 'method':
                           'cmle', 'ic': None, 'trend': 'c', 'solver':
                           'lbfgs'},
        "ARMA": {"predicts": predicts, "plot": 0, 'method': 'cmle', 'ic':
                           None, 'trend': 'c', 'solver': 'lbfgs'},
        "ARIMA (Autoregression integrated moving average)": {"predicts":
                           predicts, "plot": 0, 'method': 'cmle', 'ic':
                           None, 'trend': 'c', 'solver': 'lbfgs'},
        "SARIMAX (Seasonal ARIMA)": {"predicts": predicts, "plot": 0, "p": 1,
                           "d": 1, "q": 1, "pp": 1, "dd": 1, "qq": 1,
```

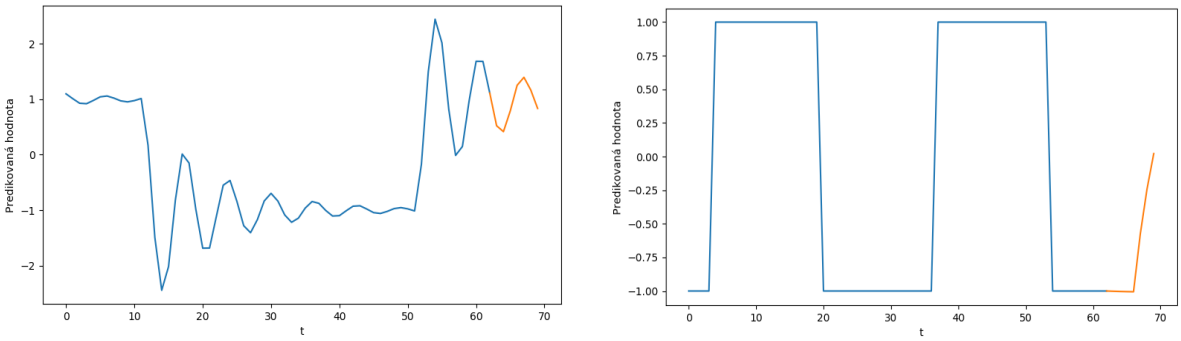
```
                                "season": 12, "method": "lbfgs",
                                "enforce_invertibility": False,
                                "enforce_stationarity": False, "verbose": 0},

                                        . . .

                                          }

models_parameters_limits = {

        "AR (Autoregression)": {"ic": ['aic', 'bic', 'hqic', 't-stat'],
                                "trend": ['c', 'nc'], "solver": ['bfgs',
                                'newton', 'nm', 'cg']},

        "Autoregressive Linear neural unit": {"lags": steps, "mi": [1.0,
                                10.0], "minormit": [0, 1], "tlumenimi": [0.0,
                                100.0]},
        "Linear neural unit with weigths predict": {"lags": steps, "mi": [1.0,
                                10.0], "minormit": [0, 1], "tlumenimi": [0.0,
                                100.0]},
        "Conjugate gradient": {"steps": steps, "epochs": epochs},
        "Extreme learning machine": {"n_steps": steps, "n_hidden": [2, 300],
                                "alpha": alpha, "rbf_width": [0.0, 10.0],
                                "activation_func": activations},
        "Batch extreme learning machine": {"n_steps_in": steps, "alpha":
                                alpha},
        "Batch Gen Extreme learning machine": {"n_steps_in": steps, "alpha":
                                alpha},

                                        . . .

                                          }
```

6.3. Test_data

Jedná se o kolekci testovacích dat. V ideálním případě by se mělo jednat o co nejrozmanitější data, která podchytí různé vlastnosti. Periodičnost, nahodilost, stacionaritu, konstantní růst a podobně. Vybral jsem dvoje reálná data – jedny z předmětu statistická mechanika z ČVUT ve formě *.txt*, druhá naměřená teplotní data ve formátu *.csv*. Dále funkci sinus a funkci signum a systém s vlastní dynamikou. Následuje ukázka testovacích dat včetně hodnot predikovaných pomocí modelu AR.







Obrázek 20 – Ukázka testovacích dat

6.3.1. Database

Python můžeme používat přímo na SQL serveru, která umí python zpracovávat. K těmto účelům slouží uložená procedura *execute_external_script*. Další možností je data pouze exportovat například v datovém typu dataframe a následně s daty z SQL serveru pracovat externě. Použity budou oba postupy. Přestože pracovat přímo v IDE je co se vývoje týče mnohem rychlejší, po uvedení softwaru do provozu bude využíván SQL job, který spustí proceduru obsahující potřebné skripty přímo na serveru. Důvodem pro tento postup je automatizace datového toku a napojení na zbytek BI řešení, které má na starosti plnění dat a podobně.

Modul obsahuje dvě funkce. *Database_load* a *database_deploy*. První jmenovaná má za úkol načíst data z databáze, převést je do datového formátu dataframe a vytvořit index v časovém formátu. Podle toho, zda chceme predikovat data na následující hodiny, dny nebo týdny, si musí funkce vytvořit pokaždé jiný select. Funkce má zhruba následující postup. Nejdříve si vytvoříme select podle toho, v jakém intervalu chceme předpovídat. Data je nutné vždy agregovat. Select vytvoříme tak, že přes if podmínku přidáváme následující úroveň jak do příkazu select, tak i do příkazu GROUP BY a ORDER BY. Data setřídíme sestupně a pomocí příkazu TOP limitujeme velikost dat. Hodnotu délky dat z které vycházíme bereme ze souboru *config*. Nastavíme název serveru, databáze a parametry pro připojení.

```
server = 'SERVER={};'.format(server)
database = 'DATABASE={};'.format(database)
sql_params = r'DRIVER={ODBC Driver 13 for SQL Server};' + server +
                          database + 'Trusted_Connection=yes;'

sql_conn = pyodbc.connect(sql_params)
```

Následně určíme, které úrovně se mají vybrat a agregovat.

```
columns = '''   D.[DateBK],
                D.[IsoWeekYear]'''
if freq == 'M':
    columns = columns + ''',
                D.[MonthNumberOfYear]'''
```

```
if freq == 'D':
    columns = columns + ''',
                D.[MonthNumberOfYear],
                D.[DayNumberOfMonth]'''

if freq == 'H':
    columns = columns + ''',
                D.[MonthNumberOfYear],
                D.[DayNumberOfMonth],
                D.[HourOfDay]'''

columns_desc = '''   D.[DateBK] DESC,
                    D.[IsoWeekYear] DESC'''
if freq == 'M':
    columns_desc = columns_desc + ''',
                D.[MonthNumberOfYear] DESC'''

if freq == 'D':
    columns_desc = columns_desc + ''',
                D.[MonthNumberOfYear] DESC,
                D.[DayNumberOfMonth] DESC'''

if freq == 'H':
    columns_desc = columns_desc + ''',
                D.[MonthNumberOfYear] DESC,
                D.[DayNumberOfMonth] DESC,
                D.[HourOfDay] DESC'''
```

Následně vytvoříme select, z kterého budeme dále vycházet.

```
query = '''

    SELECT TOP ({})
        {col},
        sum([Number]) SumNumber,
        sum([Duration]) SumDuration

    FROM [dbo].[FactProduction] F
        INNER JOIN dbo.DimDateTime D
        ON F.DimDateTimeId = D.DimDateTimeId

    WHERE   DimScenarioId = 1
            and     DimProductionEventId = 1
            and     DimOperationOutId = 69

    GROUP BY
        {col}

    ORDER BY
        {col_desc}'''.format(data_limit,
         col=columns, col_desc=columns_desc)
```

Dalším úkolem je vytvořit dataframe a přidělit časový index. Dataframe vytvoříme pomocí následujícího příkazu.

```
df = pd.read_sql(query, sql_conn)
```

Následně vytvoříme ze sloupce s datumem index a v případě, že je tak nastaveno, tak smažeme poslední hodnotu, jelikož nemusí být kompletní.

Další funkcí je funkce *database_deploy*, která má za úkol vypočtená data zapsat do databáze, konkrétně do schéma *stage*.

```
def database_deploy(last_date, sum_number, sum_duration, freq='D'):

    lenght = len(sum_number)

    dataframe_to_sql = pd.DataFrame([])
    dataframe_to_sql['EventStart'] = pd.date_range(start=last_date,
                                periods=lenght+1, freq=freq)
    dataframe_to_sql = dataframe_to_sql.iloc[1:]

    dataframe_to_sql['DimDateId'] = dataframe_to_sql['EventStart'].dt.date
    dataframe_to_sql['DimTimeId'] = dataframe_to_sql['EventStart'].dt.time
```

Následuje definování hodnot pro všechny potřebné sloupce. Nakonec definujeme parametry pro připojení a pomocí funkce *dataframe_to_sql* data deployujeme na server.

```
params =
                                urllib.parse.quote_plus(r'DRIVER={driver};SERVER=
                                {server};DATABASE={database};Trusted_Connection=y
                                es'.format(driver=r'{SQL Server}',
                                server=config.server, database=config.database))
conn_str = 'mssql+pyodbc:///?odbc_connect={}'.format(params)

engine = create_engine(conn_str)
dataframe_to_sql.to_sql(name='FactProduction',con=engine, schema='Stage',
                                if_exists='append', index=False)
```

Pokud bychom chtěli pythonní kód spustit přímo v SQL management studiu, stačí jednou povolit externí skripty pomocí.

```
EXEC sp_configure 'external scripts enabled', 1
RECONFIGURE WITH OVERRIDE
```

Dále do T-SQL příkazu před každý pythonní skript uvést

```
EXEC sp_execute_external_script
@language = N'Python',
@script = N'
```

Druhou možností načítání dat je pomocí CSV, to je však řešeno přímo v souboru *main*, který bude popsán v poslední kapitole.

6.4. Analyze

Analyze má za úkol přinést informace o datech, která chceme predikovat. Podle zjištěných informací můžeme přizpůsobit parametry výpočtu na míru. Hlavní informací pro nás je stacionarita a nahodilost dat. Dále funkce obsahuje vykreslení korelační matice

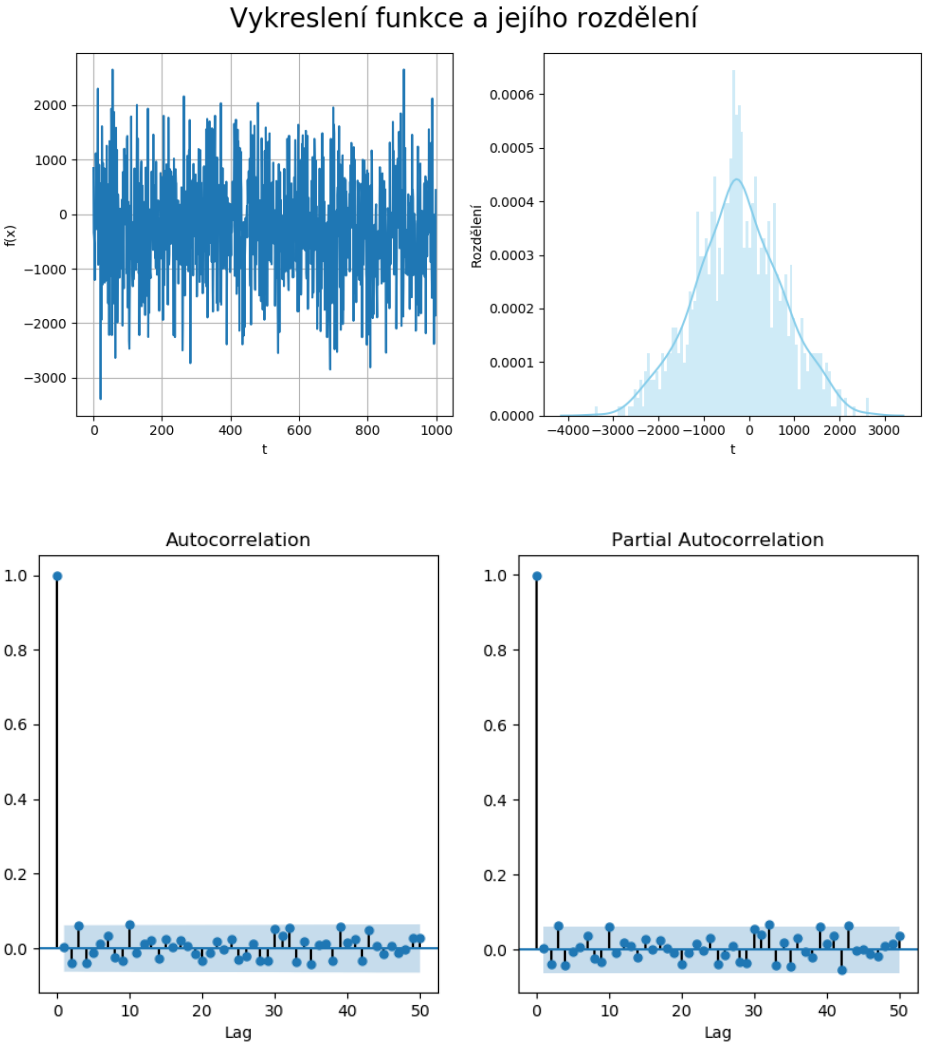
6.4.1. Vykreslení dat, jejich distribuce a jejich autokorelační funkce

Studujeme-li některá data, je potřeba je nejdřív přehledně vyobrazit. Zjistíme autokorelační (ACF) a částečnou autokorelační funkci (PACF), za účelem zjištění počtu předchozích kroků, které mi následující členy ovlivňují. PACF neuvažuje pro daný lag vliv lagů předchozích. Díky těmto krokům můžeme na první pohled říci, zda mohou fungovat autoregresivní funkce.

Účelem je tedy rychlá analýza a prvotní seznámení nejen se vstupními daty. Skript lze pro data *x* volat takto

```
from analyze import analyze
analyze(x)
```

Funkce potom pro vložená data vykreslí následující.



Funkce je velmi krátká, proto ji uvedu celou.

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

```
from statsmodels.graphics.tsaplots import plot_acf
from statsmodels.graphics.tsaplots import plot_pacf
import pandas as pd

def analyze(y, lags = 50):
    try:
        plt.figure(figsize=(10,5))
        plt.subplot(1, 2, 1)
        plt.plot(y)
        plt.xlabel('t')
        plt.ylabel("f(x)")

        plt.subplot(1, 2, 2)
        sns.distplot(y, bins=100, kde=True, color='skyblue')
        plt.xlabel('f(x)')
        plt.ylabel("Rozdělení")

        plt.tight_layout()
        plt.suptitle("Vykreslení funkce a jejího rozdělení", fontsize=20)
        plt.subplots_adjust(top=0.88)
        plt.show()

        fig, (ax, ax2) = plt.subplots(ncols=2, figsize=(10,5))
        plot_acf(y, lags=lags, ax=ax)
        ax.set_xlabel('Lag')
        plot_pacf(y, lags=lags, ax=ax2)
        ax2.set_xlabel('Lag')
        plt.show()
        if isinstance(y, pd.DataFrame):
            print(y.describe())
        if isinstance(y, (np.ndarray, list)):
            bb = pd.DataFrame(data=y)
            print(bb.describe())

    except:
        print('Wrong datatype for more stats or more lags, than values')
```

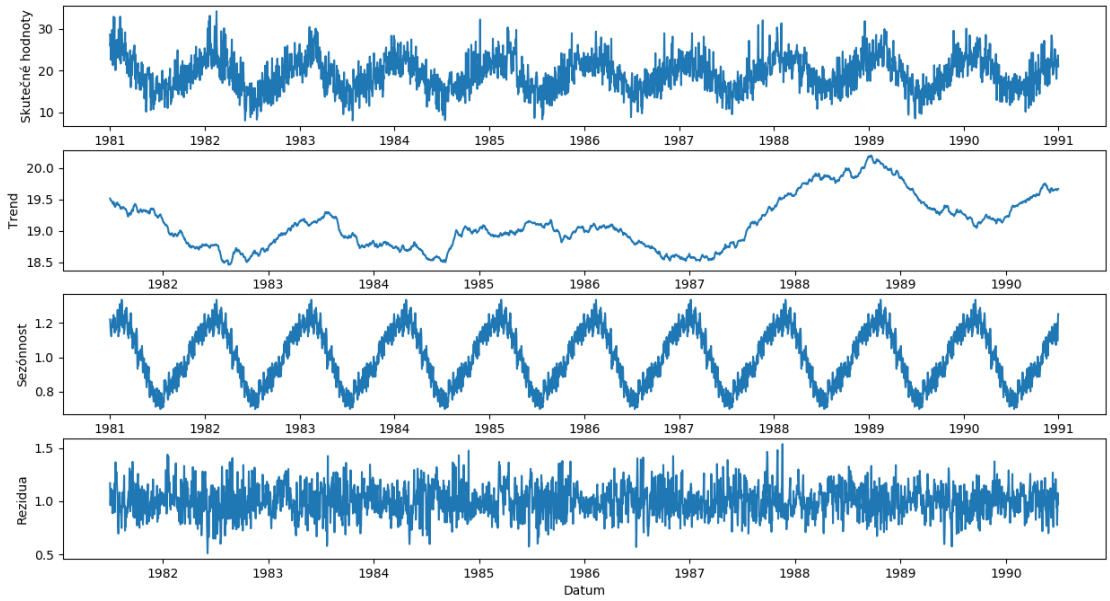
6.4.2. Decompose

Chceme-li něco predikovat, je moudré v datech hledat určité paterny, které se periodicky opakují. Pokud je najdeme, snadno je budeme moci předpovídat. Naším úkolem je tedy najít takovou frekvenci, že kombinace tzv. sezónnosti a trendu bude co nejpodobnější naší funkci původní, čili dá nejmenší sumu reziduí. Můžeme mít model trendu a sezónnosti vypočtený jak pomocí násobení, tak i pomocí sčítání.

```
def decompose(data, freq=365, model='multiplicative'):

    decomposition = seasonal_decompose(data, model=model, freq=freq)

    plt.figure(figsize=(15,8))
    plt.subplot(4, 1, 1)
    plt.plot(decomposition.observed)
    plt.xlabel('Datum')
    plt.ylabel("Skutečné hodnoty")

    plt.subplot(4, 1, 2)
    plt.plot(decomposition.trend)
    plt.xlabel('Datum')
    plt.ylabel("Trend")
```

```
plt.subplot(4, 1, 3)
plt.plot(decomposition.seasonal)
plt.xlabel('Datum')
plt.ylabel("Sezónnost")

plt.subplot(4, 1, 4)
plt.plot(decomposition.resid)
plt.xlabel('Datum')
plt.ylabel("Rezidua")

plt.suptitle("Sezónní dekompozice", fontsize=20)
plt.subplots_adjust(top=0.88)

plt.show()
```

Voláme-li funkci nad testovacími daty, konkrétně nad naměřenými teplotními daty, může výsledek vypadat takto.



Podíváme-li se na význam výše uvedeného obrázku, tak sezónnost nám určuje průměrnou teplotu v jednotlivých dnech, trend potom značí jak se teplota mění nikoliv ze dne na den, ale jak se mění meziroční průměr – patern definovaný v sezónnosti.

6.4.3. Stacionarita

Otázka stacionarity může být pro naše další rozhodnutí zásadní. To, zda jsou statistické hodnoty v čase neměnné, můžeme zjistit například pomocí rozšířeného Dicker-Fullerova testu.

```
pvalue = adfuller(data)
cutoff=0.01
if pvalue < cutoff:
    print('p-value = ' + str(pvalue) + 'Data jsou pravděpodobně
                    stacionární')
else:
    print('p-value = ' + str(pvalue) + 'Data jsou pravděpodobně
                    nestacionární')
```

```
        except Exception as excp:
            print(excp, 'Wrong datatype for more stats or more lags, than
                         values')
```

6.5. Data_prep

Skript obsahuje funkce na datový preprocessing jako například funkci *remove_outliers*, která vyřadí všechna evidentně chybná data, funkci *split*, která data rozdělí na trénovací a testovací množinu, nebo funkce *make_sequencies* a *make_x_input*, které data rozdělují na vektor vstupů a vektor výstupů.

6.5.1. Remove_outliers

Funkce ořeže data, která jsou evidentně chybná. V případě, že chceme předpovídat především odstávky, je naopak nutné tuto funkci mít vypnutou. Otázkou je, zda by tato funkce nešla řešit lépe, jelikož prostým ořezáním částečně porušujeme kauzalitu, což může mít za následek špatný výpočet sezónnosti. Ideálnější by patrně bylo data proložit. Funkce funguje postupuje podle následujících kroků: Spočte průměr, směrodatnou odchylku, následně průměr odečte a vymaže všechny hodnoty větší než několikanásobek absolutní hodnoty směrodatné odchylky.

```
def remove_outliers(data, predicted_column_index = 0, threshold = 3):
    data = np.array(data)
    data_shape = data.shape
    if len(data_shape) == 1:
        data = data[data > data.std()]

        return data

    else:
        data_mean = data[predicted_column_index].mean()
        data_std = data[predicted_column_index].std()

        counter = 0
        for i in range(len(data[predicted_column_index])):

            if abs(data[predicted_column_index, counter] - data_mean) >
                            threshold * data_std:
                data = np.delete(data, counter, axis=1)
                counter -= 1
            counter += 1

    return data
```

6.5.2. Difference

Někdy je výhodné provést nějakou transformaci. Jednou z možností je hodnoty transformovat na rozdíl sousedních hodnot. Transformace má za následek vyrušení trendu, což může znamenat lepší predikce.

```
def do_difference(dataset):
    diff = list()
    for i in range(1, len(dataset)):
```

```
        value = dataset[i] - dataset[i - 1]
        diff.append(value)
    return np.array(diff)

def inverse_difference(differenced_predictions, last_undiff_value):
    first = last_undiff_value + differenced_predictions[0]
    diff = [first]
    for i in range(1, len(differenced_predictions)):
        value = differenced_predictions[i] + diff[i - 1]
        diff.append(value)
    return np.array(diff)
```

6.5.3. Split - Rozdělení na trénovací a testovací množinu

Funkce split funguje poněkud jinak než například u problémů klasifikace. Data jsou chronologicky uspořádána a proto musíme dodržovat pořadí měřených dat. Rozdělení proto provedeme tak, že posledních *x* dat v řadě označíme jako data testovací, zbytek dat jako data trénovací. Konkrétní implementace potom vypadá takto.

```
data = np.array(data)
data_shape = data.shape

if len(data_shape) == 1:
    train = data[:(len(data)-predicts)]
    test = data[-predicts:]

    if predicts > (len(data)):
        print('To few data, train/test not returned')
        return ([np.nan], [np.nan] * predicts)

else:
    if predicts > (data_shape[1]):
        print('To few data, train/test not returned')
        return ([np.nan], np.array([np.nan] * predicts))

    train = data[:, :-predicts]

    test = data[predicted_column_index, -predicts:]

return (train, test)
```

Funkci poté můžeme zavolat obvyklým způsobem.

```
from data_prep import split
split(data)
```

6.5.4. Make_sequences – Rozdělení na vstup a výstup

Jedná se o funkci, která přidá datům další rozměr tím, že použije historická data. Uvedu-li pro srozumitelnost jednoduchý případ – list [1, 2, 3, 4]. Co je vstupem a co výstupem? U autoregresivních funkcí můžeme brát jako výstup jednotlivé hodnoty a jako vstup jejich přímé předchůdce. Pro případ jedné historické hodnoty budou výstupem z funkce následující listy [2, 1], [3, 2], [4, 3]. Konkrétní implementace vypadá takto. Potom, co definujeme funkci hlavní, definujeme si vnořenou funkci, která vytvoří sekvence pro jednorozměrná data.

```
def make_sequences(data, n_steps_in, n_steps_out=1, constant=None,
                           predicted_column_index=0,
                           other_columns_lenght=None):

    def make_seq(data, n_steps_in, n_steps_out=1, constant=None):
        X, y = list(), list()

        for i in range(len(data)):
            # find the end of this pattern
            end_ix = i + n_steps_in
            out_end_ix = end_ix + n_steps_out
            # check if we are beyond the data
            if out_end_ix > len(data):
                break
            # gather input and output parts of the pattern
            seq_x, seq_y = data[i:end_ix], data[end_ix:out_end_ix]
            X.append(seq_x)
            y.append(seq_y)

        return X, y
```

Dále postup rozdělíme podle toho, jakého rozměru data jsou. V případě vícerozměrných dat data serializujeme daným způsobem.

```
    data = np.array(data)
    data_shape = data.shape

    if len(data_shape) == 1:
        X_list, y = make_seq(data=data, n_steps_in=n_steps_in,
                           n_steps_out=n_steps_out, constant=constant)

    else:

        for_prediction = data[predicted_column_index, :]
        other_data = np.delete(data, predicted_column_index, axis=0)

        X_only, y = make_seq(for_prediction, n_steps_in=n_steps_in,
                           n_steps_out=n_steps_out)

        other_columns = []
        X_list = []

        for i in range(len(other_data)):
            other_columns_sequentions, e = make_seq(other_data[i],
                           n_steps_in=n_steps_in, n_steps_out=n_steps_out)
            other_columns.append(other_columns_sequentions)

        other_columns_array = np.array(other_columns)
```

Ostatní sloupce nemusí být stejného rozměru jako sloupec predikovaný, ale jejich délka lze nastavit. Jako poslední věc přidáme možné přidání konstanty na začátek každého vstupu.

```
            if other_columns_lenght:
                other_columns_array = other_columns_array[:, :, -
                                other_columns_lenght:]

            for i in range(len(X_only)):

                other_columns_list = list(other_columns_array[:, i, :].reshape(-
                                1))
                X_list.append(other_columns_list + list(X_only[i]))

        X = np.array(X_list)

        if constant:
            X = np.insert(X, 0, constant, axis=1)

        return np.array(X), np.array(y)
```

Funkci poté voláme takto.

```
from data_prep import make_sequences
make_sequences(x)
```

6.5.5. Make_x_input – Generování dalšího vstupu

V případě, že máme model již natrénovaný, tak potřebujeme vytvořit vstupy, z kterých model pomocí natrénovaných parametrů vytvoří výstupy – námi hledané predikce.

```
def make_x_input(data, n_steps_in, predicted_column_index=0,
                                other_columns_lenght=None, constant=None):

    data = np.array(data)
    data_shape = data.shape

    if len(data_shape) == 1:

        x_input = data[-n_steps_in:]
        if constant:
            x_input = np.insert(x_input, 0, constant)

    else:
        for_prediction = data[predicted_column_index, -n_steps_in:]
        other_data = np.delete(data, predicted_column_index, axis=0)

        if other_columns_lenght:
            other_cols = other_data[:, -other_columns_lenght:]
        else:
            other_cols = other_data[:, -n_steps_in:]

        x_input = np.concatenate((other_cols, for_prediction), axis=None)

        if constant:
            x_input = np.insert(x_input, 0, constant)

    x_input = x_input.reshape(1, -1)

    return x_input
```

6.6. Best_params

Modely jsou napsány jako funkce obsahující různé parametry. Pro různé parametry dosahují různých výsledků. U některých modelů může být správná volba parametrů doslova stěžejní. Mezi takové parametry může patřit učící krok, počet lagů pro výpočet, solver, trend nebo metoda. Funkce zajišťující optimalizaci se jmenuje optimize. Postup je následující. Zaprvé si vytvoříme vnořenou funkci *evaluate_model*, která zavolá daný model, vypočte predikce nad vstupními hodnotami a vyčíslí hodnotící kritérium. Funkci voláme pomocí bloku try/except, jelikož tak můžeme zrychlit výsledek a navíc odchytávat očekávané errory a upozornění. Jako další krok vezmeme meze daných parametrů a rozdělíme je na daný počet intervalů, následně vytvoříme kombinace parametrů každého s každým. Voláme funkci *evaluate_model*, která vyhodnotí nejlepší kombinaci parametrů. Následně vezmeme vždy nejbližší sousedy nejlepších parametrů a jako nový interval označíme právě toto rozmezí. Postupujeme stejným způsobem. Uvedu snadný příklad pro pochopení. Pokud jsou meze parametru 0 a 10, tak si onen interval rozdělíme například na 10 částí. Vyhodnotíme pro který parametr vyšla nejlépe hodnotící funkce a jako následující interval zvolíme pouze rozmezí ±1. Během pouhých pěti iterací se dostaneme na pět desetinných míst. Rozdíl mezi touto metodou a metodou bruteforce je v počtu provedených vyhodnocení, který je řádově menší. Riskujeme sice, že optimum nenalezneme, ovšem za předpokladu poměrně hladkých funkčních závislostí parametrů to můžeme snadno přejít. Pro funkci s 5 parametry s rozdělením do 5 intervalů o 5 iteracích jde o 5 x 5^5 výpočtů. Modely z tensorflow potřebují pro jeden výpočet na běžném CPU i několik minut, proto nedoporučuji optimalizaci nad těmito modely provádět pokaždé, ale pouze jednou nad danými daty a optimální parametry nastavit jako parametry inicializační.

Pokud je vstupním parametrem string, nebo nějaká třída, uvedeme list všech možností a program sám vyhodnotí, že nemá vytvářet žádný interval, ale zahrnout všechny uvedené možnosti. Další věcí, kterou je potřeba vzít v potaz je, že některé parametry musejí být celočíselné. To řešíme tak, že tyto meze zapíšeme jako integery. V opačném případě napíšeme za číslo tečku. Funkce je relativně dlouhá, proto uvedu jen některé části. Po definování knihoven definujeme funkci a její parametry.

```
def optimize(model, kwargs, kwargs_limits, data, fragments=10, iterations=3,
                            predicts=7, details=0, name='your model'):
```

Následuje dokumentace toho o čem funkce je, rozdělení parametrů na ty, které se nemění a na ty, které optimalizujeme. Vnořená funkce *evalueate_model* vypadá takto

```
def evaluate_model(kwargs):
    predictions = model(train, **constant_kwargs,  **kwargs)
    x, modeleval, z = test_pre(predictions, test, predicts=predicts, plot=0)
    return modeleval
```

Funkce je poté rovnou volána, jelikož čekáme, že inicializační parametry jsou odhadnuty s rozumem a může se jednat o parametry nejlepší. Dosaženou hodnotu si uložíme do proměnné.

Díky tomu, že voláme funkci přes try, můžeme podchytit různá upozornění podle typu a nebo i podle obsaženého slova. Můžeme tak přeskočit zbytečné pokusy, kdy některé kombinace parametrů dávají například špatně podmíněné matice. Následující ukázka odchytává chyby nejdříve podle typu erroru a následně i slova, který se v erroru nachází.

```
# Dojde-li k následujícímu warningu, výpočet kroku přerušen
warnings.filterwarnings('error', category=RuntimeWarning)
warnings.filterwarnings('error', message=r".*ambiguous*")
```

Vzhledem k tomu, že někdy může optimalizace trvat velmi dlouho, je dobré nastavit si časový limit jednoho vyhodnoceného modelu. Na [40] jsem našel vytvořenou funkci *watchout*, která plní přesně naše požadavky.

```
def watchdog(timeout, code, *args, **kwargs):
    def tracer(frame, event, arg, start=time.time()):
        now = time.time()
        if now > start + timeout:
            raise TimeoutError('Time exceeded')
        return tracer if event == "call" else None

    old_tracer = sys.gettrace()
    try:
        sys.settrace(tracer)
        result = code(*args, **kwargs)
        return result

    except TimeoutError:
        if details == 2:
            print('Time exceeded')

    finally:
        sys.settrace(old_tracer)
```

Následuje rozdělení na intervaly podle toho, jestli jde o float, integer, nebo jiný typ.

```
if not isinstance(j[0], (int, float, np.ndarray, np.generic)):
    kwargs_fragments[i] = j
elif isinstance(j[0], int):
    pomoc = np.linspace(j[0], j[1], fragments, dtype = int)
    kwargs_fragments[i] = list(set([int(round(j)) for j in pomoc]))
else:
    kwargs_fragments[i] = np.unique(np.linspace(j[0], j[1], fragments))
```

Poté vytvoříme kombinace každého s každým. Optimální knihovnou pro podobné úlohy je itertools. Celou smyčku odtud provedeme tolikrát, kolik chceme iterací. Spočteme počet všech kombinací abychom věděli v kolika procentech optimalizace se nalézáme.

```
for i in range(iterations):

    combinations = list(itertools.product(*kwargs_fragments.values()))
    combi_len = len(combinations)
    percent = round(combi_len / 100, 1)
    kombi = []
    for j in combinations:
        combination_dict = {key: value for (key, value) in
                            zip(kwargs_limits.keys(), j)}
        kombi.append(combination_dict)
```

Vyhodnotíme všechny kombinace a vypočteme hodnotící kritérium. Pokud je některé hodnotící kritérium lepší než doposud dosažené maximum.

```
for k in range(len(combinations)):
    try:
        res = watchdog(time_limit, evaluatemodel, kombi[k])
        if res < best_result:
            best_result = res
            best_params = kombi[k]
```

Dále definujeme co se má stát při daných errorech či upozorněních a nastavíme si jaké výstupy chceme vyobrazit v průběhu optimalizace. Na závěr uvedu způsob, jak zlepšit výkon. Pokud se během několika iterací výsledek nezlepší, nemá cenu pokračovat.

```
if round(memory_result, 4) == round(best_result, 4):
    return best_params
```

Nakonec již jen definujeme nové intervaly. Poslední věcí kterou nelze opomenout, je uvažování mezního případu. V případě mezní hodnoty neměníme meze do obou směrů, ale pouze tam, kde je to dovoleno. V případě nastavení *config.details* na 1 nebo 2 vypisujeme různě podrobné výstupy již během výpočtu, především v kolika procentech optimalizace se nalézáme a když narazíme na nový nejlepší výsledek a jeho parametry.

6.7. Test_pre

Vstupem do *test_pre* jsou predikované hodnoty a hodnoty testovací (skutečné). Výstupem je hodnotící kritérium MAPE nebo RMSE. Výstupem mohou být také grafy porovnávající predikci se skutečností. Vykreslení do grafů je v práci několikrát naznačeno, proto uvedu pouze výpočet kritérií.

```
error = np.array(predicted) - np.array(test)

if criterion == 'rmse':
    rmseerror = error ** 2
    rmse = (sum(rmseerror) / predicts) ** (1/2)

    return rmse
```

```
if criterion == 'mape':
    abserrormape = [abs(j / test[i]) if abs(test[i]) > 0.001 else 1 for i,j in
                            enumerate(error)]
    sumabserrormape = sum(abserrormape)
    mape = sumabserrormape / predicts

    return mape
```

7. Models

Jedná se o samostatný modul, což v případě není nic jiného, než složka obsahující soubory s příponou *.py* a souborem *__init__.py* , který pythonu dá informaci, že se jedná o modul, a vykoná všechno, co v něm je. V našem případě importujeme všechny obsažené soubory, definujeme dokumentaci o čem modul je a také definuje pár proměnných, které budou použity v optimalizaci. Nejdůležitější částí je import souborů, kterých model využívá.

```
from .autoreg_LNU_withwpred import autoreg_LNU_withwpred
from .autoreg_LNU import autoreg_LNU
from .cg import cg
```

Tímto způsobem importujeme všechny modely. Dále importujeme některé prvky, které budeme potřebovat. Především slovník aktivací, slovník chybových kritérií z knihovny *tensorflow*. Dále vytvoříme seznam všech regresorů, které jsou v knihovně *sklearn*. S pomocí návodu na stackoverflow vznikly následující řádky. Jejich využití bude popsáno v kapitole Regresní modely.

```
regressors=[]
for module in sklearn.__all__:
    try:
        module = import_module(f'sklearn.{module}')
        regressors.extend([getattr(module,cls) for cls in module.__all__  if
                            'Regress' in cls ])
    except:
        pass
regressors.append(sklearn.svm.SVR)
default_regressor = sklearn.linear_model.BayesianRidge
```

Otázkou je, zda načítat pokaždé všechny modely. Obzvláště ty z tensorflow si pro import žádají značný čas. Zatímco PEP8 doporučuje mít všechny importy přehledně nahoře, někdy může být i přes horší srozumitelnost vhodnější lazy loading a tensorflow načítat pouze v případě potřeby. Předčasná optimalizace je občas označována dokonce jako největší chyba začínajících programátorů, proto je problém ve fázi vývoje v případě potřeby řešen pouze zakomentováním daných knihoven.

Všechny použité modely jsou v modulu models. Pro lepší orientaci v příloze uvedu vedle názvů modelů i názvy souborů. Nebudu uvádět do detailu všechny modely, jelikož se několik modelů liší pouze v pár detailech. Struktura modulu se dá rozdělit do několika skupin. Vlastní neuronové

modely, modely z knihovny statsmodels jako AR a ARIMA, LSTM a multilayer perceptron z knihovny tensorflow, regresní modely z sklearn, extreme learning machine modely z sklearn extensions a jednoduchý model pro porovnání s průměrem. Jednotlivé názvy modelů jsou vidět na následujícím schématu.

```
autoreg_LNU_withwpred                    mlp
autoreg_LNU
cg                                       sklearn_universal

ar                                       regression_bayes_ridge
arma                                     regression_hubber
arima                                    regression_lasso
sarima                                   regression_linear
                                         regression_ridge
lstm_batch                               regression_ridge_CV
lstm_bidirectional
lstm_stacked_batch                       elm
lstm_stacked                             elm_gen
lstm
mlp_batch                                compare_with_average
```

To, jakým způsobem modely fungují je popsáno v teoretické části, zde bude řešena vždy již jen implementace a kód samotný. Nebudou popsány všechny modely, ale vždy modely typické pro danou knihovnu.

7.1. Autoregresivní lineární neuronová jednotka – LNU

Modul obsahuje 2 verze. Druhá se jmenuje *autoreg_LNU_withwpred*. Jak název napovídá, tak predikuje nejen budoucí hodnotu, ale i budoucí hodnotu vah. Někdy může být tento postup spíše nevýhodný, nicméně mnohdy přináší výsledek řádově lepší. Jedná se o krokovou metodu a stejně jako u většiny ostatních machine learningových krokových metod budeme mít narůstající chybu s narůstajícím počtem predikovaných hodnot. Nejvíce se proto hodí pro malý horizont predikce. Vstupními parametry funkce jsou počet regresivních hodnot a tedy i počet vstupů do neuronu, zda bude učící krok normalizován a jakým způsobem bude tlumen a jestli budou vstupní váhy zvoleny náhodně. Vlastní modely jsou zatím pouze pro jednorozměrná data. Proto nejdříve ověříme, je-li tomu tak a popřípadě vybereme sloupec, který chceme predikovat. Model je vždy vypočten pro několik hodnot učícího kroku a následně je vyhodnocen nejlepší z nich. Následuje definice prázdných schránek pro data a tzv. seed náhodných čísel, aby výsledky přestože operujeme s náhodnými čísly byly reprodukovatelné.

```
import matplotlib.pyplot as plt
import numpy as np

def autoreg_LNU(data, predicts=7, lags=100, mi=0.1, minormit=1, tlumenimi=1,
                                plot=0, random=0, seed = 0):

    miwide = np.array([mi * 10, mi, mi / 10, mi / 100, mi/1000, mi/100000,
                                mi/1000000])
    miwidelen = len(miwide)
    leng = len(data)
```

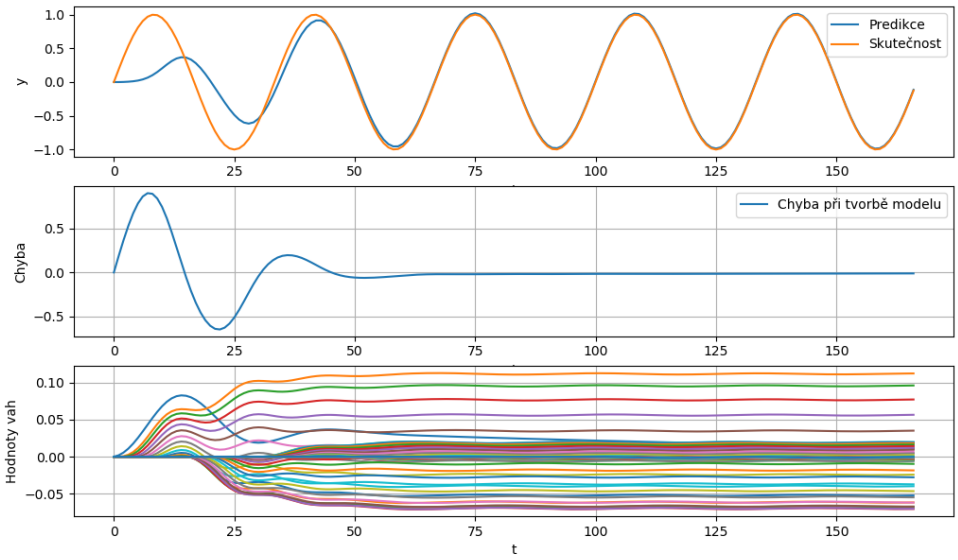
```
y = np.zeros((miwidelen, leng))
w = np.zeros((miwidelen, lags + 1))
x = np.zeros((miwidelen, lags + 1))
e = np.zeros((miwidelen, leng))
eabs = np.zeros((miwidelen, leng))
predictions = np.zeros(predicts)

if plot == 1:
    wall = np.zeros((miwidelen, leng, lags + 1))

if seed is not 0:
    random.seed(seed)
```

Následuje samotná smyčka. Pro každý z uvedených kroků je provedena dopředná propagace, výpočet reziduí a následně zpětná propagace určující hodnotu přírůstků daných vah.

```
for i in range(miwidelen):
    if random == 1:
        w[i] = np.random.rand(lags + 1)
    x[i][0] = 1

    for j in range(leng):
        y[i][j] = np.dot(w[i], x[i])
        if y[i][j] > 1000 * max(data): # Ochrana proti nekonvergujícím
                        hodnotám učícího kroku
            e[i][-1] = 1000000
            break
        e[i][j] = data[j] - y[i][j]
        x[i][2:] = x[i][1:-1]
        if j>0:
            x[i][1] = data[j - 1]
        dydw = x[i]
        if minormit == 1:
            minorm = miwide[i] / (tlumenimi + np.dot(x[i], x[i].T))
            dw = minorm * e[i][j] * dydw
        else:
            dw = miwide[i] * e[i][j] * dydw
        w[i] = w[i] + dw
        if plot == 1:
            wall[i][j][:] = w[i]

bestmi = [0] * miwidelen
for k in range(miwidelen):
    eabs[k] = [abs(i) for i in e[k]]
    bestmi[k] = sum(eabs[k][-predicts:])
bestmivalue = min(bestmi)
bestmiindex = [i for i, j in enumerate(bestmi) if j == bestmivalue][0]

for l in range(predicts):
    predictions[l] = np.dot(w[bestmiindex], x[bestmiindex])
    x[bestmiindex][2:] = x[bestmiindex][1:-1]
    if i>0:
        x[bestmiindex][1] = predictions[-1]

return (predictions)
```

V kódu není citováno vykreslení dat do grafu, abychom se mohli podívat na to, jak se model učí, jelikož je poměrně dlouhý a protože použití knihovny matplotlib bylo naznačeno v modulu *test_pre*. Výsledek může vypadat například takto.

Predikovaná vs. skutečná hodnota, chyba a váhy



Nutno podotknout že výsledek vychází velmi pěkně z toho důvodu, že se jedná o hladká testovací data. Lze vidět, že většina úseků vývoje vah má určitý trend, se kterým je dobré počítat. Tento trend můžeme predikovat pomocí neuronové sítě, což má ale značné důsledky na výpočetní čas, proto jsem zvolil mnohem rychlejší model AR. Predikce všech vah jsem vyřešil takto.

```
wwlenght = lags + 1
wwhist = np.zeros((lags + 1, leng ))
wwt = np.zeros((lags + 1, predicts))
wwhist = wall[bestmiindex].T

for i in range(lags + 1):
    wwt[i] = ar(wwhist[i], predicts = predicts)

ww = wwt.T

for j in range(predicts):
    predictions[j] = np.dot(ww[j], x[bestmiindex])
    x[bestmiindex][2:] = x[bestmiindex][1:-1]
    if i>0:
        x[bestmiindex][1] = y[bestmiindex][-1]
```

7.2. Sdružené gradienty

Nutno podotknout, že výsledky tohoto modelu vychází mnohdy nejlépe a to přestože jediný použitý modul, který jsem sám nevytvořil je numpy. Kompaktní a přesto velice výkonný model.

```
import numpy as np
from data_prep import make_sequences_with_constant
from data_prep import makesequences

def cg(data, steps = 50, predicts = 7, epochs = 100):

    X, y = makesequences(data, steps)
    w = np.zeros(steps)
```

```
    for i in range(epochs):
        b = np.dot(X.T,y)
        A = np.dot(X.T,X)
        re = b - np.dot(A,w)
        p = re.copy()

        alpha = np.dot(re.T,re)/(np.dot(np.dot(p.T,A),p))
        w = w + alpha * p
        re_prev = re.copy()
        re = re_prev - alpha * np.dot(A,p)
        beta = np.dot(re.T,re) / np.dot(re_prev.T,re_prev)
        p = re + beta * p

    predictions = []
    x_input = data[-steps:]

    for i in range(predicts):
        ypre = np.dot(w, x_input)
        predictions.append(ypre)
        x_input = np.insert(x_input, steps, ypre)
        x_input = np.delete(x_input,0)

    predictionsarray = np.array(predictions)

    return predictionsarray
```

7.3. Modely dostupné z ML knihoven

Mezi použité knihovny patří scikit-learn, která má mnoho regresních modelů, sklearn-extensions s extreme learning modely, dále statsmodels, která obsahuje AR a ARIMA modely a Tensorflow spolu s Kerasem, kde byl implementován multi layer perceptron, a LSTM model.

7.3.1. Statsmodels

ARIMA modely jsou v predikci časových řad pravděpodobně nejpoužívanějšími algoritmy. Kromě toho jsou Zpracovány i modely AR, ARMA a SARIMAX, čili sezónní ARIMA. Všechny tyto modely jsou zpracovány zatím jen pro jednorozměrná vstupní data.

7.3.1.1. AR – sm_ar

Výpočetně velmi rychlý model, který je jako pomocný používán i v několika modelech ostatních. Po importu knihoven definujeme funkci s různými parametry, které budeme moci následně optimalizovat. Následuje ověření rozměru dat. AR modely jsem použil vždy pouze nad jednorozměrnými modely. Následuje nafitování modelu a následně vyhodnocení predikcí.

```
from pandas import Series
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.ar_model import AR, _ar_predict_out_of_sample
from sklearn.metrics import mean_squared_error
import numpy as np

def ar(data, predicts=7, plot=0, predicted_column_index=0, method='cmle',
                              ic='aic', trend='nc', solver='lbfgs'):
```

```
    data = np.array(data)
    data_shape = data.shape

    try:
        if len(data_shape) == 1:
            model = AR(data)
        else:
            model = AR(data[predicted_column_index])

        model_fit = model.fit(method=method, ic=ic, trend=trend,
                              solver=solver, disp=-1)

        endogg = [i[0] for i in model.endog]
        predictions = _ar_predict_out_of_sample(endogg, model_fit.params,
                              model.k_ar, model.k_trend, steps = predicts,
                              start=0)

        if plot == 1:
            plt.plot(predictions, color='red')
            plt.show()

        predictions = np.array(predictions).reshape(-1)
        return predictions

    except Exception as err:
        print("\t", err)
        return [np.nan] * predicts
```

7.3.1.2. ARIMA – sm_arima

Nejdůležitější je na tomto modelu správně nastavit jeho řád. Řád určují 3 písmena – *p*, *d*, *q*. *P* je řád autoregrese, tedy kolik předchozích členů bude zahrnuto do regresního výpočtu. *D* znamená odčítání minulé hodnoty, takzvané diferencování. Pokud je řada stacionární, *d* má být 0. Q je řád plovoucího průměru. Velmi hezké shrnutí algoritmu je v [41]

```
from pandas import Series
import pandas as pd
import matplotlib.pyplot as plt
import statsmodels.api as sm
import numpy as np

def arima(data, predicts=7, plot=0, p=3, d=1, q=0, method='mle', ic='aic',
                              trend='nc', solver='lbfgs'):
    """Autoregressive integrated moving average model
    """
    if len(data) <= 10:
        return

    order = (p, d, q)

    model = sm.tsa.ARIMA(data, order=order)

    try:
        model_fit = model.fit(method=method, ic=ic, trend=trend,
                              solver=solver)
        predictions = model_fit.forecast(steps=predicts)

        return predictions[0]
```

7.3.2. Regresní modely – scikit-learn

Byly použity následující typy regresí

- Bayes ridge
- Hubber
- Lasso
- Linear
- Linear batch
- Ridge
- Ridge CV
- Vlastní univerzální sklearn modul

Díky tomu, že má většina modulů identický syntax, je možné vytvořit kostru a následně typ modelu vystrčit jako vstupní parametr. Využijeme dále funkce *Multioutputregressor*, která nám umožní všechny modely počítat i batchově. Díky modelu *best_params* následně najdeme pro naše data ten nejvhodnější model a zda hodnoty počítat po jedné, nebo najednou. V úvodním skriptu modulu models *__init__.py* je vytvořen list všech regresorů, které mohou následně vstupovat do modelu. Seznam vypadá takto.

```
['sklearn.ensemble.forest.RandomForestRegressor',
 'sklearn.ensemble.forest.ExtraTreesRegressor',
 'sklearn.ensemble.bagging.BaggingRegressor',
 'sklearn.ensemble.gradient_boosting.GradientBoostingRegressor',
 'sklearn.ensemble.weight_boosting.AdaBoostRegressor',
 'sklearn.ensemble.voting.VotingRegressor',
 'sklearn.gaussian_process.gpr.GaussianProcessRegressor',
 'sklearn.isotonic.IsotonicRegression',
 'sklearn.linear_model.bayes.ARDRegression',
 'sklearn.linear_model.huber.HuberRegressor',
 'sklearn.linear_model.base.LinearRegression',
 'sklearn.linear_model.logistic.LogisticRegression',
 'sklearn.linear_model.logistic.LogisticRegressionCV',
 'sklearn.linear_model.passive_aggressive.PassiveAggressiveRegressor',
 'sklearn.linear_model.stochastic_gradient.SGDRegressor',
 'sklearn.linear_model.theil_sen.TheilSenRegressor',
 'sklearn.linear_model.ransac.RANSACRegressor',
 'sklearn.multioutput.MultiOutputRegressor',
 'sklearn.multioutput.RegressorChain',
 'sklearn.neighbors.regression.KNeighborsRegressor',
 'sklearn.neighbors.regression.RadiusNeighborsRegressor',
 'sklearn.neural_network.multilayer_perceptron.MLPRegressor',
 'sklearn.tree.tree.DecisionTreeRegressor',
 'sklearn.tree.tree.ExtraTreeRegressor',
 'sklearn.compose._target.TransformedTargetRegressor',
 'sklearn.svm.classes.SVR']
```

Obdobně můžeme pouhou změnou v parseru ze slova regressor na slovo clasifier stáhnout i všechny klasifikátory. Následuje ukázka toho, jak je napsaná funkce.

```
from sklearn.linear_model import BayesianRidge
from sklearn.multioutput import MultiOutputRegressor
from importlib import import_module
import numpy as np

from data_prep import make_sequences, make_x_input

def sklearn_universal(data, n_steps_in=5, predicts=7, model=BayesianRidge,
                              predicted_column_index=0,
                              output_shape='one_step',
                              other_columns_lenght=None, constant=None):
```

Máme možnost predikovat buď batchově a nebo krokově. Aby byl výběr automatický, opět si možnost vystrčíme do parametrů, abychom mohli model automaticky optimalizovat. Použijeme funkci *make_sequences*, která nám pro daný typ výpočtu vytvoří vstupy a výstupy.

```
    data = np.array(data)
    data_shape = np.array(data).shape

    if output_shape == 'one_step':
        X, y = make_sequences(data, n_steps_in=n_steps_in,
                              predicted_column_index=predicted_column_index,
                              other_columns_lenght=other_columns_lenght,
                              constant=constant)
    if output_shape == 'batch':
        X, y = make_sequences(data, n_steps_in=n_steps_in,
                              n_steps_out=predicts,
                              predicted_column_index=predicted_column_index,
                              other_columns_lenght=other_columns_lenght,
                              constant=constant)
```

Model, který máme jako vstupní parametr následně nafitujeme. Pro případ batchového výstupu použijeme další regresor multioutputregressor.

```
    reg_model = model()
    multi_regressor = MultiOutputRegressor(reg_model)
    multi_regressor.fit(X, y)
```

Podle toho, jestli jsou data jednorozměrná, nebo vícerozměrná a také podle toho jestli počítáme krokově, a nebo batchově vytvoříme následující větve kde vypočítáme nový vstup do modelu a v případě krokového výpočtu vytvoříme smyčku o délce podle počtu požadovaných predikcí.

```
    if len(data_shape) == 1:

        if output_shape == 'one_step':

            predictions = []
            x_input = make_x_input(data, n_steps_in=n_steps_in,
                              constant=constant)

            for i in range(predicts):

                yhat = multi_regressor.predict(x_input)

                x_input = np.insert(x_input, n_steps_in, yhat[0], axis=1)
                x_input = np.delete(x_input, 0, axis=1 )
                predictions.append(yhat[0])
```

```
    if output_shape == 'batch':

        x_input = make_x_input(data, n_steps_in=n_steps_in,
                               constant=constant)

        predictions = multi_regressor.predict(x_input)
        predictions = predictions[0]
```

Podobný postup použijeme i pro případ vícerozměrných dat. Komplikace nastávají případě krokového výpočtu, jelikož již pro druhý krok nejsme schopni vytvořit požadovaný vstup, jelikož z ostatních sloupců nemáme poslední hodnotu. Operativně jsem problém vyřešil tak, že hodnoty ostatních sloupců dopočítám pomocí jednoduchého a rychlého modelu AR. Samozřejmě má tento postup výhodu pouze v některých případech a to konkrétně pokud jsou ostatní sloupce silně korelované a snadno predikovatelné narozdíl od sloupce predikovaného.

```
else:

    if not other_columns_lenght:
        other_columns_lenght = n_steps_in

    if output_shape == 'one_step':

        from models import ar

        predictions = []
        nu_data_shape = data.shape

        for i in range(predicts):

            x_input = make_x_input(data, n_steps_in=n_steps_in,
                                   predicted_column_index=predicted_column_index,
                                   other_columns_lenght=other_columns_lenght,
                                   constant=constant)

            nucolumn = []
            for_prediction = data[predicted_column_index]

            yhat = multi_regressor.predict(x_input)
            yhat_flat = yhat[0]
            predictions.append(yhat_flat)

            for_prediction = np.append(for_prediction, yhat)

            for j in data:
                new = ar(j, predicts=1)
                nucolumn.append(new)

            nucolumn_T = np.array(nucolumn).reshape(nu_data_shape[0], 1)
            data = np.append(data, nucolumn_T, axis=1)
            data[predicted_column_index] = for_prediction

    if output_shape == 'batch':

        x_input = make_x_input(data, n_steps_in=n_steps_in,
                               predicted_column_index=predicted_column_index,
                               other_columns_lenght=other_columns_lenght,
                               constant=constant)

        predictions = multi_regressor.predict(x_input)
        predictions = predictions[0]
```

```
    predictions = np.array(predictions).reshape(-1)

    return predictions
```

Modely, které vycházejí pro daná data nejlépe jsou pak řešeny zvlášť, jelikož je vždy možné optimalizovat několik dalších parametrů, viz například.

```
def regression_bayes_ridge(data, n_steps_in=50, predicts=7,
                           predicted_column_index=0,
                           output_shape='one_step',
                           other_columns_lenght=None, constant=None,
                           n_iter=100, alpha_1=1.e-6, alpha_2=1.e-6,
                           lambda_1=1.e-6, lambda_2=1.e-6):
```

7.3.3. Extreme learning machine - Sklearn-extensions

Knihovna obsahuje extreme learning machine modely. Jedná se o 2 různé modely, které jsou stejně jako u samotné sklearn téměř identické. Postup je velmi podobný modelu sklearn_universal a je tedy relativně dlouhý, proto uvedu pouze krátkou původní krokově počítanou verzi pro jednorozměrná data.

```
from sklearn_extensions.extreme_learning_machines.elm import ELMRegressor
from data_prep import makesequences
import numpy as np

def elm(data, n_steps=50, predicts=7, n_hidden=20, alpha=0.5, rbf_width=1.0,
                           activation_func='tanh'):

    X, y = makesequences(data, n_steps)

    reg = ELMRegressor(n_hidden=n_hidden, alpha=alpha, rbf_width=rbf_width,
                           activation_func='tanh')
    reg.fit(X, y)

    x_input = data[-n_steps:]
    x_input = x_input.reshape((1, n_steps))

    predictions = []
    for i in range(predicts):
        yhat = reg.predict(x_input)
        x_input = np.insert(x_input, n_steps, yhat[0], axis=1)
        x_input = np.delete(x_input,0, axis=1 )
        predictions.append(yhat[0])
    predictionsarray = np.array(predictions)

    return predictionsarray
```

7.3.4. Tensorflow, Keras

Velmi populární knihovna používaná především díky hlubokým sítím a prací s obrazem. Jsou velmi náročné na výpočet. Nutno podotknout, že natrénování jednoho takového modelu trvá déle než výpočet všech ostatních dohromady.

7.3.4.1. LSTM

Kromě obvyklých parametrů jako je volba dat a počet predikovaných hodnot jsem udělal 10 dalších parametrů, které lze zvenčí optimalizovat. Doporučuji však optimalizaci vynechávat, nebo alespoň provádět po krocích, jelikož má extrémní výpočetní nároky. Zvenčí je možné nastavit a tedy i optimalizovat nejen počet neuronů, ale i počet vrstev. Narozdíl od sklearn jsou krokové a batchové modely zvlášť stejně jako modely s jednou a více LSTM vrstvami. Ukázka základního krokového modelu s jednou LSTM vrstvou vypadá takto.

```
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense
import tensorflow as tf
import numpy as np
from data_prep import makesequences
from pathlib import Path
import os

def lstm(data, n_steps, n_features=1, predicts=7, epochs=200, units=50,
                              save=1, already_trained=0, optimizer='adam',
                              loss='mse', verbose=0, activation='relu',
                              metrics='mape', timedistributed=0):

    script_dir = os.path.dirname(__file__)
    folder = Path('stored_models')
    modelname = 'lstmvanilla.h5'
    model_path = script_dir / folder / modelname
    model_path_string = model_path.as_posix()

    X, y = makesequences(data, n_steps)
    X = X.reshape((X.shape[0], X.shape[1], n_features))
    x_input = data[-n_steps:]
    x_input = x_input.reshape((1, n_steps, 1))

    if already_trained is not 1:
        model = Sequential()
        model.add(LSTM(units, activation=activation, input_shape=(n_steps,
                              n_features)))

        if timedistributed == 1:
            model.add(TimeDistributed(Dense(1)))
        else:
            model.add(Dense(1))

        model.compile(optimizer=optimizer, loss=loss, metrics=metrics)
        model.fit(X, y, epochs=epochs, verbose=verbose)

        if save == 1:
            model.save(model_path_string)

    if already_trained == 1:
        try:
```

```
            model = tf.keras.models.load_model(model_path_string)
            model.load_weights(model_path_string)
        except:
            print("Model is not saved, first saveit = 1 in config")

    predictions = []

    for i in range(predicts):
        yhat = model.predict(x_input, verbose=verbose)
        x_input = np.insert(x_input, n_steps, yhat[0][0], axis=1)
        x_input = np.delete(x_input,0, axis=1)
        predictions.append(yhat[0][0])
    predictionsarray = np.array(predictions)

    return predictionsarray
```

Je možnost volby, zda model znovu natrénovat, a nebo načíst z disku, což je řádově rychlejší. Toto není jediný model z tensorflow. Byl vytvořen i stacked LSTM model, který obsahuje více vrstev. Obě verze jsou v krokové i batchové verzi. U vícevrstvých sítí je počet vrstev nastaven jako vstupní parametr, proto lze provést optimalizaci nejen na počet jednotlivých neuronů, ale i na počet vrstev. Výčet uzavírá tzv. bidirectional LSTM model. Podobným způsobem lze v tensorflow vytvořit i multi layer perceptron modely.

8. Co software umí

V dnešní době není žádný problém najít si na internetu nějaký zajímavý funkční model. Machine learning urazil již dlouhou cestu a některé jeho formy jsou velmi sofistikované a je velmi těžké jim do hloubky porozumět a následně je ještě vylepšit. Přidanou hodnotou této práce není nalezení nějakého nového univerzálního modelu, cílem je plná automatizace, robustnost a snadnost použití i pro uživatele, které zajímá spíše výsledek než to, jakým způsobem software funguje. Myslím, že každý z použitých algoritmů by šel dále a dále vylepšovat a zobecňovat, tento software je ale postaven spíše na široké paletě různě nastavitelných modelů, z nichž bude vybrán vždy ten nejlepší.

Během roku urazil software dlouhou cestu a v průběhu bylo změněno a předěláno jak jeho jádro, tak i některé jeho klíčové komponenty. Ideálním stavem bude, pokud tento software najde své uplatnění a lidé budou moci jeho předpovědi nějak smysluplně využít. Předpovídat může prakticky cokoliv, přesto je vždy nejdůležitější vědět to, co za nás žádný software nezařídí, totiž vědět co předpovídáme, z čeho a proč. S mnoha dalšími úkoly si ale program poradí již sám.

8.1. Automatická volba modelu

Existuje mnoho algoritmů, bylo by ale chybou spoléhat se pouze na jeden z nich. Různá data, různé optimální modely. Vzhledem k tomu, že díky testovacím datům máme možnost modely ohodnotit, můžeme je i navzájem porovnat. Vše musí fungovat bez zásahu člověka. Změní-li se například ve sledovaném procesu některá jeho podstatná část, a tedy i jeho chování, může doposud nejlepší model vykazovat z důvodu nestacionarity velmi špatné výsledky. Program obsahuje více jak 20 modelů a to už vytváří hezkou šanci pro to, aby se pro predikovaná data našel nějaký, který bude pro jejich predikci vhodný.

8.2. Automatická volba parametrů

Všechny modely vykazují různé výsledky pro různá nastavení. Někdy lze v modelu nastavit i okolo 10 parametrů, mezi které může patřit například počet neuronů ve vrstvě, počet vrstev, parametr α u regresních modelů určující vliv méně korelujících členů na výsledek, řád AR modelů, počet historických hodnot, z kterých mají vycházet. Je-li v souboru *config* zapnuta možnost *optimizeit*, bude použit skript k automatickému nalezení optimálních parametrů. Ty je potom ideální nastavit pro sledovaná data jako parametry výchozí. Při optimalizaci máme několik úrovní výstupu. Zaprvé vypsání nejlepších parametrů při dokončení, zadruhé vypsání každého zlepšení a výpisu všech chyb, které nastaly během optimalizace při výpočtu modelu. Následuje ukázka toho, jak takový výstup z průběhu optimalizace může vypadat.

```
Best result 0.07828121492412594 with parameters {'n_steps_in': 35, 'alpha': 1.0} on model Ridge regression
Best result 0.061382391480738284 with parameters {'n_steps_in': 18, 'alpha': 1.0} on model Ridge regression
Best result 0.060459732196807094 with parameters {'n_steps_in': 21, 'alpha': 1.0} on model Ridge regression
```

Obrázek 21 – Výpis průběhu optimalizace

Součástí optimalizace je funkce, která udává v kolika procentech optimalizace se nacházíme a také možnost nastavení časového limitu optimalizace jednoho kroku, čímž můžeme optimalizovat vždy všechny modely aniž bychom zastavili výpočet u výpočetně náročných LSTM modelů.

8.3. Automatická volba délky dat

Další věcí, kterou bylo potřeba vyřešit a zautomatizovat je délka vstupních dat. Někdy se data skokově mění a učení se z již neaktuálních dat by znemožnilo jakékoliv rozumné výstupy modelů. Prvním parametrem který musíme nastavit je limit maximální délky dat. Software si následně data sám rozdělí na několik různě dlouhých úseků a výpočet provede nad každým z nich. Následně vyhodnotí jaká délka dat je pro jaký model optimální.

8.4. Křížová validace

Předpovídáme-li například pověstných 7 hodnot, je možné, že nějaký model vyjde nejlépe čistě z toho důvodu, že se díky náhodě trefil. Následná predikce 'out of sample' podle modelu který byl vyhodnocen jako nejlepší by poté nemusela být vůbec uspokojivá. Proces náhody je nutné v co největší míře eliminovat. To, jakým způsobem je prováděna kontrola mimo vzorek nad kterým byl model natrénován, se nazývá křížová validace. Některé modely vykazují výborné výsledky nad množinou dat, kterou měl k dispozici, jakmile ale požadujeme výstupy nad daty, které model doposud neznal, model může naprosto selhat. To je příznakem v machine learningu velmi častého jevu zvaného overfitting. Účelem modelu není proložit všechny body křivkou tak složitou, aby byla výsledná počítaná chyba co nejmenší, ale to, aby byl model schopen generalizovat, čili nebrat v potaz to, co není důležité.

Vzhledem k tomu, že se jedná o časovou řadu

a není tak není možné vstupy různě promíchávat a střídat, vyřešil jsem vše po svém. Poté, co proběhne jedna celá iterace, což znamená, že se data rozdělí na trénovací a testovací množinu, jsou vypočteny a ohodnoceny všechny modely, data jednoduše o kus zkrátíme a celý proces opakujeme tolikrát, jakou hodnotu nastavíme proměnné *repeatit* v souboru *config*. Prosté, ale plně funkční

9. Main - Struktura hlavního programu

Jedná se o soubor, který volá a využívá soubory ostatní. Importuje veškerá nastavení, data, modely i skripty. Výstupem může být zápis výsledků do databáze, grafický výstup, nebo analýza vstupních dat. Jedná se o nejdelší soubor. Soubor má okolo 680 řádků a proto budou popsány pouze jeho důležité části. Celý kód je poté součástí přílohy.

9.1. Načtení dat a jejich úprava

Jsou tři možnosti. Buď budou použita data testovací, nebo data z CSV, nebo data z databáze. Při vývoji potřebujeme krátká rychle se načítající data. Nejprve serializujeme testovací data pomocí knihovny *pickle* a uložíme je na disk. Pokud máme v nastavení *pickleit*, pak se data na disku přepíšou. Pokud ne, pouze se načtou.

```
if config.evaluate_test_data:
m
    if config.piclkeit:
        pickle_data_all(datalength=config.datalength)

    data_folder = Path("test_data/")
```

```
    for i, j in config.data_all_pickle.items():
        file_name = i + '.pickle'
        file_adress = data_folder / file_name
        try:
            with open(file_adress, "rb") as input_file:
                data = pickle.load(input_file)
        except:
            raise FileNotFoundError("\n \t Error - Nejprve dej v config.py
                                    pickleit = 1, tím se data uloží na disk \n")
```

Data_for_prediction je veličina která vstupuje do modelu. Všechna testovací data jsou jednorozměrná a proto jsou data totožná s *column_test_data*, což je obecně sloupec, který chceme predikovat.

V případě reálných dat je postup následující.

```
if config.evaluate_test_data == 0:

    if config.data_source == 'csv':
        try:
            data_for_predictions_full = pd.read_csv(config.csv_adress,
                                    header=0, index_col=0)
        except Exception as exc:
            raise FileNotFoundError("\n \t ERROR - Data load failed - Setup
                                    CSV adress and column name in config \n")

    if config.data_source == 'sql':

        try:
            data_for_predictions_full = database(server=config.server,
                                    database=config.database,
                                    index_col=config.index_col,
                                    data_limit=config.datalength)
        except Exception as exc:
            raise ConnectionError("\n \t ERROR - Data load from SQL server
                                    failed - Setup server, database and predicted
                                    column name in config \n")

        data_for_predictions_full = data_for_predictions_full.iloc[::-1]
        data_for_predictions_full.index =
                                    pd.to_datetime(data_for_predictions_full.index)
```

Nyní máme data ve formátu dataframe. Následuje vytvoření proměnné *data_shape* abychom věděli rozměr dat. Provedeme kontrolu máme-li data jedno, a nebo vícerozměrná (pro každý případ je syntax řezu poněkud jiný a proto budeme muset často pomocí if podmínky mít 2 větve). Další důležitou kontrolou je máme-li data ve správném tvaru. Zvolil jsem rozměry takové, že zatímco u dataframu mám řádky na indexu 0, v numpy poli mám řádky na indexu 1, je proto nutné data transformovat.

Následuje očištění dat, odstranění outlierů a smazání nekorelujících sloupců.

```
cleaned_data = data_clean(data_for_predictions_full)

if config.remove_outliers:
```

```
cleaned_data = remove_outliers(data_for_predictions,
                               predicted_column_index=predicted_column_index,
                               threshold=config.remove_outliers)


# Matice korelací
corr = cleaned_data.corr()
names_to_keep = corr[corr[predicted_column_name] >
                               config.correlation_threshold].index
corelated_data = data_for_predictions_full[names_to_keep]
```

V případě, že je v nastavení nastaveno, že k výpočtu se mají použít i jiné sloupce, data normalizujeme.

```
for i in range(len(corelated_data.columns)):
    normalized_column = data_for_predictions_unnormalized[i, :].reshape(-1, 1)
    scaler = preprocessing.MinMaxScaler(feature_range=(0, 1))
    scaled = scaler.fit_transform(normalized_column)
    data_for_predictions_unnormalized[i, :] = scaled.reshape(-1)
```

Dále voláme funkci analyze (pouze pokud jsme tak nastavili soubor *config*). V případě, že nepoužíváme data testovací, chceme dále data rozdělit na různě dlouhé úseky. Definujeme si tyto délky.

```
min_data_length = 3 * config.predicts + config.repeatit * config.predicts
data_lengths = [data_length, int(data_length / 2), int(data_length / 4),
                               min_data_length + 50, min_data_length]
data_lengths = [k for k in data_lengths if k >= min_data_length]
data_number = len(data_lengths)
```

V případě krátkých dat vyřadíme křížovou validaci i zkracování dat. Dále si vytvoříme proměnnou *data_all*, která bude v případě testovacích dat souhrn všech dat, v případě reálných dat schránka na různě zkrácená data, která budou definována později.

9.2. Optimalizace

Dalším blokem je volání funkce pro nalezení optimálních parametrů popsané výše. Změříme si také čas, kolik který model pro optimalizaci potřeboval.

```
if config.optimizeit:
    best_model_parameters = {}
    models_optimizations_time = config.used_models.copy()

    for i, j in config.used_models.items():
        model_kwargs = {**config.models_parameters[i], **params_everywhere}
        start_optimization = time.time()

        try:
            best_kwargs = optimize(j, model_kwargs,
                                   config.models_parameters_limits[i],
                                   data_for_predictions, fragments=config.fragments,
                                   iterations=config.iterations,
                                   time_limit=config.optimizeit_limit, name=i,
                                   details=config.optimizeit_details)
            best_model_parameters[i] = best_kwargs
```

```
            for k, l in best_kwargs.items():

                config.models_parameters[i][k] = l

        except Exception as exc:
            warnings.warn("\n \t Optimization didn't finished - {}
                          \n".format(exc))

        finally:
            stop_optimization = time.time()
            models_optimizations_time[i] = (stop_optimization -
                          start_optimization)
```

9.3. Hlavní výpočetní smyčka

Ještě než provedeme hlavní výpočet, definujeme si prázdné schránky pro data a vyplníme je NaN hodnotami. Předtím jsme si zjistili počet použitých modelů, počet predikovaných hodnot a počet délek dat.

```
results_matrix = np.zeros((config.repeatit, models_number, data_number,
                           config.predicts))
test_matrix = np.zeros((config.repeatit, models_number, data_number,
                           config.predicts))
evaluated_matrix = np.zeros((config.repeatit, models_number, data_number))
results_matrix.fill(np.nan)
test_matrix.fill(np.nan)
evaluated_matrix.fill(np.nan)
results_shape = results_matrix.shape
```

Následuje smyčka, která celý výpočet opakuje, čímž zajišťuje křížovou validaci.

```
for r in range(config.repeatit):
```

Data si nastříháme na různé délky.

```
    if config.evaluate_test_data == 0:

        data_all = {}

        if len(data_shape) == 1:
            for i in range(len(data_lengths)):
                data_all['Trimmed data ' + str(data_lengths[i])] =
                           data_for_trimming[-data_lengths[i]:]

        else:
            for i in range(len(data_lengths)):
                for j in range(data_shape[0]):
                    data_all['Trimmed data ' + str(data_lengths[i])] =
                           data_for_trimming[:, -data_lengths[i]:]
        data_names = list(data_all.keys())
```

A všechny následně rozdělíme na množinu trénovací a testovací.

```
    train_all = []
    test_all = []
```

```
        for i in range(data_number):
            train_all.append([0] * len(list(data_all.values())[i]))
            test_all.append([0] * len(list(data_all.values())[i]))
            train_all[i], test_all[i] = split(list(data_all.values())[i], predicts
                                = config.predicts,
                                predicted_column_index=predicted_column_index)
```

Jádro výpočtů vypadá následovně.

```
        for m, (n, o) in enumerate(config.used_models.items()):
            for p, q in enumerate(train_all):

                try:
                    start = time.time()
                    results_matrix[r, m, p] = o(q, **params_everywhere,
                                **config.models_parameters[n])
                    test_matrix[r, m, p] = test_all[p]

                except Exception as err:
                    warnings.warn("\n \t Error in compute {} model on data {} : \n
                                \t \t {}".format(n, p, err))

                finally:
                    end = time.time()

            models_time[n] = (end - start)
```

Smyčku uzavíráme tím, že celá data zkrátíme, abychom mohli postup opakovat.

9.4. Ohodnocení modelů

Podíváme se na matici výstupů a na to, jak následně postupovat. V případě, že celý výpočet budeme 3x opakovat kvůli křížové validaci, že pro výpočet vybereme 15 modelů, data rozdělíme na 4 různé délky a chceme predikovat 7 hodnot, výsledná matice bude mít rozměr (3, 15, 4, 7).

Vytvoříme smyčku, kde všechny vypočtené hodnoty ohodnotíme pomocí modulu *test_pre* a hodnotícího kritéria MAPE, nebo RMSE.

```
    for i in range(results_shape[0]):
        for j in range(results_shape[1]):
            for k in range(results_shape[2]):

                mape_matrix[i, j, k] = test_pre(results_matrix[i, j, k, :],
                                test_matrix[i, j, k, :],
                                predicts=config.predicts)
```

Uděláme si průměry skrze první dimenzi, abychom vyloučili náhodný úspěch a měli relevantnější data. Zkrácená matice nese název *repeated_average*. Podle toho jestli jsou vstupem data reálná nebo data testovací, hledáme pro každý model buď průměr výsledků, a nebo jejich minimum. U testovacích dat nás totiž zajímá jak si vede model nad různým typem dat. Chceme-li naopak něco predikovat, zajímá nás nad jak dlouhými daty vychází výsledky nejlíp.

Uvedu verzi pro reálná data. Nejprve najdeme v matici hodnotících kritérií nejmenší hodnotu. Její index v první dimenzi označuje o který model se jedná.

```
best_model_index = np.unravel_index(np.nanargmin(model_results),
                                    shape=model_results.shape)[0]
best_model_matrix = repeated_average[best_model_index]
```

Obdobným způsobem najdeme i index nejlepších dat. Máme-li indexy, uložíme si název nejlepšího modelu, parametry nejlepšího modelu a model jako takový, dále si uložíme název dat, jejich délku, také výsledky nejlepšího modelu a hodnotu hodnotícího kritéria. Všechny hodnoty související s nejlepším modelem následně vymažeme z kopie matice výsledků a celý proces několikrát opakujeme abychom měli hodnoty dalších dobře vycházejících modelů. Terminologii jsem zvolil takovou, že výše uvedené hodnoty u nejlepšího modelu nesou názvy jako *best_model*, *best_model_name, best_mode_data*, zatímco u dalších modelů nesou názvy *next_model*, *next_model_data* a podobně. Výsledky dalších modelů musejí být logicky zapsané v listu a nikoliv v proměnné. Počet vyhodnocených modelů k porovnání lze nastavit v configu pomocí hodnoty *compareit*.

Následně máme možnost dooptimalizovat nejlépe vyhodnocený model a provést více iterací nad jemněji rozdělenými intervaly. Poslední částí před vyhodnocením výsledků, popřípadě zapsání do databáze je velmi důležitá část vytvoření predikcí. Tentokrát však nikoliv nad trénovací množinou, ale nad aktuálními daty, které končí poslední měřenou hodnotou. Data si ořežeme na takovou velikost, pro kterou vyšel výsledek nejlepšího modelu nejlépe. Vytvoříme si prázdné schránky pro data a vyplníme je NaN hodnotami. Výpočet opět zabalíme do bloku try, jelikož i kdyby výpočet z nějakého důvodu nedopadl, ostatní modely mohou vyjít.

```
best_model_predicts = np.zeros(config.predicts)
best_model_predicts.fill(np.nan)
try:
    if len(data_shape) == 1 or not config.other_columns:
        best_model_predicts = best_model(trimmed_prediction_data,
                            **params_everywhere, **best_model_param)

        if config.data_transform == 'difference':
            best_model_predicts = inverse_difference(best_model_predicts,
                            last_undiff_value)

    else:
        best_model_predicts_unnormalized = best_model(trimmed_prediction_data,
                            **params_everywhere, **best_model_param)
        best_model_predicts_unnormalized =
                            np.array(best_model_predicts_unnormalized).reshap
                            e(-1, 1)
        best_model_predicts =
                            final_scaler.inverse_transform(best_model_predict
                            s_unnormalized)

        if config.data_transform == 'difference':
            best_model_predicts = inverse_difference(best_model_predicts,
                            last_undiff_value)
```

```
                best_model_predicts = best_model_predicts.reshape(-1)

        except Exception as err:
            warnings.warn("\n \t Error in compute {} model on data {}: {}".format(n,
                                        p, err))
```

Velmi podobný kód následně provedeme ve smyčce, kde postupně dosazujeme další modely podle toho, jakých výsledků podle hodnotícího kritéria dosáhly.

9.5. Výstup – Zápis do databáze a graf

Hlavním výstupem softwaru je zápis predikovaných hodnot do databáze. Výsledek je zapsán do schématu *stage*, konkrétně do faktové tabulky *FactProduction* vedle reálných naměřených dat. Hlavním rozdílem je sloupec *DimScenarioId*, kde reálná data, plánovaná data a data předpovídaná mají každá svou hodnotu, čímž je následně můžeme snadno vizualizovat a porovnávat. 7 předpovězených hodnot počtu vyrobených kusů a doby trvání výroby, tedy sloupce Number a Duration, zapsaných v databázi může vypadat například takto.

	DimDateId	DimTimeId	DimShiftOrigId	DimOperationOutBk	DimProductionEventBk	DimProductOutBk	DimEmployeeCode	DimOrderBk	DimScenarioBk	EventStart	Number	Duration
463	2019-05-18	00:00:00.0000000	-1	K1	-1000	-1	D	-1	Prediction D	2019-05-18 00:00:00.000	3230...	80176...
464	2019-05-19	00:00:00.0000000	-1	K1	-1000	-1	D	-1	Prediction D	2019-05-19 00:00:00.000	3232...	80490...
465	2019-05-20	00:00:00.0000000	-1	K1	-1000	-1	D	-1	Prediction D	2019-05-20 00:00:00.000	3234...	80756...
466	2019-05-21	00:00:00.0000000	-1	K1	-1000	-1	D	-1	Prediction D	2019-05-21 00:00:00.000	3259...	81308...
467	2019-05-22	00:00:00.0000000	-1	K1	-1000	-1	D	-1	Prediction D	2019-05-22 00:00:00.000	3261...	81142...
468	2019-05-23	00:00:00.0000000	-1	K1	-1000	-1	D	-1	Prediction D	2019-05-23 00:00:00.000	3263...	81233...
469	2019-05-24	00:00:00.0000000	-1	K1	-1000	-1	D	-1	Prediction D	2019-05-24 00:00:00.000	3266...	81384...
470	2019-05-19	00:00:00.0000000	-1	K1	-1000	-1	D	-1	Prediction H	2019-05-19 00:00:00.000	1401....	3481....
471	2019-05-19	01:00:00.0000000	-1	K1	-1000	-1	D	-1	Prediction H	2019-05-19 01:00:00.000	1398....	3475....
472	2019-05-19	02:00:00.0000000	-1	K1	-1000	-1	D	-1	Prediction H	2019-05-19 02:00:00.000	1398....	3475....
473	2019-05-19	03:00:00.0000000	-1	K1	-1000	-1	D	-1	Prediction H	2019-05-19 03:00:00.000	1399....	3477....
474	2019-05-18	21:00:00.0000000	-1	K1	-1000	-1	D	-1	Prediction H	2019-05-18 21:00:00.000	1386....	3452....
475	2019-05-18	22:00:00.0000000	-1	K1	-1000	-1	D	-1	Prediction H	2019-05-18 22:00:00.000	1395....	3468....
476	2019-05-18	23:00:00.0000000	-1	K1	-1000	-1	D	-1	Prediction H	2019-05-18 23:00:00.000	1399....	3479....

Jelikož musíme předpovídat zvlášť pro kategorii hodin, dnů, týdnů i měsíců, a navíc často chceme předpovídat více než jeden sloupec dat, obalíme vše v souboru *main* do smyčky. Následně voláme celý soubor jako funkci pro všechny sloupce, které chceme předpovídat. Pro různé časové intervaly se vždy vytvoří různě agregovaná data a jiné hodnoty pro zápis.

Aby se nekryly hodnoty z prvního týdne například s hodnotami z prvního měsíce, má každý predikovaný interval v *DimScenarioId* svou hodnotu. V případě dnů například je hodnota *DimScenarioId* rovna *'Prediction D'*. V databázi následuje řada dalších sloupců, v nichž je většinou kód pro hodnotu nezadáno, jelikož hodnoty nepřísluší konkrétnímu zaměstnanci ani zakázce.

V debug módu jsou vypsány výsledky všech modelů nade všemi testovacími daty včetně potřebného výpočetního času. Lze nastavit, zda je požadován grafický výstup modelu nejlepšího, popřípadě dalších dobře vycházejících modelů ke srovnání. Výsledný graf je vytvořen pomocí knihovny *plotly* a je uložen ve formátu HTML, který by byl v případě uložení na server snadno přístupný odkudkoliv.

Nejdříve si vytvoříme dataframe se všemi požadovanými výsledky, historickými hodnotami a správným časovým indexem. Následně přidáváme do grafu historii, nejlepší výsledek tlustě a černě, meze nejistoty vypočtené zvlášť pomocí ARIMA modelu a nakonec ve smyčce další úspěšné modely. Jednotlivé linie přidáváme následujícím způsobem.

```
trace = go.Scatter(
    name = '1. {}'.format(best_model_name),
    x = complete_dataframe.index,
    y = complete_dataframe['Best prediction'],
    mode = 'lines',
    line = dict(color='rgb(51, 19, 10)', width=4),
    fillcolor = 'rgba(68, 68, 68, 0.3)',
    fill = 'tonexty')
```

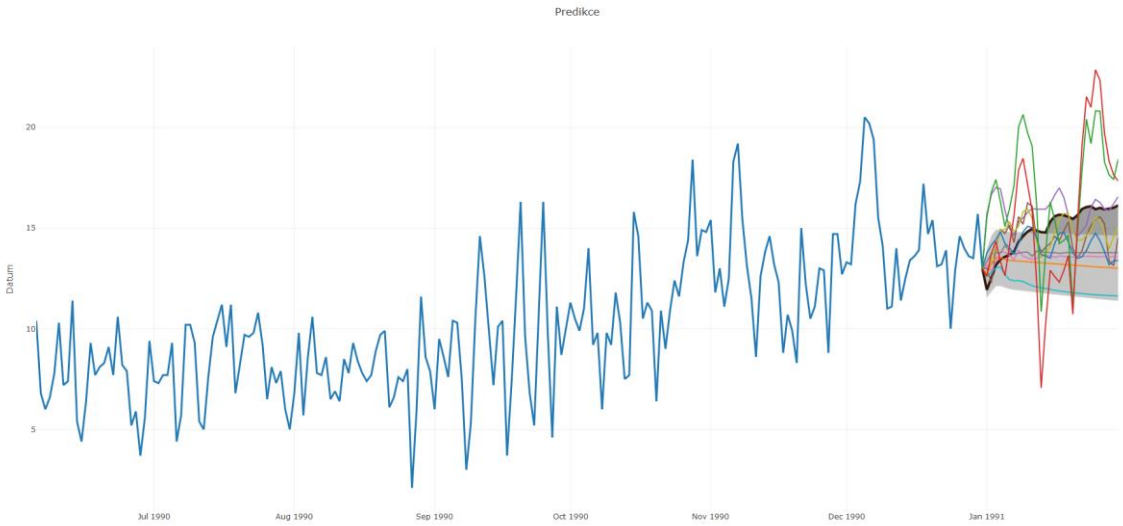
Smyčku s ostatními modely přidáme takto.

```
for i in range(next_number):
    fig.add_trace(go.Scatter(
            x = complete_dataframe.index,
            y = complete_dataframe[next_models_names[i]],
            mode='lines',
            name='{}. {}'.format(i + 2, next_models_names[i])))
```

Definujeme layout, definujeme adresu kam se má graf uložit a graf vykreslíme.

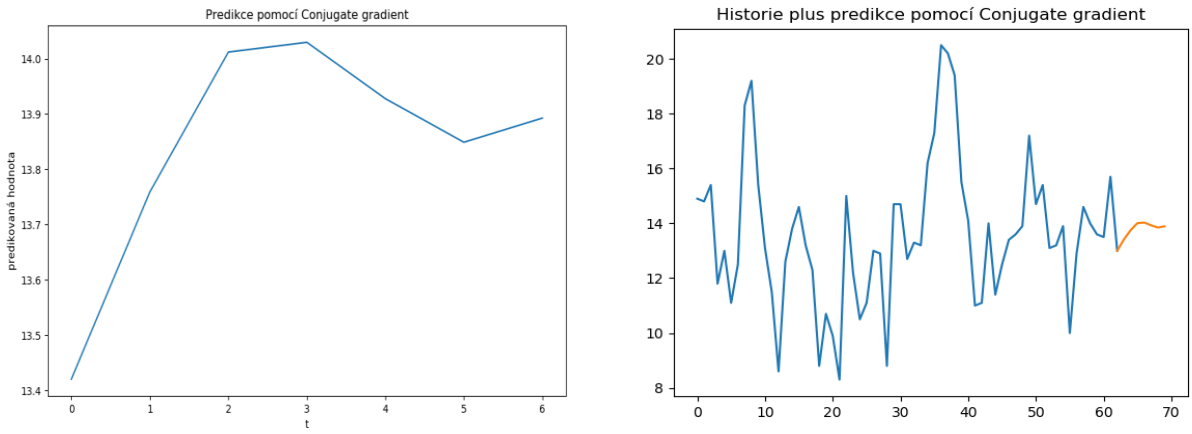
```
graph_data = [lower_bound, trace, upper_bound, history]
fig = go.Figure(data=graph_data, layout=layout)
py.plot(fig, filename='predictions.html')
```

Grafický výstup je interaktivní a po najetí na daný datum vypíše názvy a hodnoty jednotlivých modelů. Ukázka jednoho vytvořeného grafu vypadá takto.



Obrázek 22 – Vyhodnocení n nejlepších modelů

Další možností v debug módu je nastavení *plotallmodels*, které způsobí, že všechny vyhodnocené modely nad všemi daty budou také vykresleny. To se hodí pro seznámení se se všemi modely, jelikož někdy model s horším hodnotícím kritériem může být ve skutečnosti zajímavější. Obzvláště je to možné pokud se snažíme předpovídat anomálie s velkou odchylkou, které nemají periodický charakter. V takovém případě bych také asi doporučil změnit kritérium na RMSE. Vykreslené grafy vypadají tak, jako na následujícím obrázku, vlevo jsou predikované hodnoty a vpravo jsou zahrnuty i hodnoty historické.



Obrázek 23 – Výsledky v grafu

Kromě grafických výstupů jsou vytvořeny i výsledky v tabulkové formě. Tabulka je vytvořena pomocí knihovny *prettytable* pomocí následujících řádků.

```
models_table = PrettyTable()
models_table.field_names = ["Model", "Average MAPE error", "Time"]

for i, j in enumerate(models_names):
    models_table.add_row([models_names[i], model_results[i],
                          models_time[models_names[i]]])
```

Tabulka pro vybrané modely pak může vypadat například takto.

Model	Average MAPE error	Time
AR (Autoregression)	0.07655999014433461	0.0019943714141845703
Autoregressive Linear neural unit	0.045524722708644376	0.0708150863647461
Linear neural unit with weigths predict	0.051659342133590073	0.2672858238220215
Conjugate gradient	0.03852283576452349	0.0040094852447509766
Linear regression	0.06150834324224163	0.001970052719116211
Linear batch regression	0.060474710294667715	0.0009970664978027344
Ridge regression	0.06045973219680705	0.0009975433349609375
Lasso Regression	0.056508214926878464	0.006982326507568359
Bayes Ridge Regression	0.05644200384333423	0.0029916763305664062
Hubber regression	0.057893314358807714	0.031914472579956055
Extreme learning machine	0.09455457153050349	0.0049877166748046875
Batch extreme learning machine	0.048252165497558064	0.00299072265625
Batch Gen Extreme learning machine	0.05623430821034858	0.0019948482513427734

Obrázek 24 – Výsledky v tabulce

Závěr

V této diplomové práci byl vytvořen program na tvorbu predikcí. První teoretická část je především motivační a marketingová. Rozebírá, proč vůbec je dobré pokoušet se některá, konkrétně například výrobní data predikovat. Nabízí možnosti budoucího využití jako například levnou prediktivní údržbu, pokročilé plánování nebo optimalizaci skladových prostor. Popisuje širší rámec tzv. bussiness intelligence, do něhož tato práce spadá. Druhá část je teoretická, popisuje různé způsoby toho, jak donutit stroj něco předpovídat. Nejdříve provádí rešerši možných řešení a následně konkrétně popisuje některé z nich. Jsou rozebrány některé typy neuronových sítí, AR a ARIMA modely nebo některé regresní modely.

Praktická část poté tyto modely implementuje a zajistí jak automatickou volbu modelu, tak i optimálních parametrů či optimální délku dat. Je popsána celá struktura softwaru včetně všech jeho součástí. Důležité části kódu jsou okomentovány, dokumentace je poté součástí kódu samotného, který je celý součástí přílohy. Software je plně funkční a jeho výstupem je buď zápis do databáze nebo interaktivní graf ve formátu HTML. Co možná nejvíc problémů se program pokouší vyřešit sám. Postup při práci se softwarem je zhruba takový. Importujeme data, která chceme předpovídat, vypočteme všechny modely, následně u těch, které vycházejí dobře, optimalizujeme parametry. Není nutné vždy počítat všechny, proto vybereme pár nejlepších a nastavíme zápis do databáze. Výsledkem je zápis predikcí definovaných sloupců v definovaných agregovaných intervalech jako jsou například dny a hodiny.

Jak již to bývá, na softwaru je stále co optimalizovat. Tzv. TODO list stále obsahuje několik položek, které by bylo dobré dořešit, ovšem jinak to v tomto dynamickém světe možná ani nejde, protože s každou vyřešenou položkou se dvě nové objeví. Například by bylo ideální transformaci dat použít jako vstupní parametr a vypočítat pro všechny modely různě transformovaná data. Již je vytvořena transformace na přírůstkový model, který odstraňuje trend, avšak transformaci lze zapnout pouze na všechny modely najednou, nikoliv selektivně. Optimální by bylo vytvořit transformace další, například tzv. Box-Cox, nebo také jinak power transformaci, čímž můžeme zvýraznit neobvyklé hodnoty, čímž v praxi lépe předpovíme například odstávky strojů, které nás právě z důvodu prediktivní údržby zajímají. Zajímavé by také bylo použít hodnotící kritérium s posunem v čase, jelikož pokud predikujeme velkou odstávku jen o krok vedle, model je hodnocen hůře než model, který odstávku nepředpověděl. Obecně lze softwaru vytknout, že v případě velkého počtu predikovaných hodnot vychází relativně malý rozptyl a některé modely mají tendenci stát se hladkou křivkou sledující obecný trend dat, což má za následek celkově menší součet chyb, ale na druhou výsledek nevypadá na první pohled věrohodně. Možností v nastavení je relativně hodně a tak další životní fází softwaru je fáze testování, přizpůsobování

pro různé datasety a především doslova hraní si s nespočtem různých kombinací nastavitelných parametrů.

Na závěr spíše pro odlehčení mohu uvést, kolik toho takový software musí vlastně spočítat. Predikujeme-li 7 hodnot, musíme je vynásobit 20 modely, což dále násobíme 4, jelikož počítáme data různých délek. Dalším číslem může být 5, jelikož kvůli křížové validaci počítáme vše několikrát abychom vyloučili náhodu. To celé počítáme zvlášť pro hodiny, dny a minuty. Nesmíme zapomenout, že pro dobré výsledky je velmi důležité vhodně zvolit vstupní parametry, a jelikož vše musí fungovat automaticky, tak v případě, že budou mít modely v průměru 4 nastavitelné parametry, které při optimalizaci v každé iteraci rozdělíme do 4 intervalů, kdy iteraci provedeme 4 krát, tak finální číslo, kolikrát musíme vypočítat některý z modelů je okolo 8 600 000.

S touto diplomovou prací však život tohoto softwaru nekončí. Stále je a bude co zlepšovat. Nelze opomenout, že některé parametry z nastavení která lze nastavit pouze pro všechny modely najednou, by bylo ideální diverzifikovat pro jednotlivé modely. Pro některé modely bude lépe vycházet MAPE pro některé zase kvadratické chybové kritérium. Pro některé modely je vhodné provést určitou transformaci dat, pro některé nikoliv. Na závěr uvedu, že by bylo ideální v předpřípravě provést tzv. smooting, a to v několika úrovních. V případě, že bychom chtěli plně automatizovaný model, dosáhli bychom hranice sta milionů vypočtených modelů. Každý model přitom provádí někdy i velice složité výpočty, aby se k výsledku vůbec dostal. Co se týče výpočetního času, tak se dnes již nejedná o zásadní problém. Vyjmeme-li výpočetně náročné modely a optimalizujeme-li v rozumné míře, tak se i na běžném počítači bavíme v řádu minut, nemluvě o možnostech distribuovaného výpočtu. Myslím že výše uvedené řádky relativně pěkně vystihují povahu umělé inteligence. Pomineme-li nadšené články lidí z marketingových oddělení, můžeme umělou inteligenci směle označit jako poměrně zajímavou bruteforce metodu, kterou bych se asi neodvážil označit za hloupou. Přes to všechno bych však zatím raději zůstal u termínu pokročilé statistické metody, kterým dnešní velmi pokročilý výpočetní výkon umožnil výstupy, které by možná i samotné otce počítačových technologií příjemně překvapily.

Bibliografie

[1] KIMBALL, Ralph a Margy ROSS. *The Data Warehouse Toolkit - Second Edition* [online]. 2002. ISBN 0-471-20024-7. Dostupné z: http://www.dsc.ufcg.edu.br/~sampaio/Livros/alph Kimball. The Data Warehouse Toolkit.. The Complete Guide to Dimensional Modelling (Wiley,2002)(ISBN 0471200247)(449s).pdf

[2] *Seznamte se s BI | DAQUAS* [online]. [vid. 2019-07-15]. Dostupné z: https://www.daquas.cz/articles/379-seznamte-se-s-bi

[3] *MAE and RMSE — Which Metric is Better? – Human in a Machine World – Medium* [online]. [vid. 2019-04-03]. Dostupné z: https://medium.com/human-in-a-machine-world/mae-and-rmse-which-metric-is-better-e60ac3bde13d

[4] SVETUNKOV, Ivan. *Naughty APEs and the quest for the holy grail* [online]. Dostupné z: http://forecasting.svetunkov.ru/en/2017/07/29/naughty-apes-and-the-quest-for-the-holy-grail/

[5] *Ljung–Box test* [online]. Dostupné z: https://en.wikipedia.org/wiki/Ljung–Box_test

[6] MATIGNON, Randall. *Neural Network Modeling Using SAS Enterprise Miner*. B.m.: AuthorHouse, 2005. ISBN 1418423416.

[7] MAKRIDAKIS, Spyros, Evangelos SPILIOTIS a Vassilios ASSIMAKOPOULOS. Statistical and Machine Learning forecasting methods: Concerns and ways forward. *PLoS ONE* [online]. 2018, **13**(3), 1–26. ISSN 19326203. Dostupné z: doi:10.1371/journal.pone.0194889

[8] CHAN, Hing Kai, Shuojiang XU a Xiaoguang QI. A comparison of time series methods for forecasting container throughput. *International Journal of Logistics: Research and Applications* [online]. 2018, **0**(0), 1–10. ISSN 1367-5567. Dostupné z: doi:10.1080/13675567.2018.1525342

[9] HANSEN, James V., James B. MCDONALD a Ray D. NELSON. Time series prediction with genetic-algorithm designed neural networks: an empirical comparison with modern statistical models. *Computational Intelligence* [online]. 1999, **15**(3), 171–184. ISSN 08247935. Dostupné z: doi:10.1111/0824-7935.00090

[10] ZHANG, Xingyu, Yuanyuan LIU, Min YANG, Tao ZHANG, Alistair A. YOUNG a Xiaosong LI. Comparative Study of Four Time Series Methods in Forecasting Typhoid Fever Incidence in China. *PLoS ONE* [online]. 2013, **8**(5). ISSN 19326203. Dostupné z: doi:10.1371/journal.pone.0063116

[11] BIRATTARI, Mauro, Gianluca BONTEMPI a Hugues BERSINI. Lazy learning meets the recursive least squares algorithm. *Advances in Neural Information Processing Systems* [online]. 1999, (1997), 375–381. Dostupné z: http://dl.acm.org/citation.cfm?id =340534.340673

[12] BONTEMPI, Gianluca a Souhaib Ben TAIEB. LNBIP 138 - Machine Learning Strategies for Time Series Forecasting [online]. 2013, (January). Dostupné z: doi:10.1007/978-3-642-36318-4

[13] TEAM AUQUAN. *Time Series Analysis for Financial Data VI— GARCH model and predicting SPX returns* [online]. [vid. 2019-03-29]. Dostupné z: https://medium.com/ auquan/time-series-analysis-for-finance-arch-garch-models-822f87f1d755

[14] DANEL, ROMAN ING., Ph.D. Predikce Časové Řady Pomocí Autoregresního Modelu. 2004.

[15] GUO, Wenge. Chapter 2 Multiple Regression I (Part 1). nedatováno, (Part 1), 1–11.

[16] MICHAŁ OLESZAK. *Regularization: Ridge, Lasso and Elastic Net (article) - DataCamp* [online]. [vid. 2019-08-09]. Dostupné z: https://www.datacamp.com/community/ tutorials/tutorial-ridge-lasso-elastic-net

[17] KONEČNÝ, V, A MATIÁŠOVÁ a I RÁBOVÁ. Učení n-vrstvé neuronové sítě. 2005.

[18] *Gradient Boosting explained [demonstration]* [online]. [vid. 2019-07-30]. Dostupné z: https://arogozhnikov.github.io/2016/06/24/gradient_boosting_explained.html

[19] GOH, Gabriel. Why Momentum Really Works. *Distill* [online]. 2017, **2**(4), e6 [vid. 2019-03-19]. ISSN 2476-0757. Dostupné z: doi:10.23915/distill.00006

[20] RUDER, Sebastian. An overview of gradient descent optimization algorithms [online]. 2016 [vid. 2019-04-09]. Dostupné z: http://arxiv.org/abs/1609.04747

[21] *ReLU and Softmax Activation Functions · Kulbear/deep-learning-nano-foundation Wiki* [online]. [vid. 2019-04-05]. Dostupné z: https://github.com/Kulbear/deep-learning-nano-foundation/wiki/ReLU-and-Softmax-Activation-Functions

[22] *Activation Functions in Neural Networks – Towards Data Science* [online]. [vid. 2019-

04-03]. Dostupné z: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6

[23] GROSSE, Roger. Lecture 15 : Exploding and Vanishing Gradients. *Cs.Toronto.Edu* [online]. 2017, 1–11. ISSN 10706631. Dostupné z: http://www.cs.toronto.edu/~rgrosse/courses/csc321_2017/readings/L15 Exploding and Vanishing Gradients.pdf

[24] SCIENCE, Computer. University of Science and Technology in Cracow Master of Science Thesis A comparative analysis and evaluation of various machine learning algorithms for facial recognition Analiza porównawcza wybranych algorytmów uczenia maszynowego do rozpoznawania twarz. nedatováno.

[25] *Master MATLAB: visualize the matrix quadratic form - YouTube* [online]. [vid. 2019-04-11]. Dostupné z: https://www.youtube.com/watch?v=-pdeXfoAl_k

[26] KRUIS, Jaroslav. Nelineární optimalizace a numerické metody. In: . nedatováno.

[27] REFSENAES, Runar Heggelien. A Breif Introduction to the Conjugate Gradient Method (Lecture Notes) [online]. 2009. Dostupné z: www.idi.ntnu.no/~elster/tdt24/tdt24-f09/cg.pdf

[28] *GitHub - rcassani/mlp-example: Code for a simple MLP (Multi-Layer Perceptron)* [online]. [vid. 2019-07-14]. Dostupné z: https://github.com/rcassani/mlp-example

[29] *Understanding LSTM Networks -- colah's blog* [online]. [vid. 2019-07-14]. Dostupné z: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

[30] DANIEL SMILKOV, Shan Carter. *A Neural Network Playground* [online]. [vid. 2019-04-02]. Dostupné z: http://playground.tensorflow.org

[31] *Numba: A High Performance Python Compiler* [online]. [vid. 2019-07-21]. Dostupné z: https://numba.pydata.org/

[32] *GitHub - numba/numba: NumPy aware dynamic Python compiler using LLVM* [online]. [vid. 2019-07-21]. Dostupné z: https://github.com/numba/numba

[33] *Dask: Scalable analytics in Python* [online]. [vid. 2019-07-21]. Dostupné z: https://dask.org/

[34] *Step-by-Step Use of Google Colab's Free TPU - Heartbeat* [online]. [vid. 2019-07-30]. Dostupné z: https://heartbeat.fritz.ai/step-by-step-use-of-google-colab-free-tpu-75f8629492b3

[35] *Watson Studio - Watson Studio | IBM* [online]. [vid. 2019-04-03]. Dostupné z: https://www.ibm.com/cloud/watson-studio

[36] *Retail Forecasting: Step 4 of 6, train regression models | Azure AI Gallery* [online]. [vid. 2019-04-03]. Dostupné z: https://gallery.azure.ai/Experiment/Retail-Forecasting-Step-4-of-6-train-regression-models-2

[37] NUMPY COMMUNITY. NumPy Reference. *October*. 2016, **1**(May), 1–1146.

[38] MCKINNEY, Wes a PyData Development TEAM. Pandas - Powerful Python Data Analysis Toolkit. *Pandas - Powerful Python Data Analysis Toolkit*. 2015, 1625.

[39] ROSSUM, Guido Van a Fred L DRAKE. The Python Library Reference. *October* [online]. 2010, 1–1144. Dostupné z: http://scholar.google.com/scholar?q=intitle:Python+Library+Reference#0

[40] *Max execution time for function in python (flask) - Stack Overflow* [online]. [vid. 2019-08-09]. Dostupné z: https://stackoverflow.com/questions/18581174/max-execution-time-for-function-in-python-flask/18751931

[41] *ARIMA Model - Complete Guide to Time Series Forecasting in Python | ML+* [online]. [vid. 2019-07-16]. Dostupné z: https://www.machinelearningplus.com/time-series/arima-model-time-series-forecasting-python/