FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Klasifikace časových řad v Julia
Student:	Antonín Kříž
Vedoucí:	Ing. Tomáš Kalvoda, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Znalostní inženýrství
Katedra:	Katedra aplikované matematiky
Platnost zadání:	do konce letního semestru 2023/2024

Pokyny pro vypracování

1. Seznamte se s problematikou časových řad a jejich klasifikací. Nastudujte state-of-the-art algoritmy pro klasifikaci časových řad MiniROCKET a DTW+KNN (Dynamic Time Warping with k Nearest Neighbors).
2. Implementujte tyto dva algoritmy ve formě balíčku v programovacím jazyce Julia.
3. Balíček vybavte dokumentací a testy.
4. Na vhodně zvolených volně dostupných datasetech porovnejte výkonnost své implementace s implementací těchto algoritmů v Python balíčku sktime.

Elektronicky schválil/a Ing. Magda Friedjungová, Ph.D. dne 22. ledna 2023 v Praze.

Bakalářská práce

KLASIFIKACE ČASOVÝCH ŘAD V JULIA

Antonín Kříž

Fakulta informačních technologií
Katedra aplikované matematiky
Vedoucí: Ing. Tomáš Kalvoda, Ph.D.
11. května 2023

Obsah

Poděkování vii

Prohlášení viii

Abstrakt ix

Seznam zkratek xi

Motivace 1

1 Úvod 3
- 1.1 Co je klasifikace časových řad? . . . 4
- 1.2 Vybrané metody klasifikace časových řad . . . 4
  - 1.2.1 Metody založené na vzdálenostech . . . 5
  - 1.2.2 Metody založené na *shapeletech* . . . 5
  - 1.2.3 Slovníkové metody . . . 6
  - 1.2.4 *Deep Learning* a Konvoluční Neuronové Sítě . . . 6
  - 1.2.5 Další metody . . . 7
- 1.3 Struktura práce . . . 7

2 Analýza existujících nástrojů a technologií 9
- 2.1 Algoritmy . . . 9
  - 2.1.1 KNN s DTW . . . 9
  - 2.1.2 MINIROCKET . . . 14
- 2.2 Jazyky . . . 24
  - 2.2.1 Python . . . 24
  - 2.2.2 Julia . . . 26
- 2.3 Nástroje . . . 27
  - 2.3.1 sktime . . . 28
  - 2.3.2 MLJ . . . 30
  - 2.3.3 UCR Archive . . . 30

3 Implementace 33
- 3.1 Rozšiřování MLJ o další modely . . . 33
- 3.2 Porovnání implementace . . . 37
  - 3.2.1 KNN s DTW . . . 37
  - 3.2.2 MINIROCKET . . . 39
- 3.3 Struktura balíčku . . . 41

4 Experimenty **47**
4.1 Testovací prostředí 47
4.2 Testovací *datasety* 48
4.3 Pozorování 49

5 Závěr **55**

A Tabulka výsledků *bake off* testu **57**
A.1 Tabulka výsledků *bake off* testu seřazena dle času 57
A.2 Tabulka výsledků *bake off* testu seřazena dle času 60

Bibliografie **63**

Obsah přiloženého média **71**

Seznam obrázků

1.1 Nákres pro porovnání EKG měření bez (horní) a se (dolní) srdeční arytmií [38] . . . 5
1.2 Ukázka možné architektury konvoluční neuronové sítě . . . 6

2.1 Ukázka principu fungování Dynamic Time Warping algoritmu, respektive možného mapování (přerušovaná čára) indexů časových řad τ_α a τ_β. Pro názornost jsou časové řady vykresleny spojitou čarou, ačkoliv DTW pracuje pouze s diskrétními sekvencemi. . . . 11
2.2 *Heatmap* hodnot z matice vzdáleností DTW pro výše uvedené časové řady z Obrázku 2.1 značící vzájemné relativní vzdálenosti mezi prvky. Tmavě modrá barva značí nejnižší, oranžová naopak nejvyšší vzdálenost. 12
2.3 Sakoe–Chiba pásmo s $window = 3$ a Itakurův parallelogram se $slope = 2.0$ pro 2 časové řady o délce $n = 12$. Prostor, na který by bylo DTW omezeno, je vyznačen modře. . . . 12
2.4 Ilustrace funkce algoritmu LB_Keogh. τ_c značí obalenou časovou řadu, τ_q porovnávanou časovou řadu, e obal (horní a spodní meze) a d zvýrazňuje rozdíly τ_q a obalu e. . . . 13
2.5 Ukázka $1D$ konvoluce. f značí *feature map*, ω značí filtr (*kernel*) a τ značí časovou řadu, nad kterou probíhá konvoluce . . . 14
2.6 Ukázka *dilatace kernelu*. d značí míru dilatace *kernelu*, bílé buňky značí mezery, respektive nulové hodnoty, vložené do *kernelu* o které byl *kernel* roztažen . . . 15
2.7 Průměrné pořadí (umístění) algoritmu ROCKET v *benchmarku* při různých počtech kernelů [32] . . . 17
2.8 Průměrné pořadí (umístění) algoritmu MINIROCKET v *benchmarku*, zmíněném již v Sekci 2.1.2.1, při různých délkách a různých podmnožinách kernelů [31] . . . 19
2.9 Průměrné pořadí (umístění) algoritmu MINIROCKET v *benchmarku*, viz Sekce 2.1.2.1, s použitím pouze *ppv* proti *ppv* spolu s max jako sdružovací funkce [31] . . . 20
2.10 Hodnoty prvních 100 členů posloupnosti používané algoritmem MINIROCKET . . . 21
2.11 100 náhodně vygenerovaných hodnot z rovnoměrného rozložení $\mathbb{U}(0, 1)$ . . . 21

3.1 Zjednodušené uspořádání balíčku s exportovanými moduly, strukturami a funkcemi . . . 42
3.2 Struktura modulu `MiniRocket` . . . 43
3.3 Struktura modulu `KNNDTW` . . . 44

3.4 Struktura modulu `DataSets` 45

4.1 Porovnání rychlosti algoritmu MiniRocket v *bake off* testu 50
4.2 Porovnání rychlosti algoritmu *k-Nearest Neighbors* a *Dynamic Time Warping* se Sakoe–Chiba pásmem poloměru 10 v *bake off* testu 51
4.3 Počet provedených alokací algoritmem MiniRocket z balíčku sktime v závislosti na počtu transformovaných časových řad 52

Seznam tabulek

2.1 Rocket vs MiniRocket 18

4.1 Vybrané *datasety* 49
4.2 Rychlost jednotlivých algoritmů na vybraných *datasetech*. 49
4.3 Porovnání DTW a DTW s Itakurovým parallelogramem na vybraných *datasetech*. 53

Seznam výpisů kódu

1 Ukázka kódu v jazyku Python 24
2 Ukázka kódu v jazyku C 25
3 Ukázka kódu v jazyku Julia 26
4 Pseudokód znázorňující vektorizaci kódu na sčítání vektorů, respektive sčítání prvků dvou polí 27
5 Ukázka *multiple dispatch* v Julia [69] 28
6 Ukázka chybějícího *multiple dispatch* v C++ [69] 29
7 Implementace struktury modelu KNN v MLJ 34
8 Implementace funkce `MLJModelInterface.fit(...)` pro KNN v MLJ . . . 34
9 Implementace funkce `MLJModelInterface.predict(...)` pro KNN v MLJ 35
10 Implementace funkce `reformat(...)` pro KNN pomocí rozhraní MLJ . . 35
11 Implementace funkce `selectrows(...)` pro KNN pomocí rozhraní MLJ . 36
12 Inzerce modelu KNN pro balíček MLJ 36
13 Znázornění míst alokujících paměť ve funkci `transform` v sktime implementaci algoritmu MiniRocket 40
14 Znázornění míst alokujících paměť ve funkci `transform` v algoritmu MiniRocket implementovaném v této práci 41

Rád bych poděkoval Ing. Tomáši Kalvodovi, Ph.D. za odborné a neutuchající vedení a konzultace po celou dobu mé práce, své rodině a stejně tak přítelkyni Lindě a ostatním kamarádům za veškerou podporu a skupince Tangus, bez které bych se až k těmto řádkům třeba nedostal.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací. Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 11. května 2023 Antonín Kříž

Abstrakt

Klasifikace časových řad je komplexní poblém v oboru strojového učení. Moderní metody řešící tento problém jsou náročné na výkon a jejich efektivní implementace je důležitější než kdy dřív. Tato práce se zabývá analýzou metod pro klasifikaci časových řad a i efektivně implementuje metody MiniRocket a k-Nejbližších Sousedů s *Dynamic Time Warping*.

V rámci této práce je navrženo a implementováno několik různých optimalizací těchto algoritmů v porovnání s jejich implementací v Python balíčku sktime. Mezi tyto optimalizace, které jsou podrobně popsány, patří redukce nutných alokací paměti, paralelizace a vektorizace výpočtů a redukce nutných kroků algoritmu pro dosažení výsledku. Výkonnost této implementace je experimentálně ověřena na 113 *datasetech* z archivu časových řad Kalifornské univerzity v Riverside, kdy dosahuje až 17krát vyššího výkonu než tytéž algoritmy v balíčku sktime.

Klíčová slova časové řady, klasifikace časových řad, strojové učení, konvoluční neuronové sítě, MINIROCKET, k-Nejbližších Sousedů, Dynamická Časová Deformace, UCR archiv, Julia, Python, porovnání výkonu

Abstract

Time series classification is a complex problem in the field of machine learning. Modern methods addressing this problem are demanding in performance and their efficient implementation is more important than ever. This paper analyzes methods for time series classification and also efficiently implements the MiniRocket and k-Nearest Neighbors methods with *Dynamic Time Warping*.

In this work, several different optimizations of these algorithms are proposed and implemented against their implementation in the Python package sktime. These optimizations, which are described in detail, include reducing the necessary memory allocations, parallelization and vectorization of the computations, and reducing the necessary algorithm steps to achieve the result. The performance of this implementation is experimentally verified on 113 *datasets* from the University of California, Riverside time series archive, achieving up to 17 times better performance than the same algorithms in the sktime package.

Keywords time series, time series classification, machine learning, convolutional neural networks, MINIROCKET, k-Nearest Neighbors, Dynamic Time Warping, UCR archive, Julia, Python, benchmark, performance comparison

Seznam zkratek

AOT	Ahead-of-Time
CNN	Convolutional Neural Networks
CUDA	Compute Unified Device Architecture
DTW	Dynamic Time Warping
GIL	Global Interpreter Lock
JIT	Just-in-Time
KAIST	Korea Advanced Institute of Science and Technology
KNN	k-Nearest Neighbors
LLVM	Low-Level Virtual Machine
MLJ	A Machine Learning Framework for Julia
MMI	MLJModelInterface
MWE	Minimal Working Example
SOTA	State Of The Art
TSC	Time Series Classification

Motivace

V listopadu 2022 jsem se jako výměnný student na KAIST v Korejské republice zúčastnil meziuniverzitní týmové soutěže v datové analýze[1] [1] ve stylu Kaggle soutěží[2] [2] na téma detekce typu uživatelů na základě způsobu využívaní mobilní aplikace pro *smart banking*, kde se jedna z částí soutěže řešila problém klasifikace časových řad.

Postupně náš tým při sestavování modelů začal díky velikosti *datasetu* narážet s každou další iterací na problémy s rychlostí námi použitých modelů, kdy se doba trénování modelu stávala pro naše použití neúprosně dlouhou.

Modely, které jsme používali, například ty z balíčku sktime [3, 4], jsou převážně implementované v Pythonu [5] spolu s numpy[3] [6] a numba[4] [7] JIT kompilací. Python, ačkoliv patří v *data-science* mezi nejrozšířenější jazyky, není samotný pro výpočty vyžadující vysokou rychlost optimální a to někdy ani přes zrychlení, které dokáží balíčky numpy a numba přinést [8].

Jedním z mých pokusů o řešení tohoto problému bylo nalezení ekvivalentní implementace těchto modelů v jazyku Julia, který se díky své kompilaci skrze LLVM vyznačuje výrazně vyšším výkonem, zatímco si zachovává uživatelskou přívětivost [9]. K mému překvapení jsem bohužel ale pro jazyk Julia nenašel žádnou již dostupnou implementaci algoritmů pro klasifikaci časových řad.

Rád bych touto prací tedy položil základní kameny pro balíček implementující vybrané algoritmy pro klasifikaci časových řad v Julia a zároveň porovnal výkon s implementací v populárních balíčcích pro Python.

Cíle práce

Cílem teoretické části bakalářské práce je popsat problematiku klasifikace časových řad, popsat algoritmy MiniRocket a KNN+DTW a popsat rozdíly mezi jazyky Julia a Python, který je využíván v případě nejrozšířenějších již existujících balíčků (sktime), které se tématem klasifikace časových řad již zabývají. Cílem praktické části bakalářské práce je implementovat balíček v jazyku Julia s algoritmy MiniRocket a KNN+DTW,

[1] https://datascience-contest.com/
[2] https://www.kaggle.com/docs/competitions#simple-competitions
[3] https://numpy.org/
[4] https://numba.pydata.org/

vybavit balíček testy a dokumentací a porovnat výkonnost nové implementace v jazyku Julia s implementací v balíčku sktime na vybraných *datasetech* z archivu *UCR Time Series Classification datasets*.

Kapitola 1
Úvod

Cílem této kapitoly je definovat a vysvětlit pojmy, se kterými je možné se v této práci setkat, jako například: co jsou časové řady, co znamená klasifikace v rámci strojového učení, vysvětlit problém klasifikace časových řad a některé metody, kterými je možné ke klasifikaci časových řad přistupovat. V závěru této kapitoly se nachází popis struktury této práce a přiblížení obsahu následujících kapitol.

Časová řada je soubor chronologicky seřazených pozorování [10, 11, 12, 13, 14], kde každé pozorování je asociované s daným okamžikem nebo časovým intervalem, což zajišťuje řazení [13, 14]. Většinou se předpokládá, že pozorování jsou od sebe stejně vzdálená v čase, tedy že jsou zaznamenána v pravidelných intervalech [10, 11, 13].

Časové řady mohou být jednorozměrné (*univarate*) či vícerozměrné (*multivariate*). Jednorozměrné časové řady sledují pouze jednu proměnnou, zatímco vícerozměrné sledují více proměnných najednou. Tyto proměnné nejsou závislé pouze na čase, ale mohou být vzájemně závislé i mezi sebou [14]. Jako možné příklady vícerozměrných časových řad je možné uvést ekonomické a meteorologické indikátory jako jsou inflace a HDP [15], nebo třeba teplota, vlhkost a rychlost větru [16].

Matematicky je možné zapsat diskrétní časovou řadu jako

$$\{x_t\}_{t=1}^{N} = \{x_t \mid t = 1, 2, \ldots, N\}$$

kde N je počet pozorování.V případě časově spojitého stochastického procesu by se časová řada $\{x_t\}$ mohla definovat jako množina datových bodů indexovaná spojitou proměnnou času t, tedy

$$\{x(t) \mid t \in [t_1, t_N]\},$$

kde $[t_1, t_N]$ je uzavřený interval, ve kterém probíhalo měření [17].

V rámci této práce se budu zabývat pouze jednorozměrnými diskrétními časovými řadami. Problém klasifikace vícerozměrných časových řad přesahuje svou komplexitou rozsah této práce, vzhledem k tomu, že implementované algoritmy MiniRocket a KNN+DTW vícerozměrné časové řady samy o sobě nepodporují bez dalších úprav. Spojité časové řady tyto algoritmy nepodporují už vůbec.

Analýza časových řad se zabývá vytěžováním dat z časových řad, nalezením trendů a předpovídáním budoucích událostí. Analýza časových řad je rozšířená napříč mnoha

obory, mezi které patří finance [18], strojírenství [19, 20], zdravotnictví [21, 22], marketing a společenské vědy [23] a další. Je možné ji využít například pro analýzu dat, která se mění v čase, například pro předpověď ceny akcií [18] či počasí [24] nebo hledání demografických trendů [23, 25] či detekce anomálií v elektrokardiografii (ECG/EKG) [22].

1.1 Co je klasifikace časových řad?

V oboru strojového učení se můžeme setkat také s pojmem „model“. Modelem se rozumí počítačový program, který je schopný se učit na základě zkušeností E, s ohledem na určitou třídu úloh T a metriku výkonnosti P, tak, že jeho výkonnost, měřená metrikou P, v úlohách T stoupá s množstvím zkušeností E. [26]

„Učení se s učitelem“ (nebo také učení pod dohledem či supervizované strojové učení *supervised machine learning*) je metoda strojového učení, kde je model trénován nad *datasetem* (souborem zkušeností E) s již známými dvojicemi vstupů a výstupů [27, 28]. Model se na takovém vstupním (trénovacím) *datasetu* naučí generalizovat vlastnosti těchto dat (získá zkušenosti E), na základě kterých dokáže pracovat s do té doby neznámými daty [28]. Klasifikace je jednou z metod supervizovaného strojového učení (jedna ze tříd úloh T), kde se model učí kategorizovat nebo klasifikovat vstupní data do předem daných (diskrétních) kategorií, respektive je označit předem danými štítky, na základě generalizovaných vlastností naučených z trénovacích dat [27, 28].

Klasifikace časových řad (anglicky *Time Series Classification*, zkráceně TSC) je specializovaná forma klasifikace, která se zaměřuje na kategorizaci dat ve formě časových řad. Cílem je zařadit data do jedné z několika tříd (nejen) na základě v nich obsažených časových vzorů (*temporal patterns*) [29, 30].

1.2 Vybrané metody klasifikace časových řad

Pro klasifikaci časových řad existuje mnoho algoritmů, které je z většiny možné zařadit do několika kategorií dle toho, jak přistupují k nalezení či extrakci příznaků z časových řad. V následujících sekcích se pokusím některé z nich představit. Zaměřím se převážně na metody založené na vzdálenostech a na slovníkové metody, do kterých je možné zařadit algoritmy MiniRocket a KNN+DTW, se kterými budu pracovat v následujících kapitolách.

Klasifikace časových řad je důležitou součástí mnoha aplikací a ačkoliv byla v průběhu let vyvinuta řada metod, modelů a algoritmů, které, obzvlášť v posledních letech [31], výrazně zvyšovaly přesnost v porovnání se svými předchůdci, tak výpočetní komplexita a škálovatelnost nadále zůstávala závažným problémem [32, 31]. Doba trvání výpočtů dle *benchmarku* popisovaném v práci MiniRocket [31] dosahuje v lepším případě mnoha hodin (u algoritmu cBOSS [33]), v horším několik týdnů (HIVE-COTE [34] a TS-CHIEF [35]). Algoritmus Rocket a jeho evoluce MiniRocket tento problém řeší a při zachování přesnosti podobné ostatním SOTA (*State Of The Art*) algoritmům se doba běhu pohybuje v jednotkách hodin v případě algoritmu Rocket a jednotkách minut v případě MiniRocket [31].

1.2.1 Metody založené na vzdálenostech

Metody založené na vzdálenostech (*distance-based methods*) přistupují k problému klasifikace tak, že porovnávají relativní vzdálenost (respektive odlišnost) mezi jednotlivými časovými řadami, kde menší vzdálenost znamená vyšší podobnost porovnávaných časových řad [36, 37]. Je možné se také setkat s názvem *whole series similarity TSC algorithms*, tedy algoritmy podobnosti celé časové řady, protože tyto algoritmy zpravidla porovnávají časové řady jako celky mezi sebou [37].

V případě klasifikace časových řad se standardně používají pro měření vzdáleností mezi časovými řadami metriky, které jsou schopné kompenzovat drobné odchylky, obzvlášť v rámci časové osy. Takové metriky se nazývají pružné metriky vzdálenosti (*elastic distance measure*) [37]. Ty se hodí hlavně v případě, kdy se příznaky vhodné pro rozhodnutí klasifikace nacházejí po celé délce časové řady, ale časové řady mohou být v rámci časové osy různě posunuté nebo se mohou jejich zajímavé příznaky lišit v délce [37]. Mezi tyto metriky patří například eukleidovská vzdálenost a *Dynamic Time Warping* spolu s dalšími variantami *DTW* jako je *Weighted Dynamic Time Warping* nebo *FastDTW*.

Hodnoty vycházející z těchto metrik se poté finálně vyhodnocují některým ze známých algoritmů pro klasifikaci založeném na vzdálenostech porovnávaných objektů [36], zpravidla pomocí 1-NN modelu (*k-Nearest Neighbors*, kde $k = 1$) [36, 37].

1.2.2 Metody založené na *shapeletech*

V časových řadách se často vyskytují charakteristické tvary (útvary, struktury) nazývané *shapelet*, které mohou od sebe rozlišovat jednotlivé třídy časových řad [36, 37]. Zároveň ale lokace tohoto vzoru nemusí být podstatná, a vzor se může tedy vyskytovat kdekoliv v časové řadě. *Shapelety* jsou „subsekvence“ či „podřady“ (části původní časové řady), které dokáží tento typ charakteristik zaznamenat [37].

Shapelet algoritmy tedy zpracováním časových řad naleznou minimální vzdálenost mezi *shapeletem* a všemi úseky časové řady. Přístup je tedy podobný jako u metod založených na vzdálenostech, ale s tím rozdílem, že se měří vzdálenosti jen mezi dílčími úseky časových řad a ne celými časovými řadami [36].

Obrázek 1.1 Nákres pro porovnání EKG měření bez (horní) a se (dolní) srdeční arytmií [38]

Například je možné takto odlišit EKG měření zdravého pacienta a pacienta se srdeční arytmií, ukázané na Obrázku 1.1, kde jedno z těchto měření může vykazovat charakteristiky, které se nemusí nenacházet v měření druhém a zároveň se může lišit například rychlost srdečního tepu [36, 37].

1.2.3 Slovníkové metody

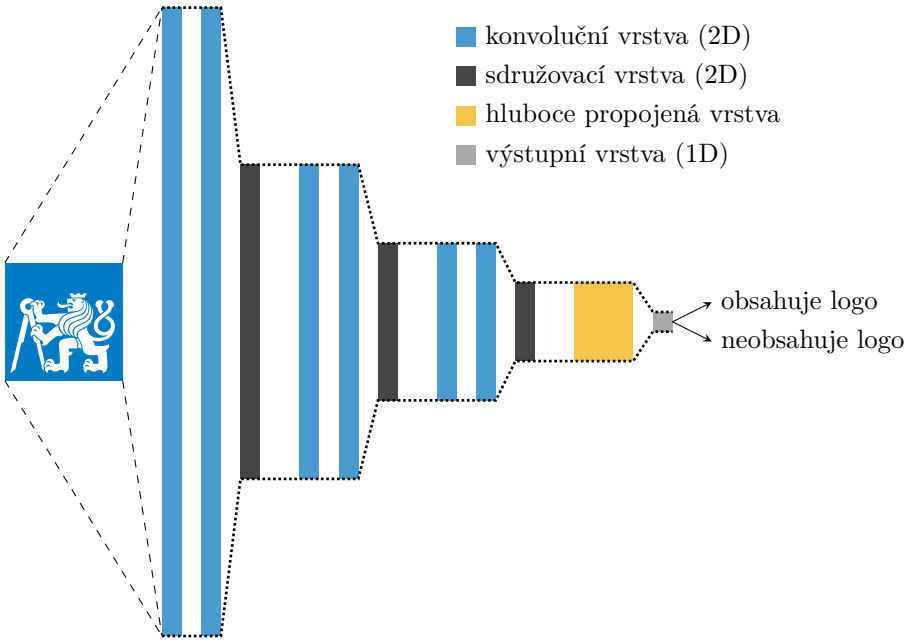
Mezi další metody patří slovníkové metody. *Shapeletové* metody se zabývají pouze tím, zda se daný *shapelet* v časové řadě nachází nebo nenachází [36]. Ve chvíli, kdy časové řady neodlišuje pouze existence nějakého vzoru, ale i frekvence jeho výskytu v časových řadách, tak přístup *shapelet* metod nejspíše selže [36]. Slovníkové metody se na rozdíl od *shapeletových* metod snaží nalézt i počet výskytů daného *shapeletu* [36, 37]. Mezi tyto metody patří i ROCKET a MINIROCKET [32] algoritmy, které budou podrobněji popsané v následující kapitole.

1.2.4 *Deep Learning* a Konvoluční Neuronové Sítě

Ačkoliv ROCKET a ani MINIROCKET nepatří mezi metody, které by využívaly *Deep Learning* či konvoluční neuronové sítě, tak si jsou s nimi v jistých ohledech podobné. Proto si i tyto metody zaslouží krátké představení.

Deep Learning, tedy hluboké neuronové sítě, a konkrétně Konvoluční Neuronové Sítě (CNN, *Convolutional neural networks*) si za dobu své existence našly mnoho úspěšných aplikací v mnoha odvětvích, hlavně při zpracování obrazu, respektive pro segmentaci, klasifikaci a rozpoznávání [36, 29, 39].

Hluboké konvoluční neuronové sítě se hojně využívají pro zpracování obrazu, se kterým časové řady sdílejí podobnost ve struktuře zpracovávaných dat, která se liší hlavně svou dimenzí. Je tedy možné využít podobný princip jako u zpracování obrazu, kdy konvoluce v případě časových řad probíhá dle toku času v jedné dimenzi pro zachycení časových (temporálních) charakteristik, zatím co při zpracování obrazu konvoluce probíhá podél dvou dimenzí pro zachycení prostorových charakteristik [29, 39].



Obrázek 1.2 Ukázka možné architektury konvoluční neuronové sítě

Konvoluční neuronové sítě se povětšinou skládají z vícero vrstev. Zpravidla se jedná o konvoluční vrstvy, které aplikují různé filtry na vstupní data, tedy ve kterých probíhá samotná konvoluce, sdružovací vrstvy, které sníží měřítko aby se předešlo *overfittingu*, plně propojenou, která hledá vzory a kombinace vzorů v nalezených příznacích v předchozích vrstvách, a výstupní vrstvu, která na základě naučených příznaků a své aktivační funkce z předchozích vrstev přinese finální výsledek. Je možné se ale setkat s mnoha rozdílnými architekturami, které využívají i jiné druhy vrstev [29]. Ukázku možné architektury konvoluční neuronové sítě je možné vidět na Obrázku 1.2. Co to je vlastně konvoluce a jak funguje, je rozebrané podrobněji v následující kapitole 2.1.2.

1.2.5 Další metody

Pro klasifikaci časových řad je možné využít i mnoho dalších metod. Mezi *state of the art* metody patří ale hlavně metody, které kombinují vícero přístupů a modelů v jeden (tzv. *model ensembles*) [36]. Každý z obsažených modelů je zpravidla natrénován odděleně na ostatních a o výsledné třídě modely následně „hlasují“ a nejčastěji zastoupená či nejvíce vážená (například dle přesnosti při trénování) třída pak vyhrává, jako tomu je například v HIVE-COTE [34]. Další možností je, že se nad výsledky, které tyto modely poskytnou, natrénuje nějaký další model (například *Random Forest*), který výslednou třídu klasifikace určí sám.

1.3 Struktura práce

Tato práce se skládá celkem z 5 kapitol vyjma nulté „kapitoly“ Motivace (viz), ve které jsou popsány osobní důvody a pozadí vedoucí k vypracování této práce. Následuje Úvod, tedy Kapitola 1, který poskytuje základní kontext o celé práci. Začíná vysvětlením toho, co je klasifikace časových řad, poté se seznamuje čtenáře v Sekci 1.2 s již dostupnými metodami pro klasifikaci časových řad, které jsou dále rozebrány podle svých hlavních charakteristik a principů. Mezi tyto metody metody patří založené na vzdálenostech, metody založené na *shapeletech*, slovníkové metody, *Deep Learning* a Konvoluční Neuronové Sítě, a další metody.

Kapitola 2, „Analýza existujících nástrojů a technologií“, se zabývá analýzou současných nástrojů a technologií používaných v oblasti klasifikace časových řad a obsahuje podrobný rozbor algoritmů *k-Nearest Neighbors* s *Dynamic Time Warping* a MiniRocket. Následně se zabývá jazyky, které jsou v této oblasti běžně používány, konkrétně jazykem Python a jazykem Julia, ke které je implementována praktická část této práce. Tato kapitola také přibližuje nástroje a *frameworky*, se kterými je možné se setkat v dalších kapitolách.

Kapitola 3, „Implementace“, se zaměřuje na praktickou stránku této práce a popisuje implementaci algoritmů KNN s DTW a MiniRocket, respektive jejich zakomponování do rozhraní MLJ.jl, které je podrobně rozebráno včetně ukázek zdrojového kódu. Následuje porovnání implementací těchto algoritmů z balíčku sktime s implementací v této práci, Dále je detailně popsána struktura balíčku TsClassification a jeho komponent, který v rámci této práce vznikl.

Předposlední Kapitola 4 popisuje prostředí, v rámci kterého probíhaly testy a porovnání algoritmů *k-Nearest Neighbors* s *Dynamic Time Warping* a MiniRocket s implementací v Python balíčku sktime, testovací *datasety*, které byly využity pro porovnání výkonu implementace v jazyce Python z sktime a té z této práce v jazyku Julia.

Poslední kapitolou je „Závěr“ (Kapitola 5), ve kterém lze najít shrnutí této práce a výsledků v ní dosažených.

Kapitola 2

Analýza existujících nástrojů a technologií

Cílem této kapitoly je hlouběji se seznámit s implementovanými algoritmy *k-Nearest Neighbors* s *Dynamic Time Warping*, zkráceně KNN s DTW, metrikou a algoritmem MiniRocket a následně porovnat programovací jazyky Julia a Python a popsat nejrozšířenější balíčky, které se zaměřují na práci s časovými řadami. Mezi ně patří Python balíček sktime a balíček MLJ.jl, který poskytuje rozhraní pro modely strojového učení v jazyku Julia, a vysvětlit jejich funkcionalitu.

2.1 Algoritmy

V této sekci podrobněji popíši implementované algoritmy *k-Nearest Neighbors*, *Dynamic Time Warping* a MiniRocket spolu se svým předchůdcem Rocket [32] a vysvětlím jejich funkčnost a rozdíly.

2.1.1 KNN s DTW

KNN s DTW je kombinace dvou algoritmů strojového učení, *k-Nearest Neighbors* a *Dynamic Time Warping*, jež je využívána především pro klasifikaci časových řad. Účinnost DTW spočívá v tom, že DTW dokáže zohlednit potenciální časové nesoulady v datech, například chybné zarovnání či různé doby trvání jednotlivých úseků časových řad, a poskytuje tedy robustnější řešení ve srovnání s použitím KNN s tradičními metrikami vzdálenosti, jako je třeba Euklidovská vzdálenost. Při klasifikaci časových řad se pak zpravidla používá KNN kde $k = 1$ (1-NN) společně s DTW [36], tedy že se hledá taková časová řada z již známých časových řad, která je dle DTW nejpodobnější (nejblíže) nové časové řadě, dle které se pak určí třída nové časové řady.

2.1.1.1 *K-Nearest Neighbors*

Algoritmus k-Nearest Neighbors je univerzální a jednoduchá technika „supervizovaného strojového učení“, která se používá pro klasifikační a regresní úlohy. Od svého vzniku v roce 1951 [40], respektive svého rozšíření v roce 1967 [41] do podoby, ve které je znám dnes, byl široce aplikován v různých oblastech, od rozpoznávání vzorů [42] po doporučování obsahu uživatelům [43] a používá se často jako *baseline* model pro úlohy klasifikace. Algoritmus KNN je založen na myšlence, že objekty s podobnými vlastnostmi mají tendenci být seskupovány dohromady, tedy že objekty, které jsou si podobné, respektive blízké, budou sdílet stejnou klasifikační třídu.

Funguje tak, že identifikuje k nejpodobnějších (nejbližších) datových bodů k dotazovanému bodu a předpovídá třídu nebo hodnotu na základě většinového hlasu, resp. (váženého) průměru těchto sousedů [41]. Na rozdíl od většiny metod pro „supervizované učení“, kde je fáze trénování modelu zpravidla řádově náročnější než výpočet predikcí, je KNN neparametrický model [41], tedy nevytváří žádné předpoklady o základním rozdělení dat a data nijak dále neabstrahuje (narozdíl od rozhodovacích stromů) a trénovací data jsou sama o sobě naučeným modelem.

Aby KNN mohlo změřit míru podobnosti jednotlivých prvků, vyžaduje metriku (zvanou také „vzdálenost“), což je zobrazení, respektive funkce, na množině dat $\mathcal{X}$

$$f : \mathcal{X} \times \mathcal{X} \to [0, +\infty)\,,$$

která kvantifikuje (ne)podobnost dvou prvků.

Taková metrika by měla splňovat určité podmínky $\forall x, y, z \in \mathcal{X}$ [44]:

Podmínka nezápornosti $f(x, y) \geq 0$
Vzdálenost mezi dvěma libovolnými prvky musí být vždy větší nebo rovna nule.

Podmínka identity $x = y \Leftrightarrow f(x, y) = 0$
Vzdálenost mezi dvěma identickými body musí být nulová.

Podmínka symetrie $f(x, y) = f(y, x)$
Vzdálenost mezi prvkem A a a prvkem B musí být jako vzdálenost mezi prvkem B a prvkem A.

Podmínka trujůhelníkové nerovnosti $f(x, y) \leq f(x, z) + f(z, y)$
Vzdálenost mezi dvěma prvky je vždy menší nebo rovna součtu vzdáleností mezi těmito dvěma a dalším prvkem.

Volba vhodné metriky je zásadní součástí procesu použití KNN, protože přímo ovlivňuje výsledky algoritmu.

Mezi běžně používané metriky v KNN patří například Euklidovská vzdálenost f_e

$$f_e(x, y) = \|x - y\|_2 = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}\,,$$

která měří vzdálenost dvou prvků x a y

$$x = (x_1, x_2, \ldots, x_n) \in \mathbb{R}^n, y = (y_1, y_2, \ldots, y_n)$$

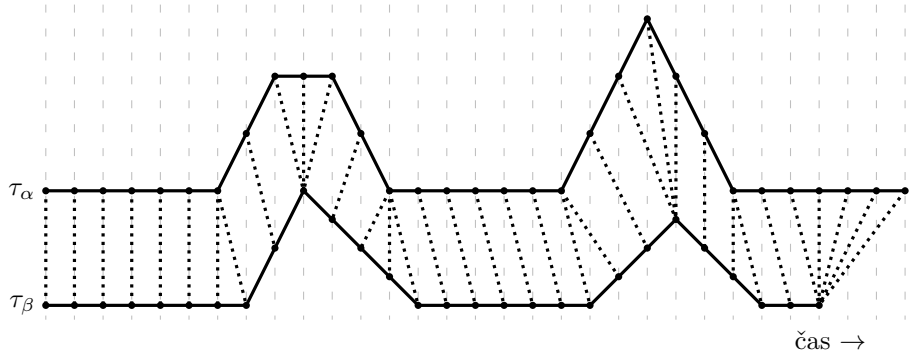
v Euklidovském prostoru [44]. Ne všechny používané metriky ale splňují všechny podmínky, mezi takové patří například Dynamic Time Warping. Výběr vhodné metriky záleží vždy na řešeném problému.

2.1.1.2 *Dynamic Time Warping*

Dynamic Time Warping je algoritmus, původně zamýšlený pro detekci mluveného slova [45], pro měření odlišnosti dvou sekvencí, které se mohou lišit v délce a rychlosti. Cílem algoritmu je změřit podobnost dvou sekvencí tak, že najde jejich optimální zarovnání, při kterém je suma maximálních vzdáleností prvků takto zarovnaných sekvencí minimální [46]. DTW je možné zapsat jako

$$\mathrm{DTW}(\tau_\alpha, \tau_\beta) \to [0, +\infty)\,,$$

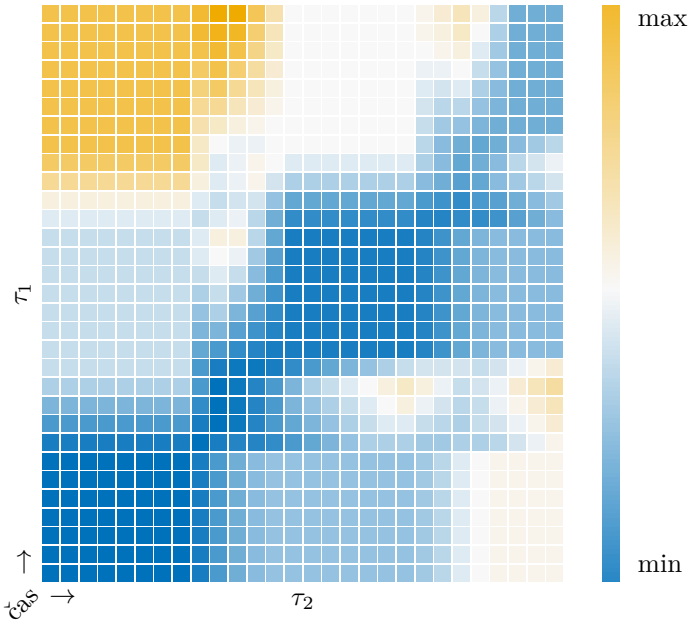
kde $\tau_\alpha \in \mathbb{R}^m$ a $\tau_\beta \in \mathbb{R}^n$, jsou časové řady a $m, n \in \mathbb{N}^+$ je jejich délka, kde $\mathbb{N}^+$ přirozená čísla bez nuly, respektive celá kladná čísla.



Obrázek 2.1 Ukázka principu fungování Dynamic Time Warping algoritmu, respektive možného mapování (přerušovaná čára) indexů časových řad τ_α a τ_β. Pro názornost jsou časové řady vykresleny spojitou čarou, ačkoliv DTW pracuje pouze s diskrétními sekvencemi.

DTW docílí optimálního zarovnání pomocí série deformací (*warping*) dvou posloupností tak, aby se minimalizovala suma vzdáleností mezi jednotlivými prvky časových řad. Časové řady řady jsou reprezentovány jako dvě pole, přičemž každý prvek pole reprezentuje příznaky (naměřené hodnoty) v určitém časovém bodu. Algoritmus pracuje tak, že pomocí dynamického programování sestrojí matici vzdáleností (*cost matrix*), viz Obrázek 2.2, která představuje vzdálenosti mezi všemi dvojicemi bodů obou vstupních sekvencí a následně hledá cestu s nejmenší kumulativní vzdáleností skrz tuto matici. Vybraná cesta představuje optimální zarovnání mezi oběma sekvencemi [45, 46]. Asymptotická časová složitost DTW algoritmu je tedy $O(mn)$, respektive $O(n^2)$ [47].

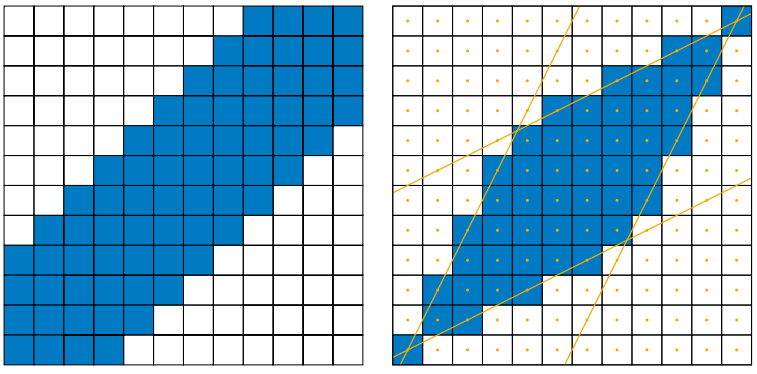
V mnoha případech optimální zarovnání (*warping*) časových řad, z pohledu DTW, ale často nemusí být „optimální“ z pohledu řešeného problému. Navíc časová složitost DTW $O(n^2)$ může vést k problémům se škálováním v případě větších *datasetů* [47]. DTW také trpí v případě, když si začátek či konec dvou (různě dlouhých) časových řad není podobný, ať už kvůli nevhodnému oříznutí měřeného úseku či kvůli vlastnostem naměřených dat, protože se princip fungování algoritmu neumožňuje s takovými případy vhodně vypořádat. V takovém případě může být vzdálenost jinak podobných časových



Obrázek 2.2 *Heatmap* hodnot z matice vzdáleností DTW pro výše uvedené časové řady z Obrázku 2.1 značící vzájemné relativní vzdálenosti mezi prvky. Tmavě modrá barva značí nejnižší, oranžová naopak nejvyšší vzdálenost.

řad nepřiměřeně vysoká, protože konce těchto řad takto disproporčně přispějí k velké výsledné vzdálenosti [48]. Dalším problémem může být také, když DTW v rámci svého „optimálního“ zarovnání velmi krátký úsek jedné časové řady zarovná na disproporčně větší úsek řady druhé [47].

Tyto problémy lze řešit různými přístupy. Mezi nejčastější patří zavedení různých omezení prostoru možného *warpingu* ve formě různých pásem. Tím se zmenší prostor, ve kterém je nutné počítat vzdálenosti mezi body jednotlivých časových řad. Mezi nejčastěji používané patří Sakoe–Chiba pásmo [45] a Itakurův parallelogram [49], jejichž ukázku je možné najít na Obrázku 2.3.



Obrázek 2.3 Sakoe–Chiba pásmo s $window = 3$ a Itakurův parallelogram se $slope = 2.0$ pro 2 časové řady o délce $n = 12$. Prostor, na který by bylo DTW omezeno, je vyznačen modře.

Sakoe–Chiba pásmo je pásmo podél diagonály matice vzdáleností, u kterého je mož-

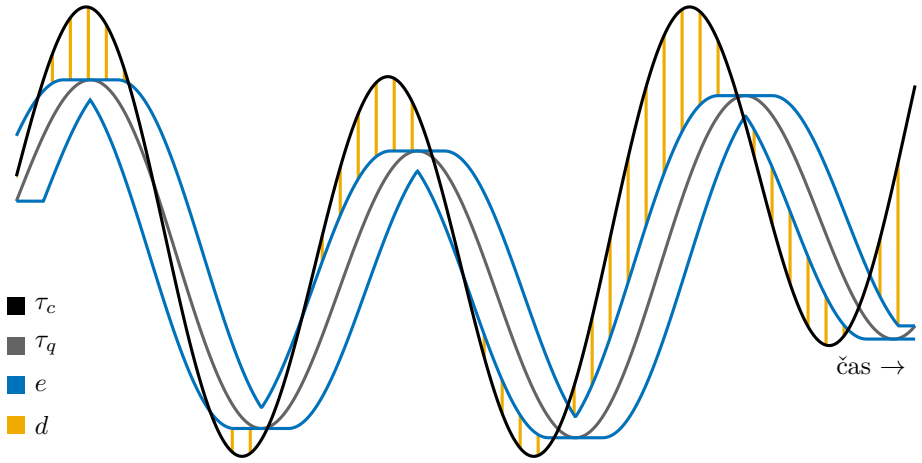
né nastavit jeho šířku *window* W (někdy také okno, respektive *window*). Značení $\widehat{N}$, se kterým je možné se setkat nejen v následujících rovnících, symbolizuje množinu $\{1, 2, \ldots, N\}$. Množinový zápis určující, na které indexy matice bude omezeno prohledávání, může vypadat následovně:

$$\text{sakoechiba} = \{(i, j) \in \widehat{m} \times \widehat{n} \mid W \geq |i - j|\}$$

Itakurův rovnoběžník (*Itakura parallelogram*) pracuje s myšlenkou, že warping je důležitý převážně ve středu časové řady a ne na jejím počátku a konci, kterou aplikuje tak, že místo pouhého pásma vytváří parallelogram, definovaný parametrem *slope* S, který určuje jeho sklon. Množinově lze Itakurův rovnoběžník zapsat následovně:

$$\text{itakura} = \left\{ (i, j) \in \widehat{m} \times \widehat{n} \;\middle|\; \begin{array}{l} j \leq \max(S \cdot \frac{n}{m-1}, 1) \cdot i \\ j \geq \min(S^{-1} \cdot \frac{n-1}{m}, 1) \cdot i \\ j \geq \max(S \cdot \frac{n}{m-1}, 1) \cdot (i - m) + n \\ j \leq \min(S^{-1} \cdot \frac{n-1}{m}, 1) \cdot (i - m) + n \end{array} \right\}$$

Další přístup pro urychlení vyhledávání k nejbližších sousedů je nálezt spodní odhad vzdálenosti porovnávaných časových řad ještě před tím, než je nutné spočítat přesnou vzdálenost pomocí výpočetně náročného DTW, algoritmem, který je výpočetně efektivnější než DTW. Jedním z těchto algoritmů je LB_Keogh [50], jenž je znázorněn na Obrázku 2.4.



Obrázek 2.4 Ilustrace funkce algoritmu LB_Keogh. τ_c značí obalenou časovou řadu, τ_q porovnávanou časovou řadu, e obal (horní a spodní meze) a d zvýrazňuje rozdíly τ_q a obalu e.

Princip tohoto algoritmu stojí na tom, že se porovnává obal e dotazované časové řady τ_c, který vzniká pomocí posuvného okna o daném poloměru r. Minimum v tomto okně je je bráno jako spodní mez obalu e_L, maximum z okna je naopak bráno jako horní mez obalu e_U proti právě porovnávané řadě τ_q. Jako výsledný spodní odhad se pak počítá vzdálenost prvků, které se nachází mimo obal, porovnávané časové řady τ_q, která se již nachází v *datasetu*, a mezí obalu [50]. Rovnice definující spodní odhad vzdálenosti

LB_Keogh vypadají následovně:

$$\tau_c, \tau_q \in \mathbb{R}^n; r \in \mathbb{N}$$

$$e_U(\tau) = \{\max(\tau_1 : \tau_{1+r}), \ldots, \max(\tau_{i-r} : \tau_{i+r}), \ldots, \max(\tau_{n-r} : \tau_n)\}$$

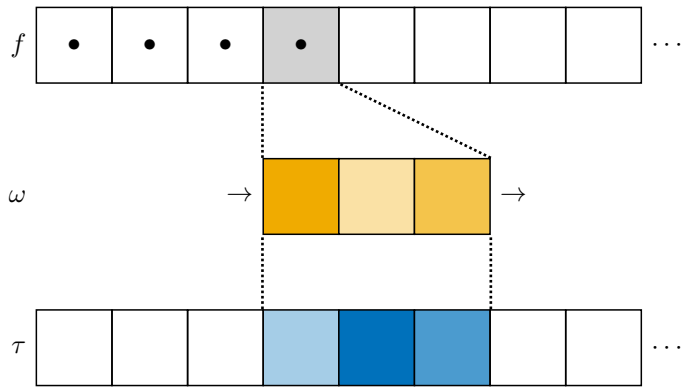
$$e_L(\tau) = \{\min(\tau_1 : \tau_{1+r}), \ldots, \min(\tau_{i-r} : \tau_{i+r}), \ldots, \min(\tau_{n-r} : \tau_n)\}$$

$$LB\_Keogh(\tau_q, L \coloneqq e_u(\tau_c), U \coloneqq e_l(\tau_c)) = \sqrt{\sum_{i=1}^{n} \begin{cases} (\tau_{q_i} - U_i)^2 & \text{pokud } \tau_{q_i} > U_i \\ (\tau_{q_i} - L_i)^2 & \text{pokud } \tau_{q_i} < L_i \\ 0, & \text{jinak} \end{cases}}$$

Tím je možné přeskočit výpočet DTW pro řady, u kterých je tento odhad vyšší, než již nalezené nejlepší známé vzdálenosti. S využitím těchto technik je možné DTW v případě velkých *datasetů* prohlásit za *v podstatě lineárně složitý* algoritmus [47].

2.1.2 MINIROCKET

Pro hlubší pochopení ROCKET i MINIROCKET algoritmu bude vhodné představit některé z často vyskytujících se pojmů.



Obrázek 2.5 Ukázka $1D$ konvoluce. f značí *feature map*, ω značí filtr (*kernel*) a τ značí časovou řadu, nad kterou probíhá konvoluce

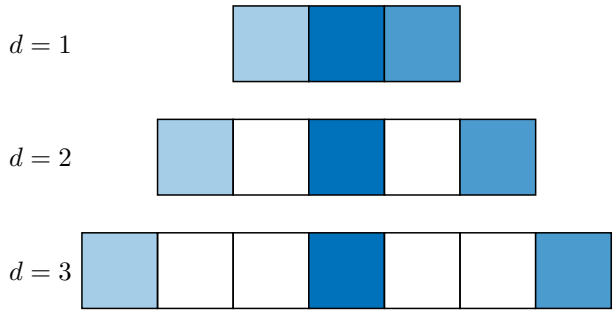
Prvním z nich je konvoluce. Konvoluci je možné si představit jako posuv okna, respektive *kernelu*, nad časovou řadou [39] jako například na Obrázku 2.5. V každém kroku se provede skalární součin vah *kernelu* a prvků časové řady nacházejících se v tomto okně, čímž postupně vznikne tzv. *feature map*, která obsahuje takto zvýrazněné příznaky.

Konvoluci je také možné zapsat jako následující matematickou operaci

$$C_i = \omega \cdot \tau_{i-\lfloor \ell_\omega/2 \rfloor : i+\lfloor \ell_\omega/2 \rfloor}, \forall i \in 1, 2, \ldots, \ell_\tau \,,$$

kde C_i označuje výsledek konvoluce v čase i. Konvoluce probíhá jako skalární součin, značený $\cdot$, kernelu $\omega \in \mathbb{R}^{\ell_\omega}$ a subsekvence časové řady $\tau \in \mathbb{R}^{\ell_\tau}$. Subsekvence je dána rozsahem a až b značeným notací $a{:}b$, kde $a, b \in \widehat{\ell_\tau}$ jsou indexy prvků časové řady, [39].

Další je „dilatace“ *kernelu*. *Dilatací kernelu* se rozumí metoda, která dokáže rozšířit *kernel* vkládáním děr (či mezer) do původního *kernelu*. Míru *dilatace* určuje parametr d,



Obrázek 2.6 Ukázka *dilatace kernelu*. d značí míru dilatace *kernelu*, bílé buňky značí mezery, respektive nulové hodnoty, vložené do *kernelu* o které byl *kernel* roztažen

který je možné si vyložit jako $d-1$ přeskočených prvků mezi samotnými prvky původního kernelu (s *dilatací* $d = 1$). Jak vypadá dilatace, je možné najít na Obrázku 2.6. *Dilatace* umožňuje pokrýt větší rozsah vstupních dat a tedy i nacházet vzory na větším rozsahu, aniž by bylo nutné agregovat či jinak slučovat hodnoty ve vstupních datech [51].

Rocket i MiniRocket nejsou klasifikátory, ale pouze metody pro efektivní extrakci a reprezentaci příznaků ze vstupních dat. Vstupní data tedy pouze *transformují*, respektive z nich za pomocí konvoluce s mnoha náhodnými filtry, respektive *kernely*, vytvoří nový soubor příznaků, které je možné předat už jinému, více tradičnímu, klasifikátoru. Ten pak provede finální klasifikaci [32, 31].

2.1.2.1 ROCKET

Algoritmus Rocket, *RandOm Convolutional KErnel Transform*, aplikuje oproti ostatním SOTA (*state of the art*) algoritmům nový přístup, kdy využívá, podobně jako Konvoluční Neuronové Sítě (CNN, *Convolutional Neural Networks*), konvoluční filtr (*kernel*) pro extrakci příznaků ze vstupních dat (jednorozměrných časových řad). Na rozdíl od konvolučních neuronových sítích ale Rocket nevyužívá žádné další (například sdružovací) vrstvy, nebuduje žádnou neuronovou síť a tedy ani nevyžaduje učení se vah filtrů zpětnou propagací (*back propagation*). Místo toho Rocket využívá soubor náhodných filtrů (*kernelů*), které má již předpočítané v paměti, a aplikuje je na časovou řadu, čímž vygeneruje soubor příznaků, nad kterými stačí provést klasifikaci některou z tradičních metod, respektive nějakým lineárním klasifikátorem, jako je například logistická či hřebenová regrese [32]. To proto, že samotný Rocket neprovádí klasifikaci jako takovou a pouze transformuje vstupní data. V případě použití logistické regrese pro klasifikaci je možné říct, že se Rocket chová podobně, jako jednovrstvá konvoluční síť s náhodnými *kernely*, kdy generované příznaky tvoří vstup pro softmax funkci. V praxi se však pro všechny *datasety*, vyjma těch největších, používá klasifikátor s hřebenovou regresí [32].

Rocket se proti typickým konvolučním neuronovým sítím liší v několika ohledech:

Používá velké množství *kernelů* Protože existuje pouze jedna konvoluční „vrstva“ a váhy jednotlivých *kernelů* se nijak neučí (jsou náhodné a již předem vypočítané), jsou náklady na výpočet samotné konvoluce velmi nízké a je možné tedy použít velké množství různých *kernelů* bez většího dopadu na rychlost výpočtů [32].

Používané *kernely* jsou vysoce rozmanité Oproti konvolučním neuronovým sítím,

kde je běžné, že skupiny jader mají stejnou velikost, *dilataci* a výplň, ROCKET používá náhodnou délku, *dilataci*, výplň, váhy a zkreslení pro každý *kernel* [32].

Využívá různých *dilatací kernelů* V porovnání s využitím *dilatace* v konvolučních neuronových sítích, kde *dilatace* roste exponenciálně s hloubkou, respektive počtem *dilatovaných* konvolučních vrstev neuronové sítě, ROCKET staví svou vysokou přesnost na náhodném výběru *dilatace* každého z *kernelů*, čímž je možné zachytit vzory a příznaky na mnoha škálách a frekvencích [32].

Sdružování příznaků skrz výpočet proporce pozitivních hodnot ROCKET jako svoji sdružovací metodu používá nejen maximální hodnotu z vygenerovaných souborů příznaků (přirovnatelnou ke globálnímu *max poolingu*, tedy sdružení hodnot ze souboru příznaků do nejvyšší hodnoty, která se v něm nachází), ale i proporci pozitivních hodnot (*ppv*, *proportion of positive values*), což umožňuje klasifikátoru posoudit i množství výskytu (prevalenci) daného příznaku v časové řadě. Tato vlastnost je hlavním důvodem vysoké přesnosti algoritmu [32].

ROCKET se skládá z několika zásadních částí či kroků.

Kernely Mezi tyto části patří, v podstatě ve všech směrech náhodné, *kernely* (značené ω). Jejich hodnoty, rozsahy a pravděpodobnostní rozložení autoři vybrali experimentálně tak, aby algoritmus dosahoval co nejvyšší klasifikační přesnosti měřené na osmdesáti pěti *datasetech* složených z jednorozměrných časových řad [32] dostupných v archivu časových řad UCR (University of California, Riverside) [52]. Více informací o archivu je možné nalézt v Kapitole 2.3.3. Je důležité zmínit, že byly vybrány s předpokladem, že časové řady byly normalizovány tak, aby měly průměr svých hodnot roven nule a směrodatná odchylka byla rovna nule. Kernely používané ROCKET algoritmem vypadají následovně:

Délka – ℓ_ω Délka *kernelu* je vybrána náhodně z hodnot $7, 9, 11$ se stejnou pravděpodobností. Takové *kernely* jsou pak kratší než většina časových řad, se kterými se autoři setkali v UCR archivu časových řad [32]

Váhy – W_{ℓ_ω} Váhy (myšleno hodnoty samotného *kernelu*) jsou vybírány z normálního rozdělení $\forall w \in W; w \sim \mathcal{N}(0, 1)$ a jejich průměr je centrován v nule. Tím je myšleno, že z každé váhy *kernelu* odečten průměr všech vah *kernelu* $\omega = W - \overline{W}$ [32].

***Bias* – b** *Bias*, nebo také zkreslení či zaujatost, je vybrán z uniformního rozdělení $b \sim \mathcal{U}(-1, 1)$. Protože ve výsledném setu příznaků se vyhodnocují pouze kladné hodnoty díky metodám *max pooling* a *ppv*, tak *bias* takto dokáže, v případě dvou jinak si podobných *kernelů*, zvýraznit rozdílné aspekty posunutím hodnot před jejich vyhodnocením těmito metodami [32].

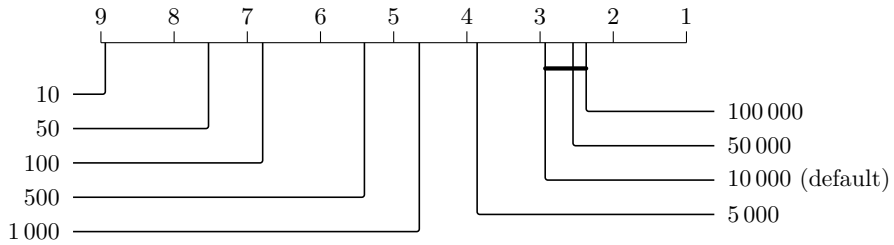
Míra *dilatace* – d *Dilatace* je brána z exponenciální stupnice $d \sim \lfloor 2^x \rfloor; x \sim \mathcal{U}(0, A)$, kde $A = \log_2 \frac{\ell_\tau - 1}{\ell_\omega - 1}$ a ℓ_τ značí délku časové řady. Rozdílná míra *dilatace kernelů* tedy zajišťuje, že délka *kernelů* může sahat od ℓ_ω až po délku celé časové řady. V případě dvou jinak si podobných kernelů je takto možné zachytit podobný vzor či příznak nehledě na jeho frekvenci či škálu [32].

Padding Pro každý *kernel* je náhodně rozhodnuto, zda se při konvoluci použije *padding*, respektive výplň či zarovnání. V případě, že je použit *padding*, je na začátek i konec každé časové řady přidán řetězec nul, čímž je možné „prostřední“ prvek *kernelu* vždy zarovnat s každým prvkem časové řady. Bez *paddingu* se *kernel* nedokáže „zaměřit“ na prvky na okrajích časové řady a hledá příznaky hlavně v prvcích, které jsou blíže ke středu časové řady. S *paddingem* naopak zvládne *kernel* nacházet příznaky i na začátku či konci časových řad [32].

Transformace Dalším krokem je transformace, tedy proces extrakce příznaků ze vstupních dat. Transformace probíhá tak, že proběhne konvoluce každého *kernelu* s každou časovou řadou, čímž vždy vznikne soubor příznaků C. Konvoluce samotná je pohyblivý skalární součin dilatovaného *kernelu* ω_d s mírou *dilatace* d a o délce ℓ_ω a časové řady τ, který je možné zapsat jako následující rovnici, kde i je aktuální pozice v časové řadě [32].

$$C_i = \omega_d * \tau_i = \sum_{j=1}^{\ell_\omega} \omega_j \tau_{i+(j \cdot d)}$$

Rocket má jediný *hyperparametr*, kterým je počet použitých *kernelů* při transformaci. Ačkoliv je tento *hyperparametr* možné měnit, autoři, stejně jako u většiny ostatních vlastností algoritmu, experimentálně nalezli jako ideální hodnotu 10 000 kernelů, viz Obrázek 2.7, kdy algoritmus dosahuje ideálního poměru klasifikační přesnosti, rychlosti a nároků na paměť (které rostou lineárně s počtem kernelů) v porovnání s ostatními hodnotami, a nepředpokládá se, že by uživatel algoritmu toto číslo měnil [32].



Obrázek 2.7 Průměrné pořadí (umístění) algoritmu Rocket v *benchmarku* při různých počtech kernelů [32]

Diagram z Obrázku 2.7 se nazývá *Critical Difference diagram*, zkráceně CD diagram [53, 54], a slouží k porovnání vícero metod na základě jejich výsledků z vícero pozorování. V tomto případě je CD diagram tedy způsob znázornění přesnosti různých modelů nad vícero *datasety*. Pozice každého modelu na diagramu znázorňuje jeho průměrné pořadí napříč všemi výsledky pozorování, tedy v případě Obrázku 2.7 se Rocket s 10 000 *kernely* umístil se svou přesností průměrně třetí, zatím co se Rocket s 500 *kernely* pohyboval v průměru na 5. a 6. místě. Ačkoliv se Rocket s 100 000 *kernely* umístil zpravidla na lepší pozici jak Rocket s 10 000 a 50 000 *kernely*, jejich spojení tlustou čarou říká, že rozdíly mezi výsledky těchto modelů nebyly statisticky významné [55].

Rocket tyto nezpracované soubory příznaků nijak neuchovává a rovnou z nich vypočítá oba výsledné příznaky, maximální hodnotu ze souboru příznaků (srovnatelné

s globálním *max pooling* používaném ve sdružovacích vrstvách konvolučních neuronových sítí) a *ppv*, tedy proporci pozitivních hodnot (*proportion of positive values*), která dle autorů ROCKET produkuje významně vyšší přesnost následné klasifikace než ostatní metody včetně globálního sdružování maximální hodnotou i průměrem [32]. Pro n *kernelů* tedy ROCKET vytvoří finální soubor příznaků o $2n$ příznacích. Symbolem $\#X$, se kterým je možné setkat v této práci, je značena kardinalita, tedy počet prvků, množiny X.

$$\max(C) = \max\{C_i \mid i \in \{1, \ldots, \ell_\tau\}\}$$
$$\text{ppv}(C) = \frac{\#\{C_i \mid C_i > 0, i \in \{1, \ldots, \ell_\tau\}\}}{\ell_\tau}$$

Klasifikace Posledním krokem je klasifikace, kterou ROCKET sám o sobě neprovádí, protože ROCKET jako takový pouze extrahuje příznaky ze vstupních dat. Proto je nutné ROCKET spojit s nějakým klasifikátorem. ROCKET je možné použít spolu s v podstatě jakýmkoliv klasifikátorem, autoři ale doporučují využít „lineární klasifikátory“, jako je například hřebenová či logistická regrese spolu se stochastickým gradientním sestupem [32], které se vyznačují tím, že zvládají efektivně pracovat i s velkým počtem příznaků.

2.1.2.2 MINIROCKET

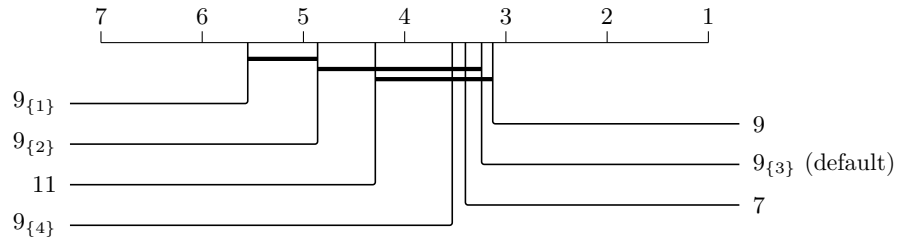
MINIROCKET, celým názvem *MINImally RandOm Convolutional KErnel Transform*, je evoluce algoritmu ROCKET, která má za svůj cíl minimalizaci „náhodnosti“, respektive dosažení vyšší „deterministočnosti“ algoritmu a vyšší efektivity (rychlosti) než ROCKET při zachování podobné či vyšší klasifikační přesnosti. MINIROCKET dosahuje obdobné klasifikační přesnosti jako ROCKET, ale 19krát až 75krát rychleji [31]. Těchto výsledků dosahuje několika způsoby.

Prvně se autoři soustředili na minimalizaci náhodnosti algoritmu. Na rozdíl od ROCKET, který má své parametry (váhy *kernelů*, jejich dilataci, délku, …) čistě náhodné, viz Sekce 2.1.2.1, jsou hodnoty všech parametrů *kernelů* předem známé, viz Tabulka 2.1, což umožňuje implementovat řadu optimalizací, ze kterých pak plyne i zrychlení MINIROCKET.

	ROCKET	MINIROCKET
délka *kernelů*	$\{7, 9, 11\}$	9
váhy *kernelů*	$\mathbb{N}(0, 1)$	$\{-1, 2\}$
bias	$\mathbb{U}(-1, 1)$	dle výstupu konvoluce
dilatace	náhodně	fixní
padding	náhodně	předem určená
příznaky	ppv, max	ppv

Tabulka 2.1 Porovnání parametrů ROCKET a MINIROCKET

Váhy a délka *kernelů* Na rozdíl od ROCKET, který využívá náhodné váhy *kernelů*, se MINIROCKET limituje na pouhé dvě váhy, -1 a 2, viz Tabulka 2.1. Tato změna vychází z výsledků testování ROCKET, kde autoři porovnávali náhodné váhy s váhami vybíranými náhodně z množiny celých čísel $\{-1, 0, 1\}$. V *benchmarku*, zmíněném již v Sekci 2.1.2.1, ROCKET s *kernely* s těmito váhami dosahoval v podstatě totožného výkonu. V případě MINIROCKET výběr vah -1 a 2 pak později výrazně usnadní výpočet konvoluce [31].



Obrázek 2.8 Průměrné pořadí (umístění) algoritmu MINIROCKET v *benchmarku*, zmíněném již v Sekci 2.1.2.1, při různých délkách a různých podmnožinách kernelů [31]

Nároky na výpočet konvoluce všech *kernelů* stoupají exponenciálně s jejich délkou a počtem možných (celočíselných) vah. V případě kernelů 2 možných vah s délkou 3 existuje pouze 2^3, tedy 8 možných *kernelů*. V případě délky 9 jejich počet vzroste na $2^9 = 512$ a s třemi možnostmi a délkou 9 to je již přes 19 tisíc možných kernelů. Minimalizovat počet různých *kernelů*, které algoritmus používá, je tedy velmi důležité pro dosažení co nejvyšší efektivity [31].

MINIROCKET používá pouze dvě různé váhy, $\alpha = -1$ a $\beta = 2$. Každý kernel, který algoritmus použije, je tedy možné charakterizovat dle počtu α či β vah obsažených v takovém kernelu, jak je vidět na těchto ukázkách:

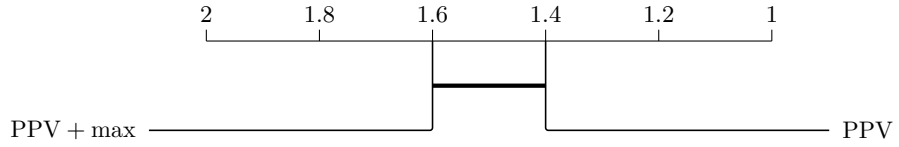
$$
\begin{aligned}
\#\beta = 0 \implies & [\alpha, \alpha, \alpha, \cdots, \alpha, \alpha, \alpha] \\
\\
\#\beta = 2 \implies & [\alpha, \alpha, \alpha, \cdots, \alpha, \beta, \beta] \\
& [\alpha, \alpha, \alpha, \cdots, \beta, \alpha, \beta] \\
& \qquad \vdots \\
& [\beta, \alpha, \beta, \cdots, \alpha, \alpha, \alpha] \\
& [\beta, \beta, \alpha, \cdots, \alpha, \alpha, \alpha],
\end{aligned}
$$

I proto se autoři MINIROCKET, na základě *benchmarku*, viz Sekce 2.1.2.1, viz Obrázek 2.8, rozhodli použít pouze subset *kernelů* typu „$9_{\{3\}}$“. Symbolem $P_{\{Q\}}$ je značena sada *kernelů* délky P, které obsahují Q hodnot β a $P - Q$ hodnot α. Tedy říká, že v případě $9_{\{3\}}$ je použito celkem pouze $\binom{9}{3} = 84$ z 512 možných *kernelů* délky 9, které se skládají

z 3 hodnot β a 6 hodnot α:

$$\begin{aligned}\#\beta = 3 \implies & [\alpha,\alpha,\alpha,\alpha,\cdots,\alpha,\beta,\beta,\beta] \\ & [\alpha,\alpha,\alpha,\alpha,\cdots,\beta,\alpha,\beta,\beta] \\ & \vdots \\ & [\beta,\beta,\alpha,\alpha,\cdots,\alpha,\alpha,\alpha,\beta] \\ & [\beta,\beta,\alpha,\alpha,\cdots,\alpha,\alpha,\beta,\alpha] \\ & \vdots \\ & [\beta,\beta,\alpha,\beta,\cdots,\alpha,\alpha,\alpha,\alpha] \\ & [\beta,\beta,\beta,\alpha,\cdots,\alpha,\alpha,\alpha,\alpha].\end{aligned}$$

Výběr vah α a β může být ale jakýkoliv, dokud je zachováno, že součet vah v *kernelu* je roven nule (tedy v případě *kernelů* $9_{\{3\}}$ musí platit $\beta = 2\alpha$). Tato podmínka zajistí, že *kernely* jsou náchylné pouze na relativní velikost hodnot časové řady, jinými slovy, výstup konvoluce je pak totožný nehledě na přičtení či odečtení konstanty k k prvkům časové řady. To znamená, že $\tau * \omega = (\tau \pm k) * \omega$, kde $*$ značí proces konvoluce a ω značí *kernel* [31].



Obrázek 2.9 Průměrné pořadí (umístění) algoritmu MiniRocket v *benchmarku*, viz Sekce 2.1.2.1, s použitím pouze *ppv* proti *ppv* spolu s max jako sdružovací funkce [31]

Bias Protože MiniRocket, dle *benchmarku* zmíněném již v Sekci 2.1.2.1, viz Obrázek 2.9, používá pouze *ppv* jako sdružovací metodu, a protože *bias* b je „čerpán“ z hodnot samotné konvoluce (jeho velikost tedy odpovídá velikostem výstupů konvoluce), je možné zapsat *ppv* s použitím *biasu* jako

$$\text{ppv}(C + b) = \frac{\#\{C_i + b \mid C_i > 0, i \in \widehat{\ell_\tau}\}}{\ell_\tau}$$

a ekvivalentně

$$\text{ppv}_b(C) = \frac{\#\{C_i \mid C_i > -b, i \in \widehat{\ell_\tau}\}}{\ell_\tau}.$$

V takovém případě je totiž možné považovat *ppv* za „v podstatě ekvivalentní“ k empirické distribuční funkci [31] a tedy velikost hodnot α a β je ve výsledku irelevantní (dokud je zachována podmínka, že součet všech vah *kernelu* je roven nule).

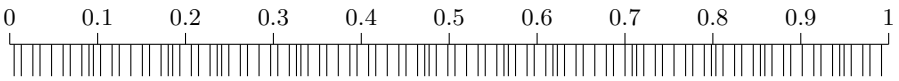
Pro každou kombinaci dilatace a *kernelu* je brán *bias* z kvantilů výstupu konvoluce s náhodně vybraným prvkem z (trénovacího) *datasetu*. Díky způsobu výběru *biasů* MiniRocket, na rozdíl od Rocket, nevyžaduje nijak normalizovaná vstupní data. Toto je v rámci MiniRocket algoritmu jediná náhodná část, proto také Mini v názvu MiniRocket znamená „minimally random“. MiniRocket je možné implementovat i plně deterministicky, kdy jsou *biasy* získány z konvolučního výstupu celého *datasetu*, který

musí být celý udržován v paměti, a ne pouze z jednoho náhodně vybraného vzorku (časové řady) [31]. Ačkoliv je pak v takovém případě algoritmus plně deterministický, tak dle měření autorů plně deterministická verze algoritmu nedosahuje vyšší přesnosti, vyžaduje téměř dvojnásobek volné paměti a zároveň dosahuje nižších rychlostí výpočtů, protože pro nalezení *biasů* je nutné hledat kvantily ne pouze ve výstupu konvoluce náhodně vybraného vzorku, ale ve všech časových řadách celého (trénovacího) *datasetu* [31].

Vybírané kvantily jsou brány z kvazi-náhodné posloupnosti, respektive „posloupnosti s nízkou diskrepancí“ (či „rozpytlností“) [31]. „Posloupnosti s nízkou diskrepancí“ vykazují rovnoměrnější rozložení prvků (na daném intervalu) svých podsekvencí, tedy že nevznikají „clustery“ (shluky) okolo některých hodnot, než náhodně generované posloupnosti a jsou deterministické (nejsou tedy nijak náhodné a ani pseudonáhodné) [56]. Posloupnost $(a_n)_{n=1}^{\infty}$, kterou používá MiniRocket pro získání kvantilů, je založena na zlatém řezu a vypadá následovně:

$$a_n = n \cdot \frac{\sqrt{5}+1}{2} \mod 1, n \in \mathbb{N}.$$

Rozložení prvků náhodně generované sekvence a sekvence, kterou používá MiniRocket je možné porovnat na Obrázku 2.10 a na Obrázku 2.11, kde lze zpozorovat nerovnoměrné rozložení, respektive zformované clustery hodnot a nebo naopak nerovnoměrně velké mezery, zatím co posloupnost založená na násobcích zlatého řezu modulo 1 má své hodnoty rozložené mnohem rovnoměrněji.



Obrázek 2.10 Hodnoty prvních 100 členů posloupnosti používané algoritmem MiniRocket

Obrázek 2.11 100 náhodně vygenerovaných hodnot z rovnoměrného rozložení $\mathbb{U}(0,1)$

Dilatace V případě MiniRocket spolu všechny *kernely* sdílí stejný soubor dilatací, na rozdíl od Rocket, kde každý *kernel* měl svou vlastní, náhodnou, dilataci. Možné míry dilatace jsou, podobně jako v případě Rocket, z množiny

$$D = \left\{ \lfloor 2^a \rfloor \;\middle|\; a \in \left[0, \log_2 \frac{\ell_\tau - 1}{\ell_\omega - 1}\right] \right\}$$

s tím rozdílem, že exponenty a již nejsou celočíselné a jsou rovnoměrně rozložené na intervalu $[0, \log_2 \frac{\ell_\tau - 1}{\ell_\omega - 1}]$. Celkový počet dilatací pro každý kernel kernel je pak určen poměrem počtu výsledných příznaků a počtu *kernelů*. Z toho plyne, že počet výsledných příznaků vypočítaných pomocí jedné dilatace klesá exponenciálně proti exponentu a míry dilatace, jinými slovy se s krátkými *kernely* (s nízkou mírou dilatace) spočítá exponenciálně více výsledných příznaků než s dlouhými *kernely* (s vysokou mírou di-

latace). S délkou časové řady ale stoupá i počet možných dilatací a v případě fixního počtu výsledných příznaků tím zároveň klesá počet příznaků vznikajících za pomoci dané dilatace a tím i efektivita algoritmu [31]. Autoři MiniRocket tedy přidali nový *hyperparametr* určující maximální počet různých exponentů na *kernel*. V základu je nastaven na hodnotu 32, která byla získána experimentálně v *benchmarku* (viz Sekce 2.1.2.1). Předpokládá se, že bude ale ponechána ve výchozí hodnotě, protože celkový vliv na přesnost algoritmu je velmi malý [31].

Padding Na rozdíl od Rocket je *padding* určen deterministicky tak, že polovina kombinací *kernelů* a dilatací *padding* nepoužívá, zatím co druhá polovina ho naopak používá [31]. Které kombinace *kernelů* a dilatací *padding* použijí je určené následovně

$$P = \left\{ (i, j) \in \widehat{84} \times \widehat{\#D} \;\middle|\; ((i \bmod 2) + j) \bmod 2 = 1 \right\},$$

kde P je množina dvojic indexů *kernelů*, značených i, a indexů dilatací, značených j.

Příznaky a sdružovací funkce MiniRocket používá pouze *ppv* jako svoji sdružovací funkci pro výstupy z konvoluce. Samotné *ppv* totiž podává lepší výsledky než při kombinaci s max sdružováním, viz Obrázek 2.9. Autoři se rozhodli max nenahradit jinou sdružovací metodou. Příznaků by v případě Rocket bez max sdružování tedy byla ve výsledku jen polovina (10 000), v případě MiniRocket tomu je ale jinak. MiniRocket nahrazuje *hyperparametr* určující počet kernelů novým *hyperparametrem* h určujícím počet příznaků. V případě MiniRocket je počet příznaků přímo závislý na počtu *kernelů*, kterých je 84 (viz Sekce 2.1.2.2). Počet výsledných příznaků K je tedy roven nejbližšímu nižšímu celočíselnému násobku hodnoty tohoto *hyperparametru*: $K = h - (h \bmod 84)$ [31]. Pro $h = 10000$ je tedy počet výsledných příznaků „pouze“ 9 996.

Shrnutí provedených optimalizací MiniRocket využívá výstup konvoluce z jedné kombinace *kernelu* a dilatace pro výpočet vícero výsledných příznaků. Toho docílí tak, že výstup z konvoluce je použitý pro vícero *biasů* ve funkci *ppv*, respektive pro výpočet exponenciálně více příznaků vycházejících z kernelů s nižší mírou dilatace (viz Sekce 2.1.2.2) [31].

Zároveň se MiniRocket vyhýbá opakovaným násobením, respektive opakovaným výpočtům celé konvoluce pro každý *kernel*. Konvoluci je si možné představit totiž i jako součet sloupců matice $C' \in R^{9 \times \ell_\tau}$, v případě *kernelů* délky 9 a časové řady délky ℓ_τ, kde každá řádka reprezentuje časovou řadu vynásobenou danou váhou *kernelu* a nuly reprezentují zarovnání (*padding*) řádků odpovídající míře dilatace [31]. Pro *kernel* ω délky 9 s mírou dilatace $d = 1$ a časovou řadu τ délky $n = \ell_\tau$ by pak C' vypadala následovně:

$$C' = \begin{bmatrix} 0 & 0 & 0 & 0 & \omega_1\tau_1 & \cdots & \omega_1 2\tau_{n-6} & \omega_1 2\tau_{n-5} & \omega_2\tau_{n-4} \\ 0 & 0 & 0 & \omega_2\tau_1 & \omega_2\tau_2 & \cdots & \omega_1 2\tau_{n-5} & \omega_1 2\tau_{n-4} & \omega_2\tau_{n-3} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \omega_9\tau_5 & \omega_9\tau_6 & \omega_9\tau_7 & \cdots & \omega_9\tau_n & 0 & 0 & 0 & 0 \end{bmatrix}$$

Ve chvíli, kdy jsou váhy *kernelů* omezeny pouze na dvě hodnoty, α a β, je možné nahradit

násobení $\omega_i\tau_j$, které by jinak muselo proběhnout s každým novým *kernelem*, za skládání předem vypočítaných hodnot, pro α tedy $A = \alpha\tau = [a_1, \ldots, a_n]$ a obdobně $B = \beta\tau = [b_1, \ldots, b_n]$ pro β. Protože každý *kernel* je možné zapsat jako kombinaci hodnot α a β, tedy například $\Omega = [\alpha, \beta, \alpha, \ldots, \alpha]$, tak je možné i C' zapsat pomocí hodnot z A a B:

$$C'_\Omega = \begin{bmatrix} 0 & 0 & 0 & 0 & a_1 & \cdots & a_{n-4} \\ 0 & 0 & 0 & b_1 & b_2 & \cdots & b_{n-3} \\ 0 & 0 & a_1 & a_2 & a_3 & \cdots & a_{n-2} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ a_4 & a_5 & a_6 & a_7 & a_8 & \cdots & 0 \\ a_5 & a_6 & a_7 & a_8 & a_9 & \cdots & 0 \end{bmatrix}.$$

Z toho plyne, že pro každou časovou řadu stačí spočítat A a B pouze jednou. Jejich hodnota může být pak použita pro všechny hodnoty dilatace a všechny *kernely*.

Protože lze konvoluci brát jako součet sloupců C' a proto, že *kernely* využívají jen dvou vah $\alpha = -1$ a $\beta = 2$, které se v kernelu nachází v poměru $\frac{3}{9} = \frac{2}{3}$, lze výpočet konvoluce (součet sloupců C') dále optimalizovat. Pro každou dilataci totiž stačí provést $\frac{2}{3}$ výpočtů konvoluce pouze jednou a využít je pro všechny *kernely* s touto dilatací. Nechť C'_α je rovno C' pro *kernel*, jehož váhy jsou pouze α, a obdobně C'_γ, kde $\gamma = 3$. Pro každý *kernel* je pak výsledkem konvoluce součet řádků C'_α sečtená s vhodně vybranými řádky pro daný *kernel* z C'_γ, protože $2 = -1 + 3 \iff \beta = \alpha + \gamma$. Součet řádků C'_α je pak možné používat znovu pro všechny *kernely* sdílející tuto dilataci. V případě *kernelu* $\Omega = [\beta, \alpha, \beta, \beta, \alpha, \alpha, \alpha, \alpha, \alpha]$ s mírou dilatace $d = 1$ je konvoluce ekvivalentní k součtu řádků C'_α sečteným s 1., 3. a 4. řádkem C'_γ [31].

Klasifikátory V případě klasifikátorů se situace MiniRocket proti Rocket nijak výrazně nemění, viz Sekce 2.1.2.1. Pro menší *datasety*, tedy takové kde počet časových řad v trénovacím *datasetu* nepřevyšuje počet příznaků získaných MiniRocket transformací, je nadále doporučována hřebenová regrese. Pro *datasety*, kde počet časových řad naopak tuto hodnotu převyšuje, je doporučena logistická regrese s metodou Adam [57] místo stochastického gradientního sestupu [31].

Časová složitost a nároky na paměť Ačkoliv je MiniRocket v praxi výrazně rychlejší než Rocket, tak je komplexita MiniRocket, stejně jako Rocket, stále $O(k \cdot n \cdot \ell_\tau)$, tedy lineární ve vztahu k počtu výsledných příznaků k, počtu n časových řad τ v *datasetu* a jejich délce ℓ_τ. Ačkoliv proběhla řada optimalizací, od snížení počtu možných kombinací *kernelů* a dilatací, výpočtu konvoluce pouze jednou pro několik příznaků (*biasů*), odstranění nutnosti násobení jednotlivých prvků při konvoluci a vypočítání „v podstatě $\frac{2}{3}$“ *kernelů* pro každou dilataci předem, tak množství sčítání je pořád úměrné vůči počtu *kernelů* a délce časových řad a výpočet zbývající $\frac{1}{3}$ *kernelů* je stále úměrný počtu příznaků [31].

Na rozdíl od Rocket ale MiniRocket již vyžaduje další paměť pro své mezivýpočty. To je konkrétně třináct vektorů stejně dlouhých jako časové řady v používaném *datasetu*, mezi kterými je $A = \alpha\tau$ a $G = \gamma\tau$, devět předem zarovnaných variant G dle dilatace, vektor součtu sloupců $C_\alpha = [1, \ldots, 1] \cdot C'_\alpha$ a vektor s vlastním výsledkem konvoluce C.

2.2 Jazyky

Pro vědecké výpočty, strojové učení a datovou analýzu je implementace efektivních algoritmů a modelů nezbytnou součástí práce. Mezi to patří i výběr konkrétního programovacího jazyku, což může mít na výkonnost těchto algoritmů významný vliv. Výběr totiž určuje faktory jako je rychlost provádění úloh, čitelnost a udržovatelnost kódu a kvalitu ekosystému, tedy například dostupnost různých nástrojů a balíčků, a nebo velikost a aktivitu, či „živost“, vybudované komunity. Může také ale přinášet různá omezení, například v přenositelnosti kódu mezi různými platformami. Záměrem této části je blíže seznámit čtenáře s použitými programovými jazyky Julia [9] a Python [5], se kterými je možné se setkat v této práci.

2.2.1 Python

Python [5, 58] je interpretovaný, dynamicky typovaný, objektově orientovaný programovací jazyk, jehož hlavní předností je jednoduchá, čitelná a snadno naučitelná a pochopitelná syntaxe. Python byl představen již v roce 1991 ve verzi 0.9 [58] a od té doby již prošel mnoha změnami. Mezi nejvýraznější patří verze 2.0 vydaná v roce 2000, která přinesla *garbage collector*, podporu pro znakovou sadu Unicode a *list comprehentions*, a verze 3.0, která měla za cíl opravit množství návrhových chyb svých předchůdců za cenu ztráty zpětné kompatibility s předchozími verzemi [59]. Dnes je Python již druhý nejpoužívanější programovací jazyk dle dotazníku komunity Stack Overflow z roku 2022 s více jak sedmdesáti jedna tisíci odpovědmi z celého světa [60].

Většinu ostatních nejpoužívanějších programovacích jazyků, jako je JavaScript, Java, C#, C++ a C, je možné označit jako *C-like* jazyky, tedy programovací jazyky, které svou syntaxi odvozují od jazyku C. Vyznačují se zpravidla využitím středníku jako symbolu pro oddělení příkazů, složených závorek pro oddělení bloku kódu a kulatých závorek pro parametry funkcí. Python se od těchto jazyků svou syntaxí liší. Bloky kódu jsou v Pythonu oddělovány různou mírou odsazení pomocí „bílých“, někdy také „netisknutelných“, znaků (mezer a znaků tabulátoru). Každý příkaz je psán na vlastní řádek a ačkoliv oddělení středníkem jazyk umožňuje, tak není nutné a je zpravidla považováno za nechtěné. Protože Python dynamicky typovaný jazyk, tak nevyžaduje specifikaci datových typů při deklaraci proměnných. Rozdíly syntaxe jazyků Python a C je možné porovnat mezi Výpisem kódu 1 a Výpisem kódu 2.

```python
# Tato funkce vypíše 5x text "Hello World" a vrátí hodnotu 123.
def speak():
    for i in range(5):
        print('Hello World')
    return 123
```

Výpis kódu 1 Ukázka kódu v jazyku Python

Python je většinou považován za „pomalý jazyk“ [8, 61, 62] a to z několika důvodů. Python, alespoň z pohledu jeho referenční (a nejpoužívanější [63]) implementace zvané CPython, je interpretovaný jazyk. To znamená, že kód se není nijak překládán do strojového kódu před svým spuštěním a místo toho interpret čte a provádí kód „řá-

```c
#include<stdio.h>

// Tato funkce vypíše 5x text "Hello World" a vrátí hodnotu 123.
void speak()
{
    for (int i = 0; i < 5; i++) {
        printf("Hello World\n");
    }

    return 123;
}
```

Výpis kódu 2 Ukázka kódu v jazyku C

dek po řádce“. Python je také dynamicky typovaný jazyk, což znamená, že datové typy proměnných se určují za běhu. Python pak musí, na rozdíl od staticky typovaných jazyků jako je C++ a Java, provádět nákladnou typovou kontrolu a případnou konverzi mezi datovými typy za běhu programu. Python také využívá GIL, *Global Interpreter Lock*, pro synchronizaci přístupu k objektům spravovaných Pythonem (například k proměnným), což znemožňuje provádění několika úkonů najednou pomocí několika vláken [5, 61]. Všechny tyto vlastnosti neumožňují Pythonu provádět velké množství různých optimalizací proti ostatním staticky typovaným kompilovaným jazykům.

Python, konkrétně CPython, je naštěstí snadno rozšiřitelný a to i o balíčky, které spouští nativní kompilovaný strojový kód [64]. Takové balíčky pak dokáží využít výhod staticky typovaného kompilovaného kódu a obejít omezení Pythonu, jako je nemožnost efektivně využívat více vláken. Další možností je využít jinou implementaci jazyku Python, která provádí *Ahead-of-time* (přímou) kompilaci do strojového kódu, jako je například Codon [65], nebo kompilaci *just-in-time* (na vyžádání) jako PyPy [66]. Tyto implementace ale nemusí být plně kompatibilní se všemi balíčky a knihovnami, které Python, respektive CPython, podporuje, či jsou distribuovány s méně permisivní licencí než je distribuován CPython. Mezi další možnosti patří Cython [67], což kompilovaný programovací jazyk rozšiřující syntaxi jazyku Python, která přidává možnost definovat datové typy jednotlivých proměnných a parametrů a návratových hodnot funkcí. Cython svůj kód přímo nekompiluje, ale převede („cythonizuje“) ho do kódu v jazyku C, který se následně zkompiluje do strojového kódu jiným kompilátorem, například GCC či Clang. Cython se zpravidla nepoužívá pro psaní samotných programů, ale pro vytváření samostatných modulů a knihoven pro jazyk Python. Cython takto přinést výrazná zrychlení proti samotnému kódu v jazyce Python, vyžaduje ale hlubší chápání programování v jazyku C (například pro kompilaci generovaných modulů) a jeho syntaxe není zdaleka tak jednoduchá. Toto se snaží řešit balíček Numba [7], který umožňuje JIT kompilovat jednotlivé funkce v jazyce Python do přímo do strojového kódu, kdy v některých případech stačí „dekorovat“ funkce určené ke kompilaci pouhým `@numba.jit()`, bohužel ale s omezenou kompatibilitou s ostatními balíčky. O Numba je možné se dočíst více v Sekci 2.3.1.2.

2.2.2 Julia

V případě vědeckých výpočtů uživatelé programovacího jazyku potřebují skloubit dvě na první pohled protichůdné vlastnosti. Nechtějí být obtěžováni složitostmi pojícími se se staticky typovanými nízkoúrovňovými jazyky a preferují rychlost a jednoduchost vývoje pomalejších dynamicky typovaných jazyků, ale potřebují si zachovat možnost psát v případě potřeby vysoce efektivní a optimalizovaný kód, který mohou používat při každodenní práci.

Programovací jazyk Julia [9] je dynamicky typovaný programovací jazyk (s možností typových anotací) s důrazem na výkonnost, snadnou čitelnost a vysokou úroveň abstrakce a staví se do role ideálního programovacího jazyku pro vědecké výpočty. Julia má za cíl být jazykem, který umožňuje „dosáhnout výkonu stroje, aniž by bylo obětováno pohodlí člověka“ [9]. Snaží se zachovat si uživatelskou příležitost, ale zároveň se i ponaučit z návrhových chyb vysokoúrovňových dynamicky typovaných programovacích jazyků, které omezují jejich rychlost a nutí vývojáře „používat dva jazyky“, kdy prvně vznikne pomalý prototyp a až poté efektivní implementace, či je nutné „slepit“ dohromady části programu napsané v rozdílných programovacích jazycích, kde první je pomalý, ale uživatelsky přívětivý (dynamicky typovaný) jazyk, například Python či Wolfram Mathematica, a druhý je kompilovaný staticky typovaný jazyk, historicky C a FORTRAN, který ale obchází problémy s výkonem jazyku prvního. Julia se tak stává jazykem, který je vhodný pro vědecké výpočty s výkonem srovnatelným [68] s tradičními staticky typovanými jazyky.

Syntaxe jazyku Julia se inspiruje syntaxí jazyku MATLAB, který bloky kódu odděluje klíčovými slovy jako jsou například `begin`, `end`, `for` a další, nevyžaduje odsazení striktním množstvím bílých znaků a ačkoliv je preferován zápis jednoho příkazu na jeden řádek, lze použít i středníky obdobně jako v *C-like* jazycích. Zajímavostí může být, že poslední vykonaný příkaz ve funkci je i návratovou hodnotou funkce i bez použití klíčového slova `x = 123`. Níže je možné najít Výpis z kódu 3 pro porovnání s Výpisem kódu 1 jazyku Python a Výpisem kódu 2 jazyku C.

```
# Tato funkce vypíše 5x text "Hello World" a vrátí hodnotu 123.
function speak()
    for i in 1:5
        println("Hello World")
    end

    123
end
```

Výpis kódu 3 Ukázka kódu v jazyku Julia

Julia kombinuje vlastnosti JIT (*just-in-time*) a AOT (*ahead-of-time*) kompilace a někdy se nazývá jako *AOT-JIT* nebo také *just-barely-ahead-of-time* kompilovaný jazyk, protože v Julii se funkce kompiluje až ve chvíli, kdy se poprvé volá, a následující volání této funkce již využívají její zkompilovanou podobu. Julia tak při kompilaci dokáže odvodit použité datové typy a zároveň využít i typové anotace poskytnuté v kódu, pro příklad `cislo::UInt32 = 1337`, kde se proměnné `cislo` nastaví fixní datový typ `UInt32` (32 bitové celé číslo bez znaménka). Díky těmto vlastnostem se Julia může pyšnit vy-

sokou výkonností, protože kompilátor může provádět velkou škálu optimalizací před spuštěním samotné funkce, respektive jejího strojového kódu. Mezi ně patří i podpora automatické vektorizace kódu. Automatickou vektorizací je myšlen proces, kdy je kompilátor schopný sám nahradit některé instrukce takovými, které dokáží provést tutéž operaci, ale na dvou, čtyřech, osmi a více po sobě jdoucích hodnotách prvcích (například hodnotách v poli). Na Obrázku 4 je možné najít pseudokód znázorňující chování vektorizace při sčítání dvou vektorů.

```
function sum(a::Array{Float, 1024}, b::Array{Float, 1024})
    c::Array{Float, 1024} = [0.0, ..., 0.0]
    for (int i = 0; i < 1024, i += 1)
        # Standardní funkce sčítá postupně každý prvek s každým
        c[i] = a[i] + b[i]
    return c

function sum_vectorized(a::Array{Float, 1024}, b::Array{Float, 1024})
    c::Array{Float, 1024} = [0.0, ..., 0.0]
    for (int i = 0; i < 1024, i += 4)
        # Vektorizovaná funkce sečte 4 prvky v rámci jedné instrukce
        c[i:i+3] = a[i:i+3] + b[i:i+3]
    return c
```

Výpis kódu 4 Pseudokód znázorňující vektorizaci kódu na sčítání vektorů, respektive sčítání prvků dvou polí

Julia se objevila v roce 2012 a od té doby prošla řadou významných aktualizací. Mezi nejvýznamnější verze patří 1.0 vydaná v roce 2018, která znamenala stabilizaci jazyku a jeho API. Kromě vysokého výkonu a (volitelného) statického typování Julia přináší i několik dalších výhod. Může se chlubit *multiple dispatch*, kdy je zavolaná optimální funkce na základě všech datových typů všech jejích parametrů, viz porovnání Julia a C++ na Výpisu z kódu 5 a Výpisu z kódu 6, více-paradigmatičností, kdy kombinuje vlastnosti imperativních, funkcionálních a objektově orientovaných jazyků a rozhraním umožňující snadné, ale efektivní, použití pro vědecké výpočty a numerickou analýzu (podobně jako „matematické“ programovací jazyky MATLAB a R), ale umožňuje psaní obecných programů.

2.3 Nástroje

V této sekci je možné se seznámit s balíčkem sktime [3, 4] pro Python, jehož implementace algoritmů MiniRocket a *k-Nearest Neighbors* a *Dynamic Time Warping* bude v následujících kapitolách porovnávána s jejich implementací v jazyku Julia, s balíčkem MLJ (*Machine Learning in Julia*, také MLJ.jl) [70, 71], který bude použitý jako rozhraní pro implementaci těchto algoritmů. Zároveň zde bude představit archiv pro klasifikaci časových řad od UCR (University of California, Riverside) [52], jehož *datasety* byly použity pro *benchmarkování* algoritmů Rocket a MiniRocket a jehož *datasety* budou použity pro porovnání implementací z balíčku sktime a z této práce.

```julia
abstract type Pet end
struct Dog <: Pet; name::String end;
struct Cat <: Pet; name::String end;

meet(a::Dog, b::Dog) = "sniffs"
meet(a::Dog, b::Cat) = "chases"
meet(a::Cat, b::Dog) = "hisses"
meet(a::Cat, b::Cat) = "slinks"

encounter(a::Pet, b::Pet) = println("$(a.name) meets $(b.name) and $(meet(a,
↪  b))")

rex = Dog("Rex")
jack = Dog("Jack")
simba = Cat("Simba")
leo = Cat("Leo")

# Vypíše: Rex meets Jack and sniffs
encounter(rex, jack)
# Vypíše: Rex meets Simba and chases
encounter(rex, simba)
# Vypíše: Leo meets Jack and hisses
encounter(leo, jack)
# Vypíše: Simba meets Leo and sniffs
encounter(simba, leo)
```

Výpis kódu 5 Ukázka *multiple dispatch* v Julia [69]

2.3.1 sktime

sktime [4] je *open source* knihovna pro Python nabízející *framework* pro strojové učení s časovými řadami a poskytuje nástroje a modely pro analýzu, předpovídání, klasifikaci, regresi a shlukování časových řad a také pro předzpracování, jako je imputace (doplnění chybějících hodnot v datasetu), normalizace (škálování různých číselných hodnot na společný rozsah) a *detrending* (odstranění trendové složky z časový řady). sktime staví na scikit-learn (sklearn) [72],jedné z nejrozšířenějších knihoven pro strojové učení, se kterým nástroje a modely obsažené v sktime udržují kompatibilní rozhraní. To umožňuje využití již existujících nástrojů z scikit-learn, které pomáhají například s optimalizací *hyperparametrů* modelů, výběrem vhodných modelů a vizualizací.

sktime nenabízí žádné grafické rozhraní (GUI) ani rozhraní příkazového řádku (CLI) a je určen pro použití pouze jako knihovna pro jazyk Python. sktime je naprogramován pouze Pythonu za využití NumPy, knihovny pro efektivní práci s (vícerozměrnými) poli dat, a numba, knihovny pro JIT kompilaci výkonově kritických částí kódu napsaném v jazyce Python.

2.3.1.1 numpy

NumPy [6] (celým jménem „Numerical Python“) je *open source* knihovna pro jazyk Python používaná pro numerické výpočty. NumPy poskytuje podporu pro (vícerozměrná) pole, matice a širokou škálu matematických funkcí pro efektivní manipulaci s takovými poli.

NumPy se často používá pro vědecké výpočty, analýzu dat a strojové učení a další oblasti, které vyžadují efektivní zpracování velkého množství numerických dat. Oproti

```cpp
#include <iostream>
#include <string>

class Pet {
    public: std::string name;
    Pet(std::string name) {
        this->name = name;
    }
};
class Dog : public Pet { using Pet::Pet; };
class Cat : public Pet { using Pet::Pet; };

std::string meets(Pet a, Pet b) { return "FALLBACK"; }
std::string meets(Dog a, Dog b) { return "sniffs"; }
std::string meets(Dog a, Cat b) { return "chases"; }
std::string meets(Cat a, Dog b) { return "hisses"; }
std::string meets(Cat a, Cat b) { return "slinks"; }

void encounter(Pet a, Pet b) {
    std::cout << a.name << " meets " << b.name
              << " and " << meets(a, b) << std::endl;
}

int main() {
    Dog rex = Dog("Rex");
    Dog jack = Dog("Jack");
    Cat simba = Cat("Simba");
    Cat leo = Cat("Leo");

    // Vypíše: Rex meets Jack and FALLBACK
    encounter(rex, jack);
    // Vypíše: Rex meets Simba and FALLBACK
    encounter(rex, simba);
    // Vypíše: Leo meets Jack and FALLBACK
    encounter(leo, jack);
    // Vypíše: Simba meets Leo and FALLBACK
    encounter(simba, leo);

    return 0;
}
```

Výpis kódu 6 Ukázka chybějícího *multiple dispatch* v C++ [69]

Pythonu, kde jsou pole čísel reprezentovány jako vektor obecných objektů, kde až daný objekt obsahuje samotnou číselnou hodnotu, jsou v NumPy pole čísel uloženy přímo jako souvislé bloky paměti, jejichž prvky jsou předem daného datového typu (jsou tedy homogenní), což výrazně urychluje přístup k datům a operace s nimi.

2.3.1.2 Numba

Numba [7] je *open-source* knihovna a *just-in-time* (JIT) kompilátor pro jazyk Python určený k optimalizaci výkonu numerických algoritmů, vědeckých výpočtů a dalších aplikací využívajících numerické výpočty.

Numba funguje tak, že překládá kód z jazyku Python do optimalizovaného strojového kódu pomocí LLVM (*Low-Level Virtual Machine*), univerzálního nástroje (nejen) pro optimalizaci a kompilaci kódu, jako *backendu*. Numba podporuje NumPy pole a nabízí funkce, jako je automatická paralelizace a vektorizace kódu bez omezení, které Python má, a podporuje akceleraci některých výpočtů pomocí grafických karet s technologií CUDA. Díky tomu lze dosáhnout výkonu srovnatelného se staticky typovanými kompilovanými jazyky bez ztráty snadného použití a flexibility jazyku Python.

2.3.2 MLJ

MLJ (*Machine Learning in Julia*) [70, 71], nebo také MLJ.jl, je *open source framework* pro strojové učení v programovacím jazyce Julia. Poskytuje jednotné rozhraní pro práci s širokou škálou modelů strojového učení, podobně jako scikit-learn v případě jazyku Python, a nabízí také funkce pro předzpracování dat, *feature engineering* a výběr vhodných modelů, jejich řetězení a skládání a optimalizaci *hyperparametrů* a je navržen tak, aby byl flexibilní a snadno rozšiřitelný. Tato rozšiřitelnost a flexibilita plyne z vlastnosti jazyku Julia, kterou je *multiple dispatch*, viz Sekce 2.2.2.

MLJ sám o sobě neobsahuje v podstatě žádné modely strojového učení mimo několika základních. Mezi ně patří například *One Hot Encoder* a *Constant Classifier* [73]. *One Hot Encoder* transformuje kategorický příznak na několik binárních příznaků pro každou kategorii pro modely, které neumí s kategorickými příznaky pracovat. *Constant Classifier* naopak slouží spíše jako ukázka implementace rozhraní MLJ pro vývojáře externích modelů, který všechny vzorky klasifikuje stejným pravděpodobnostním rozložením všech tříd v trénovacím *datasetu* a ignoruje tedy všechny příznaky trénovacího *datasetu*. MLJ nemá za cíl implementovat samotné modely a slouží hlavně jako sada nástrojů pro usnadnění práce v oblasti strojového učení a jako univerzální rozhraní pro externí modely ostatních vývojářů.

2.3.3 UCR Archive

URC Archive [52], plným názvem *UCR (University of California, Riverside) Time Series Classification Archive*, je sbírka 128 datových souborů časových řad spravovaná Kalifornskou univerzitou v Riverside. Byl vytvořen s cílem usnadnit výzkum v oblasti analýzy a klasifikace časových řad a pro použití při vývoji algoritmů a srovnávání jejich přesnosti a výkonu. Archiv obsahuje širokou škálu datových souborů z různých oblastí,

včetně ekonomie, biologie, inženýrství a fyziky. Všechny datové sady jsou jednorozměrné, ale liší se velikostí (počtem časových řad) a délkou časových řad. Soubor datasetů s jednorozměrnými a i vícerozměrnými časovými řadami je možné najít na timeseriesclassification.com[1], archivu kombinujícím *URC Archive* jednorozměrných časových řad a *UEA Archive* vícerozměrných časových řad [52].

[1] http://www.timeseriesclassification.com/index.php

Kapitola 3

Implementace

V této kapitole se nachází popis implementace algoritmů MiniRocket a *k-Nearest Neighbors* s *Dynamic Time Warping* s využitím rozhraní balíčku MLJ (*Machine Learning in Julia* nebo také MLJ.jl). Dále je zde možné najít rozdíly mezi implementacemi těchto algoritmů v balíčku sktime, respektive scikit-learn v případě KNN, s implementací v rámci této práce v jazyku Julia. Cílem je také popsat různé optimalizace, které implementace v jazyku Julia přináší.

3.1 Rozšiřování MLJ o další modely

MLJ poskytuje univerzální rozhraní pro implementaci modelů strojového učení, které je možné najít v balíčku MLJInterface.jl[1]. Samotné rozhraní je velmi „lehké“ a neobsahuje žádné funkce, respektive jejich implementaci, pro samotné použití a testování implementovaných modelů [74].

Implementaci rozhraní MLJ pro již implementovaný model je možné shrnout do několika částí, které budou znázorněny na implementaci jednoduchého klasifikačního modelu *k-Nearest Neighbors* velmi podobného tomu, který implementuje tato práce.

První z nich je definice samotného modelu, respektive struktury obsahující *hyperparametry* a jiná nastavení modelu. Toho je možné docílit použitím makra `@mlj_model` před její definicí. Toto makro umožňuje nastavit výchozí hodnoty parametrů a samo implementuje i některé funkce MLJ rozhraní pro kontrolu jejich správnosti. Zároveň je nutné určit i typ modelu, tedy zda se jedná o pravděpodobnostní model , deterministický model nebo o transformátor. To určuje, které funkce rozhraní bude potřeba implementovat a jaké parametry by měly mít [74]. Ukázku jak implementovat strukturu definující model v MLJ je možné najít ve Výpisu z kódu 7. Pro `MLJModelInterface` bude nadále používána pouze zkratka `MMI`.

[1] https://github.com/JuliaAI/MLJModelInterface.jl

```
import MLJModelInterface

MMI = MLJModelInterface

euclidean_distance = (a, b) -> sum(sqrt.((a-b).^2))

MMI.@mlj_model struct SimpleKNNClassifier <: MMI.Deterministic
    k::Integer = 1::(_ ⩾ 1)
    metric::Function = euclidean_distance
end
```

Výpis kódu 7 Implementace struktury modelu KNN v MLJ

Následně je potřeba implementovat funkci `MMI.train`[2], která slouží pro trénování modelu. Protože KNN je *supervizovaný* model, tak bude tato funkce vyžadovat jak příznaky trénovacího *datasetu* `X`, tak i k tomu příslušné třídy `y`. *Nesupervizovaný* model by si vystačil pouze s příznaky trénovacího *datasetu*. Implementaci je možné najít ve Výpisu z kódu 8. MLJ rozhraní vyžaduje, aby funkce vracela natrénované parametry modelu `fitresult`, což je v případě KNN kompletní *dataset*, `cache`, pokud model při trénování například předzpracuje příznaky, které by se mohly hodit pro případ opětovného trénování, a `report`, který může obsahovat informace o průběhu trénování, důležitosti některých příznaků a jiné informace, na kterých nezávisí následovná predikce tříd, ale mohou mít pro uživatele modelu nějaký význam. Jak `cache` tak i `report` mohou být pouze `nothing`, pokud je model neposkytuje [74].

```
function MMI.fit(model::SimpleKNNClassifier,
                 X::AbstractVector{AbstractVector{TX}},
                 y::AbstractVector{Ty}
)::Tuple{typeof(X), typeof(y)} where {TX <: Number, Ty <: Any}

    fitresult = (X, y)
    cache = nothing
    report = nothing
    return fitresult, cache, report

end
```

Výpis kódu 8 Implementace funkce `MLJModelInterface.fit(...)` pro KNN v MLJ

Dalším krokem je implementace funkce `MMI.predict`, ve které probíhá samotná predikce tříd pro zatím neviděná data, například z testovacího či validačního *datasetu*. Tato funkce vyžaduje jako parametry, kromě nových dat, také výsledek trénování `fitresult` z funkce `MMI.predict`. Implementaci je možné najít ve Výpisu z kódu 9.

[2] `MMI = MLJModelInterface`

```
function MMI.predict(model::SimpleKNNClassifier,
                     (X, y)::Tuple{
                         AbstractVector{AbstractVector{TX}},
                         y::AbstractVector{Ty}
                     },
                     newX::AbstractVector{AbstractVector{TX}}
)::Vector{Ty} where {TX <: Number, Ty <: Any}

    results::Vector{Ty} = []
    for el in newX
        distances = map(x -> model.metric(x, el), X)
        indices = sortperm(distances)
        push!(results, argmax(
            x-> count(i -> i == x, a),
            y[indices][begin:begin+model.k]
        ))
    end
    return results

    end
```

Výpis kódu 9 Implementace funkce `MLJModelInterface.predict(...)` pro KNN v MLJ

V situaci, kdy by uživatel chtěl dodávat data pro trénování či predikci v jiném formátu, například jako `Matrix{Number}` místo `AbstractVector{AbstractVector{Number}}` pro soubor příznaků, je nutné implementovat funkci `MMI.reformat`, která provede konverzi na datový typ vyžadovaný funkcemi `MMI.fit` a `MMI.predict` a funkci `MMI.selectrows`, která dokáže efektivně vybrat podmnožinu dat. To je podmíněno použitím rozhraní `MMI.Machine` a ne přímo funkce `predict` a `fit` daného modelu. Implementaci je možné najít ve Výpisu z kódu 10 a ve Výpisu z kódu 11.

```
function MMI.reformat(m::MiniRocketModel, X, y) = (
    MMI.reformat(m, X)[1], y
)

function MMI.reformat(::MiniRocketModel, X)
    return if typeof(X) <: AbstractVector && eltype(X) <:
    ↪  AbstractVector{<:Number}
        (X,)
    elseif typeof(X) <: AbstractMatrix{<:Number}
        ([view(X, :, i) for i in axes(X, 2)],)
    else
        @assert false "Unknown data format"
    end
end
```

Výpis kódu 10 Implementace funkce `reformat(...)` pro KNN pomocí rozhraní MLJ

```
function MMI.selectrows(::MiniRocketModel, I, X, y) =
    (view(X, I), view(y, I))
function MMI.selectrows(::MiniRocketModel, I, X) =
    (view(X, I),)
```

Výpis kódu 11 Implementace funkce `selectrows(...)` pro KNN pomocí rozhraní MLJ

V tento moment je implementace modelu pro MLJ hotová a stačí ho jen „inzerovat“ samotnému balíčku MLJ, viz Výpis z kódu 12. Takto pak MLJ dokáže správně balíček integrovat do svého rozhraní a najít ho v případě, že se ho uživatel zkusí načíst pomocí `SimpleKNNClassifier = MLJ.@load SimpleKNNClassifier pkg=MyKNNPackage`. Je vhodné také doplnit „docstring“ (dokumentační řetězec) jako uživatelskou dokumentaci, která popisuje daný model, respektive samotný algoritmus, jak načíst a vytvořit instance modelu, jaké jsou jeho *hyperparametry* a další informace. Kompletní výčet všech parametrů pro inzerci modelů pro MLJ je možné najít v dokumentaci[3] [74].

```
MLJModelInterface.metadata_pkg(
    (SimpleKNNClassifier,),
    name = "MyKNNPackage",
    uuid = "beef1337-6969-48eb-9615-decaf420cafe",
    is_pure_julia = true,
    is_wrapper = false,
)
MLJModelInterface.metadata_model(
    SimpleKNNClassifier,
    input_scitype = AbstractVector{AbstractVector{MMI.Continuous}},
    output_scitype = AbstractVector{<:Any},
    load_path = "MyKNNPackage.SimpleKNNClassifier"
)

"""
$(MLJModelInterface.doc_header(SimpleKNNClassifier))

This model is a simple implementation of
the K-Nearest Neighbors model. ...

### Hyperparameters

SimpleKNNClassifier has two hyperparameters, `K` and `metric`,
where K sets the number of ...
"""
SimpleKNNClassifier
```

Výpis kódu 12 Inzerce modelu KNN pro balíček MLJ

[3] https://alan-turing-institute.github.io/MLJ.jl/dev/adding_models_for_general_use/#MLJModelInterface.metadata_pkg

3.2 Porovnání implementace

V této práci byly představeny algoritmy Rocket (viz Sekce 2.1.2.1), MiniRocket (viz Sekce 2.1.2.2) a *k-Nearest Neighbors* s *Dynamic Time Warping* (viz Sekce 2.1.1.1 a Sekce 2.1.1.2) spolu s modifikacemi omezující prohledávaný prostor skrz DTW. V následujících odstavcích, sekcích a kapitolách bude pracováno s pojmem „alokace paměti“, kterým je dynamická alokace paměti na haldě (*heap*), ne zásobníku (*stack*), kterou programu přiděluje operačního systém skrz svého správce paměti (*memory manager*).

Implementované algoritmy přináší řadu optimalizací a změn zaměřujících se primárně na zlepšení výkonu, ačkoliv jádro každého algoritmu zůstává stejné a implementované algoritmy jsou si tedy stále podobné s implementací z balíčku sktime. Změny, optimalizace a další rozdíly mezi implementací algoritmů v této práci a jejich implementací v sktime budou představeny v následujících sekcích.

Většina optimalizací provedených v této práci staví na redukci celkového počtu alokací paměti, jejich realokací a cyklu alokace-uvolnění-alokace a následné přepsání (například vynulování) podobných bloků paměti a na optimalizaci přístupu k ní, kdy je preferováno čtení po sobě jdoucích bloků paměti. Přístup omezení počtu alokací staví na principu, že je často rychlejší již programu přiřazenou paměti přepsat novými daty, než žádat systém o novou paměť (která bude v některých případech poté rovnou přepsána jinými daty) [75]. Tento přístup, jednak šetří čas, který trvá samotná alokace paměti systémem, dále také benefituje z minimalizace fragmentace dat, tedy stavu, kdy jsou jednotlivé části v paměti na různých, ne po sobě jdoucích místech, či dokoce v jiných pamětech počítače (například v RAM a ne v cache přímo v procesoru) a z dodržení „principu lokality dat“. Princip lokality dat odkazuje na pozorování, že programy zpravidla přistupují k datům v paměti počítače, které jsou si blízké jak v časové, tak i prostorové rovině. Tedy že pokud program přistoupil k jednomu bloku dat, nejspíše v blízké budoucnosti přistoupí k datům v jejich okolí, čehož moderní procesory využívají pro urychlení běhu programů, například pomocí technik jako je *speculative execution*, kdy dochází k provedení instrukcí, u kterých je odhadováno, že by mohly nastat, ještě dříve, než by se měly správně objevit v programu. Na základě reálného běhu programu se pak vyberou výsledky těch instrukcí, které program doopravdy využije [75].

Tím, že budou data držena v konsekutivních blocích v paměti, které se nebudou často měnit, například dealokací starého a alokací nového bloku paměti, který navíc musí přidělit ještě správce paměti operačního systému, se je možné vyhnout momentům, kdy se požadovaná data vůbec, či ještě, nenachází v rychlejší paměti blíže k samotným jádrům procesoru a zrychlit tedy běh programu.

3.2.1 KNN s DTW

Na rozdíl od této práce, která se vydává cestou vlastní implementace, balíček sktime sám *k-Nearest Neighbors* neimplementuje, ale využívá model, který poskytuje balíček scikit-learn. Tento model implementuje několik různých technik, mimo *brute-force* přístupu, pro efektivnější vyhledávání sousedů, jako je například *Ball tree*, který urychluje hledání nejbližších sousedů dělením vysokodimenzionálních trénovacích dat (tedy trénovacích dat s mnoha příznaky) do sebe vnořených koulí, a *k-d tree*, který trénovací data

indexuje jejich postupným dělením rovinami v prostoru. Ani jedna technika není ale pro *Dynamic Time Warping* aplikovatelná, protože *Ball tree* indexuje data dělením na soubory příznaků, zatím co DTW vyžaduje pro svůj výpočet všechny příznaky (pozorování) v časové řadě, a *k-d tree* vyžaduje, aby výpočet vzdálenosti splnil všechny podmínky metriky, viz Sekce 2.1.1.1, které ale DTW nesplňuje. V této práci je tedy implementován pouze *brute-force* přístup vyžadující výpočet vzdálenosti mezi vyhledávaným vzorkem a všemi vzorky v trénovacím *datasetu*. Pro zefektivnění výpočtů není *brute-force* KNN v sktime balíčku napsaný pouze v jazyku Python, ale jako Cython (viz Sekce 2.2.1) modul[4], který pak využívá Python rozhraní[5] pro *k-Nearest Neighbors*. Díky tomu, stejně jako implementace v této práci, dokáže efektivně využít více vláken pro paralelizaci výpočtů vzdáleností mezi jednotlivými časovými řadami. Narozdíl od sktime je díky vlastní implementaci KNN možné efektivně implementovat spodní meze výpočtů DTW vzdálenosti jako je LB_Keogh (viz Obrázek 2.4 a Sekce 2.1.1.2), jenž sktime, respektive scikit-learn, nepodporuje stejně jako výpočet DTW pro datasety s různě dlouhými časovými řadami.

Balíček sktime implementuje svou DTW funkci[6] skrz *„factory“*[7] (tovární) metodu, která bere jako parametry informaci o různých omezeních prohledávaného prostoru, tedy například sklon Itakurova parallelogramu nebo šířku okna v případě Sakoe–Chiba pásma, a samozřejmě obě časové řady. Na základě těchto parametrů vytvoří matici[8], respektive masku, o stejné velikosti jako matice, kterou využívá samotné DTW při výpočtu. Tato maska se skládá z hodnot datového typu `numpy.float64`, tedy čísel s plovoucí desetinnou čárkou s dvojnásobnou přesností o celkové velikosti 64 bitů. Jednotlivé hodnoty v této masce, které jsou dle daných omezení prohledávaného prostoru nedostupné, jsou nastaveny na hodnotu nekonečna, respektive `numpy.inf`, a pozice, které jsou dostupné, jsou nastaveny na hodnotu 0. Při samotném výpočtu DTW algoritmus pak navštíví všechny indexy[9] a výpočet samotné vzdálenosti proběhne jen tehdy, pokud daný index obsahuje konečnou hodnotu (v tom smyslu, že není rovna nekonečnu). Implementace DTW v této práci jde naopak cestou, kdy není takovou masku potřeba vytvářet, ale jednotlivé instance DTW ve své matici, která se v počátku skládá pouze z hodnot `prevfloat(typemax(T))`, tedy nejvyššího konečného čísla reprezentovaného datovým typem `T <: AbstractFloat` (v případě `Float64` tedy $\sim 1.79 \cdot 10^{308}$). Zároveň je pro všechny běhy nad jednou instancí DTW struktury společná matice, která je realokována na větší pouze v případě potřeby dle právě počítaných časových řad. Tato implementace tedy šetří čas a paměť tím, že prochází pouze indexy, ve kterých je opravdu potřeba vypočítat *warping*, omezuje počet alokací v případě více výpočtů (v případě *datasetu* s N časovými řadami, které jsou všechny stejně dlouhé, pouze na 1 místo N) a umožňuje použití makra `@fastmath`, které pro rychlejší výpočty s čísly s plovoucí desetinnou čárkou povoluje porušení IEEE 754 standardu [76] a povoluje změnu pořadí instrukcí, například v případech, kdy jsou matematické operace asociativní, ale ne pro čísla dle IEEE 754, dále zavádí předpoklad, že hodnoty nekonečna a NaN (*not a number*) se

[4] https://github.com/scikit-learn/scikit-learn/blob/1.2.2/sklearn/metrics/_pairwise_distances_reduction/_argkmin.pyx.tp
[5] https://github.com/scikit-learn/scikit-learn/blob/1.2.2/sklearn/neighbors/_base.py#L824
[6] https://github.com/sktime/sktime/blob/v0.18.0/sktime/distances/_distance.py#L759
[7] https://github.com/sktime/sktime/blob/v0.18.0/sktime/distances/_dtw.py#L138
[8] https://github.com/sktime/sktime/blob/v0.18.0/sktime/distances/_lower_bounding_numba.py#L129
[9] https://github.com/sktime/sktime/blob/v0.18.0/sktime/distances/_dtw_numba.py#L21

v tímto makrem označeném bloku kódu nevyskytují a další optimalizace pro urychlení výpočtů.

3.2.2 MINIROCKET

Implementace algoritmu MiniRocket v Python balíčku sktime se v několika ohledech liší s implementací v této práci. V následujících dostavcích budou nejdůležitější změny probrány.

Jednou z úprav je „kvazi-determinizace“ MiniRocket algoritmu. Jak bylo již zmíněno v Sekci 2.1.2.2, jediná náhodná část tohoto algoritmu je náhodný výběr vzorků z trénovacího *datasetu* pro výběr *biasů*. Jednou z úprav MiniRocket je přidání hyperparametru `shuffled`. Tento hyperparametr nabývající hodnot `true` nebo `false` značí, že trénovací *dataset*, který je předán do funkce `MMI.fit(::MiniRocketModel, ...)`, je již náhodně seřazen, jak někdy bývá zvykem, a algoritmus tedy nemusí vybírat vzorky náhodně, ale může vybírat z po sobě jdoucích prvků. Cílem je redukovat nutnost načítat data z náhodných bloků paměti, které se z principu budou v paměti nacházet na náhodných místech, ale z bloků po sobě jdoucích a tím dodržovat princip lokality dat.

Další a významější úpravou je, že veřejné rozhraní MiniRocket nevyžaduje, na rozdíl od implementace v balíčku sklearn, jako datový formát trénovacího i transformovaného *datasetu* trojrozměrné pole[10], kde $\tau_{\{X,Y\}}$ značí Y-tý prvek X-té časové řady, kde $X \in \widehat{m}$, o délce n, ve formátu $[[[\tau_{\{1,1\}}, \ldots, \tau_{\{1,n\}}]], \ldots, [[\tau_{\{m,1\}}, \ldots, \tau_{\{m,n\}}]]]$, které se ale pro potřeby samotné implementace algoritmu stejně převádí[11] do „zploštěného“ formátu dvojrozměrného pole $[[\tau 1_1, \ldots, \tau 1_n], \ldots, [\tau m_1, \ldots, \tau m_n]]$, navíc s vynuceným převodem datového typu `numpy.float32`, který nemusí být vždy chtěný. Implementace rozhraní v této práci nechává výběr datového typu na uživateli, tedy s omezením na podtypy abstraktího datového typu `AbstractFloat`. Sttejně tak, pokud uživatel dodá *dataset* v preferovaném formátu, tedy `Matrix{<:AbstractFloat}` s jednotlivými časovými řadami ve sloupcích, nevznikají žádné další kopie a neprobíhá žádná konverze dat.

Nejdůležitější změnou je redukce alokací a dealokací paměti. Na Výpisu z kódu 13 a 14 pro implementaci z balíčku sktime, respektive implementaci v rámci této práce, je možné vidět korpus funkce `transform` vyznačující místa, kde probíhá alokace paměti. Z ukázky je patrné, že za dobu běhu transforace proběhne až $O(2 + N \cdot 5 + N \cdot D \cdot 10 + N \cdot D \cdot K \cdot 2)$ aktů alokace a dealokace paměti, který roste na celkový počet $O(2+N \cdot R \cdot 5+N \cdot D \cdot R \cdot 10+N \cdot D \cdot K \cdot R \cdot 2)$ v případě využití R vláken procesoru. Naštěstí jsou funkce implementující MiniRocket kompilovány pomocí Numba (viz Sekce 2.3.1.2) a JIT kompilace tedy dokáže některé z alokací omezit, bohužel ale ne všechny. Oproti tomu v implementaci, která se nachází v této práci, probíhá pouze pět alokací. Obdobné změny byly provedeny i v rámci funkce `fit`, konkrétně funkce `fit_biases`, kterou funkce `fit` využívá pro nalezení *biasů* v rámci trénovacího datasetu.

Poslední výraznou optimalizací je kompletní zbavení se pole C_γ, jak je možné vidět ve Výpisu z kódu 13 a 14, které bylo v původní implementaci v balíčku sktime pouze kopírováno z místa na místo. Z toho plyne nejen ušetření množství alokované paměti,

[10] https://github.com/sktime/sktime/blob/v0.18.0/sktime/transformations/panel/rocket/_minirocket.py#L92

[11] https://github.com/sktime/sktime/blob/v0.18.0/sktime/transformations/panel/rocket/_minirocket.py#L106

ale i času, který jinak trvá samotné kopírování.

Původní implementace v balíčku sktime nízký výkon Pythonu řeší pomocí JIT kompilace funkcí `fit_biases` a `transform` za pomoci balíčku Numba. Balíček Numba dokáže, podobně jako jazyk Julia (viz Sekce 2.2.2), provádět automatickou vektorizaci kódu a dosahovat tak ještě lepšího výkonu. V jazyku Julia není vždy všechen kód automaticky vektorizován a ne všechny funkce, které Julia nabízí, vektorizaci podporují, ať už kvůli svému návrhu nebo nedokonalé implementaci ve standardní knihovně. Tento problém řeší obzvlášť dva balíčky, které jsou v této práci využity. Konkrétně se jedná o balíček LoopVectorization.jl [77], který nabízí například makro `@turbo` pro lepší vektorizaci některých for-cyklů, a balíček VectorizedStatistics.jl [78], poskytující vektorizované funkce `vsum`, `vmaximum` a další vektorizované a jinak optimalizované statistické funkce.

```
def transform(X, parameters):
    N, L = X.shape
    K = number_of_features
    D = len(dilations)
    # ...
    # Mezi běhy kostantní indexy, které vybírají umístění β hodnot v kernelu
    indices = np.array(...).reshape(84,3) # ALOKACE - pole 84·3 prvků
    # ...
    features = np.zeros((N, K))   # ALOKACE - pole N·K prvků
    # ...
    # prange je paralelní implementace range() z balíčku Numba
    for i in prange(N):
        _X = X[i]   # ALOKACE - kopie vzroku délky L
        A = _X * -1 # ALOKACE 2× - kopie + úprava vzroku délky L
        G = _X * 3  # ALOKACE 2× - kopie + úprava vzroku délky L
        # ...
        for j in range(D):
            # ...
            C_α = np.zeros(L)       # ALOKACE - pole L prvků
            # ...
            C_γ = np.zeros((9, L)) # ALOKACE - pole 9·L prvků
            # ...
            for k in range(84):
                idx1, idx2, idx3 = indices[k]
                # ...
                C =\ # ALOKACE 2× - upravovaná kopie + nová proměnná délky L
                    C_α +\
                    C_γ[idx1] +\
                    C_γ[idx2] +\
                    C_γ[idx3]
                # ...
```

Výpis kódu 13 Znázornění míst alokujících paměť ve funkci `transform` v sktime implementaci algoritmu MiniRocket

```
# Pole tripletů indexů β hodnot je konstantní a statické
const INDICES::StaticArray{...} = [...]

function transform(X)
    # Počet dostupných vláken procesoru
    R = Threads.nthreads()
    N, L = X.shape
    K = number_of_features
    D = len(dilations)
    # ...
    features = zeros(N, K)
    C_{α·R} = zeros(N, R)
    C_R = zeros(N, R)
    A_R = zeros(N, R)
    G_R = zeros(N, R)
    # ...
    Threads.@threads for i in range(N):
        r = Threads.threadid()
        # Bez alokace - vytvoření pohledů na proměnné, nejedná se o jejich
        ↪ kopie
        _X = @views X[i]
        C_α = @views C_{α·R}[:, r]
        C = @views C_R[:, r]
        A = @views A_R[:, r]
        G = @views G_R[:, r]
        # Bez alokace, výpočty probíhají in-place
        A .*= _X .* -1
        G .*= _X .* 3
        # ...
        for j in range(D):
            # ...
            copyto!(C_α, A)
            # ...
            for k in range(84):
                # ...
                copyto!(C, C_α)
                # Následne přičtení hodnot z C_γ probíhá, včetně jejich
                ↪ vlastního výpočtu, in-place
                # ...
```

Výpis kódu 14 Znázornění míst alokujících paměť ve funkci `transform` v algoritmu MiniRocket implementovaném v této práci

3.3 Struktura balíčku

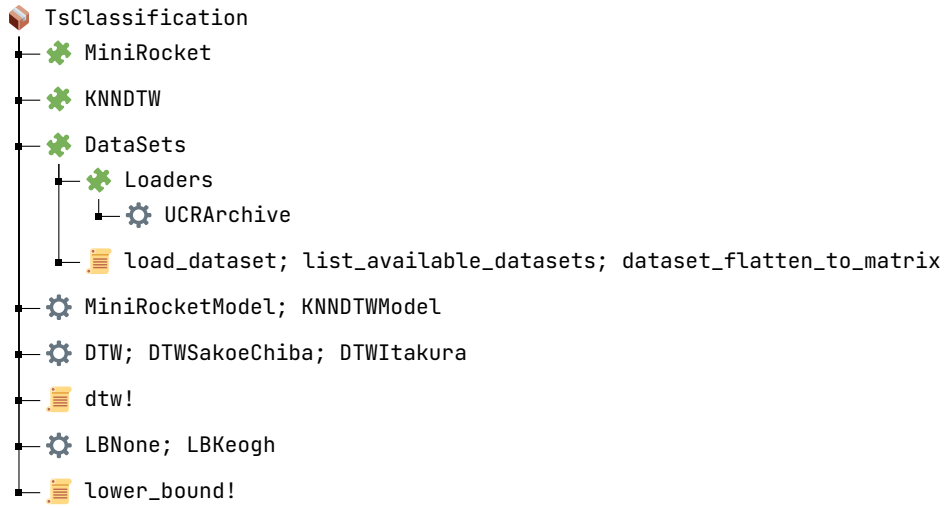
V této sekci bude rozebrána struktura a uspořádání balíčku, respektive umístění důležitých modulů, funkcí, struktur a dalších komponent. Zjednodušenou strukturu balíčku je možné vidět na Obrázku 3.1. Prvky, které jsou pro obecný přehled méně významné, jsou pro jednoduchost vynechány a bude jim věnována pozornost hlouběji v následujících odstavcích.

Na schématech, se kterými je možné se v této sekci setkat, je využité následující značení pro odlišení jednotlivých komponent balíčku této práce. Prvky, jejichž název začíná

podtržítkem, tedy například `_Utils`, nejsou určené pro použití koncovým uživatelem.

- Balíček, respektive jeho kořenový modul
- Modul uvnitř balíčku
- Vlastní abstraktní typ
- Definice struktury
- Funkce
- Makro
- Konstanta

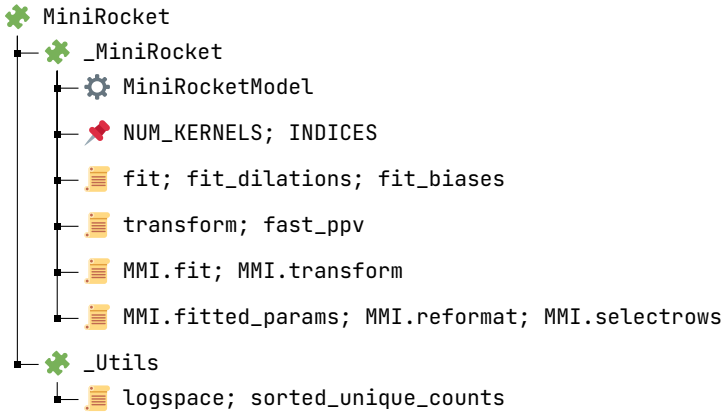
`TsClassification` Název balíčku a tedy i hlavního modulu je `TsClassification`. Je v něm možné najít exportované všechny důležité struktury, funkce a některé moduly. Ty jsou rovnou k dispozici po vykonání příkazu `using TsClassification`. Jejich seznam je možné najít na Obrázku 3.1.



Obrázek 3.1 Zjednodušené uspořádání balíčku s exportovanými moduly, strukturami a funkcemi

`MiniRocket` Prvním modulem je modul `MiniRocket`. Skládá se z dvou dalších modulů, konkrétně `_MiniRocket` a `_Utils`. První zmíněný obsahuje samotnou implementaci algoritmu MiniRocket ve formě funkcí `fit`, `fit_dilations`, `fit_biases` a `trasnform` s `fast_ppv` spolu s přetížením funkcí z `MLJModelInterface`, tedy `MMI.fit` pro trénování modelu, `MMI.transform` pro samotnou MiniRocket transformaci a `MMI.reformat` a `MMI.selectrows` pro převod dat na datový typu podporovaný algoritmem MiniRocket. Druhý zmíněný implementuje funkce, které standardní knihovna jazyka Julia neposkytuje, ale jsou nutné pro implementaci algoritmu MiniRocket. Mezi ně patří funkce `logspace` podobná funkci `numpy.core.logspace`[12] z balíčku NumPy pro Python a funkci `sorted_unique_counts` doplňující funkci `count` ze standardní knihovny Julia o možnost vracet i počet výskytu unikátních prvků, obdobně jako `numpy.lib.unique`[13] z balíčku NumPy. Strukturu celého modulu je možné najít na Obrázku 3.2

[12] https://numpy.org/doc/stable/reference/generated/numpy.logspace.html
[13] https://numpy.org/doc/stable/reference/generated/numpy.unique.html



Obrázek 3.2 Struktura modulu `MiniRocket`

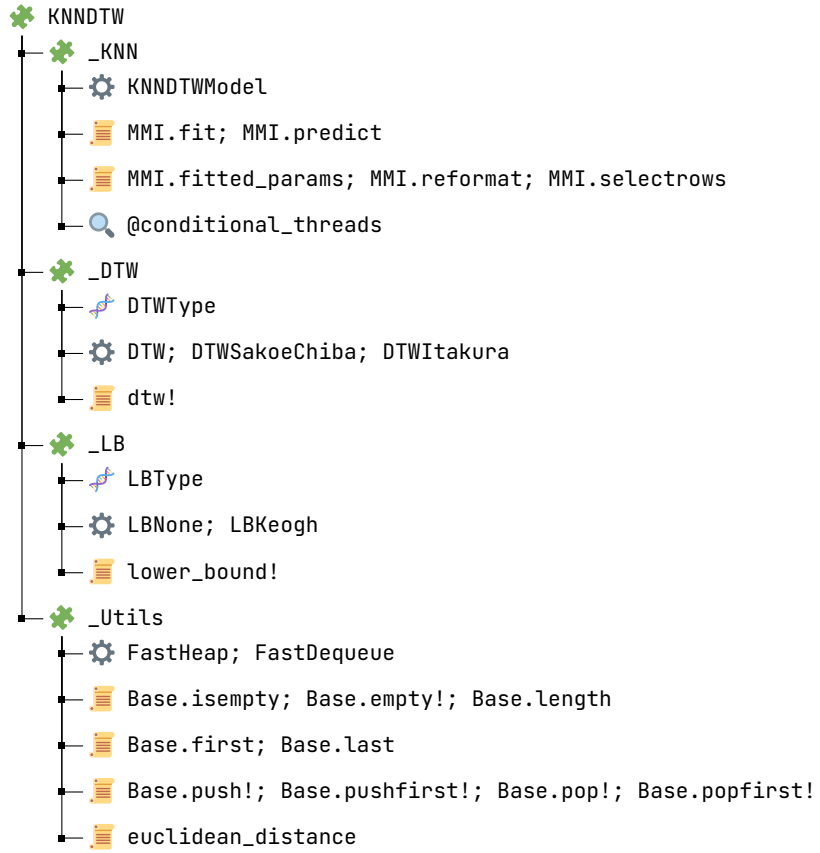
KNNDTW Dalším modulem je modul `KNNDTW`. Skládá se ze čtyř dalších modulů, z toho tři hlavní: `_KNN` obsahující algoritmus KNN, `_DTW` obsahující různé varianty *Dynamic Time Warping* algoritmů, `_LB` obsahující funkce pro spodní meze vzdálálenosti.

Modul `_KNN` obsahuje samotnou implementaci algoritmu *k-Nearest Neighbors* obdobnou formou jako MiniRocket, protože se ale nejedná o transformátor ale klasifikátor, implementuje funkci `MMI.predict` a `MMI.predict_mode` místo `MMI.transform`. Tento modul také obsahuje makro `@conditional_threads`, které dle podmínky před takto označený for cyklus zařadí makro `Threading.@threads` a dle toho ve výsledku povolí nebo zakáže paralelizaci pro daný cyklus. Makro má následující použití `@codnitional_threads <podmínka> <for ... end>`. Díky tomu je možné efektivně implementovat paralelizaci v KNN dle velikosti trénovacího či nového *datasetu* a určit tak který z nich bude zpracováván paralelně.

V modulu `_DTW` se nachází datový typ `DTWType` ze kterého vycházejí struktury `DTW`, `DTWSakoeChiba` a `DTWItakura` ukládající parametry pro stejnojmenné varianty *Dynamic Time Warping*, které se pak počítají pomocí funkce `dtw!(…)`.

Další modul `_LB` obsahuje obdobné struktury a funkce pro výpočet spodních mezí vzdáleností, jako je například LB_Keogh, ale obsahuje i LB_None, což je funkce, která vždy vrací nulu a je možné ji použít jako zástupce ve chvíli, kdy uživatel žádnou jinou funkci pro výpočet spodní meze využít nechce.

Posledním modulem je `_Utils`, kde je možné najít implementaci binární minmax-haldy `FastHeap`, kterou využívá algoritmus KNN pro efektivní uložení nejbližších k sousedů, implementaci dvoustranné fronty (*dequeue*) fixní délky `FastDequeue` pro implementaci posuvného okna pro `LB_Keogh` a funkci pro výpočet euklidovské vzdálenosti. Strukturu celého modulu je možné najít znázorněnou na Obrázku 3.3.



Obrázek 3.3 Struktura modulu `KNNDTW`

`DataSets` Poslední modul poskytuje nástroje pro stahování a načítání *datasetů*. Toho dosahuje skrze modul `Loaders`. V tomto modulu se nachází pouze abstraktní typy určující specifický soubor *datasetů* (respektive jeho „loader“). V modulu `_Loaders` se nachází funkce pro práci se soubory *datasetů* a moduly s přetíženou verzí těchto funkcí pro daný „loader“, konkrétně `list_available_datasets`, která vypíše všechny dostupné *datasety* skrz daný typ „loaderu“, a `load_dataset`, která *dataset* stáhne, pokud to je nutné, a načte ze souboru. Jedním souborem *datasetů* je například `UCRArchive`, jehož „loader“ umožňuje automatické stažení celého archivu časových řad UCR (viz Sekce 2.3.3), jeho zpracování a načtení ze souboru pomocí funkce `read_ts_file` z modulu `_Reader`. Tento archiv se stahuje jako ZIP soubor, který je nutné rozbalit, o což se stará funkce `unzip` z modulu `_Utils`, a následně přečíst jeho data ze souborů ve formátu `ts` File Format v1.0[14]. V rámci modulu `_Loader` se vyskytuje i neveřejná funkce `ts_linear_interpolate_missing!` pro interpolaci a obousměrnou extrapolaci chybějících hodnot, která byla využita pouze v rámci vývoje, ne přímo v této práci. Kompletní strukturu celého modulu je možné najít na Obrázku 3.4.

[14] http://www.sktime.net/en/v0.18.0/api_reference/file_specifications/ts.html

Obrázek 3.4 Struktura modulu `DataSets`

Testy *Unit testy*, které jsou součástí projektu, (nejsou ale přímo součástí samotného balíčku). Tyto testy mají za cíl ověřit alespoň základní funkčnost jednotlivých částí algoritmů, zpravidla na malých oddělených celcích. Porovnávané hodnoty v testech vychází zpravidla z ručních výpočtů či z výsledků běhu totožných algoritmů v Python balíčku sktime, který je v tomto případě brán jako zdroj pravdy. Takto je možné (alespoň částečně) zaručit stálou správnost těchto algoritmů v případě změn jejich zdrojového kódu. Tyto testy staví na balíčku `Test` ze standardní knihovny jazyku Julia.

Ukázky použití V projektu je možné najít také ukázky použití, které stejně jako testy nejsou ale součástí samotného balíčku. Skládají se z Pluto.jl[15] [79] notebooků s MWE (*Minimal Working Example*). Tyto ukázky obsahují příklady načítání *datasetů*, jejich zpracování a následné transformace, inicializaci KNN a DTW, respektive MiniRocket, modelu včetně zvolení *hyperparametrů*, a následné trénování modelu, predikci na testovací části *datasetu* a evaluaci výsledku.

Benchmarky Skripty pro testování výkonu jednotlivých algoritmů je možné najít ve složce `benchmarks`. Stejně jako testy nejsou přímo součástí samotného balíčku, ale nachází se v projektu jako takovém. Tyto skripty byly využity pro experimenty v následující Kapitole 4.

[15] https://plutojl.org/

Kapitola 4

Experimenty

Tato kapitola se bude zabývat experimenty, měřeními a porovnáním výkonu implementací algoritmů *k-Nearest Neighbors* s *Dynamic Time Warping* a MiniRocket, které je možné najít v této práci a implementací těchto algoritmů v balíčku sktime.

4.1 Testovací prostředí

Aby byla zachována reprodukovatelnost všech testů a měření, tak je nutné představit testovací prostředí (někdy také testovací *setup*). Testovacím prostředím je myšlena kombinace hardwaru, softwaru a jejich nastavení, na kterém byly prováděny testy.

Hardware Veškerý hardware je provozován ve svém továrním nastavení a není nijak přetaktován či upraven, vyjma výjimek zde zmíněných. V případě procesoru (CPU) je povolena virtualizace Intel® VT, Hyper-Threading a omezení spotřeby a limity spotřeby jsou v BIOSu (nastavením) vypnuty či zvýšeny na 250 W. Operační paměti (RAM) mají zapnutý svůj tovární XMP profil nastavující jejich takt a časování na 3600 MHz a CL17. Grafická karta (GPU) je zmíněna pro utvoření kompletního obrazu o testovacím prostředí, ačkoliv v této práci nebyla využita pro žádné výpočty.

CPU	Intel® Core™ i7-11700[1], 2.50 GHz, 8 jader, 16 vláken
RAM	Kingston® Fury™ Beast DDR4 8GB@3600MHz/CL17[2], 32 GB (4×8 GB)
GPU	MSI nVidia® GeForce RTX™ 3070 VENTUS 3X 8G OC LHR[3]
SSD	Samsung 980 PCIe® 3.0 NVMe® SSD 500GB[4]
MBO	ASUS® TUF™ GAMING B560M-PLUS WIFI[5]

Software Software, tedy operační systém (OS), knihovny, balíčky a programy, který byl používán v rámci testování, se vyskytoval v následujících verzích, není li zmíněno jinak.

OS	Manjaro®[6] `2023-05-07`
Kernel	`6.1.26-1-MANJARO x86_64 GNU/Linux`
Python	`3.10.10`
Julia	`1.8.5`
sktime	`0.18.0`
scikit-learn	`1.2.1`
numba	`0.55.1`

4.2 Testovací *datasety*

Pro úvodní testování, takzvaný „*bake off*“ test, bylo použito 113 *datasetů* z archivu časových řad UCR, viz Sekce 2.3.3. Ze 128 *datasetů* bylo vynecháno 15 následujících *datasetů* z následujících důvodů:

`AllGestureWiimoteX`	rozdílné délky časových řad
`AllGestureWiimoteY`	rozdílné délky časových řad
`AllGestureWiimoteZ`	rozdílné délky časových řad
`DodgerLoopDay`	chybějící hodnoty
`DodgerLoopGame`	chybějící hodnoty
`DodgerLoopWeekend`	chybějící hodnoty
`GestureMidAirD1`	rozdílné délky časových řad
`GestureMidAirD2`	rozdílné délky časových řad
`GestureMidAirD3`	rozdílné délky časových řad
`GesturePebbleZ1`	rozdílné délky časových řad
`GesturePebbleZ2`	rozdílné délky časových řad
`MelbournePedestrian`	rozdílné délky, chybějící hodnoty
`PickupGestureWiimoteZ`	rozdílné délky časových řad
`PLAID`	rozdílné délky časových řad
`ShakeGestureWiimoteZ`	rozdílné délky časových řad

Ačkoliv *k-Nearest Neighbors* společně s *Dynamic Time Warping* podporuje časové řady různých délek, MiniRocket se již takovou vlastností chlubit nemůže. Aby přístup k testování zůstal totožný napříč všemi metodami a algoritmy, byly tyto *datasety* vyřazeny a nevyskytují se tedy v rámci těchto experimentů. *Datasety*, které obsahují chybějící hodnoty, jsou vynechány u obou algoritmů ze stejného důvodu, tedy protože ani jeden z těchto algoritmů neumí pracovat s chybějícími hodnotami. V rámci vývoje této práce vznikla funkce `ts_linear_interpolate_missing!` pro lineární interpolaci a extrapolaci chybějících hodnot (viz Sekce 3.3), ale její využití, testování a obecně volba vhodné interpolační techniky je mimo rozsah této práce. Vynechání těchto *datasetů* nebude mít na testování výkonu algoritmů výrazný dopad, jelikož se svojí velikostí a délkou svých časových řad neliší nijak výrazně od zbývajících *datasetů*.

Datasety, které vyžadují déle jak 10 minut běhu v rámci své testovací či trénovací fáze (odděleně), mohou být přeskočeny. Tato skutečnost bude zanesena do výsledků jako políčko tabulky bez číselné hodnoty.

Pro následné testování bylo z těchto 113 *datasetů* vybráno šest *datasetů*, které je možné najít na Tabulce 4.1. Tyto *datasety* reprezentují výběr, který zastupuje *datasety* různých délek časových řad, od velmi krátkých (`Chinatown`) po dlouhé (`Lightning2`),

s velkým (FacesUCR) i malým množstvím testovacích a trénovacích vzorků a i velkým i malým množstvím tříd. Mimo těchto šest *datasetů* budou probíhat testy i na náhodně generovaných datech.

Dataset	Train	Test	Třídy	Délka
Chinatown	20	343	2	24
CricketX	390	390	12	300
DistalPhalanxOutlineCorrect	600	276	2	80
FacesUCR	200	2050	14	131
GunPointMaleVersusFemale	135	316	2	150
Lightning2	60	61	2	637

Tabulka 4.1 *Datasety*, které byly vybrány pro testování.

4.3 Pozorování

V této sekci budou shrnuta pozorování z proběhlých experimentů. Tato pozorování je možná rozdělit do několika nadcházejících sekcí, z nichž se každá zaměřuje na jinou část implementace.

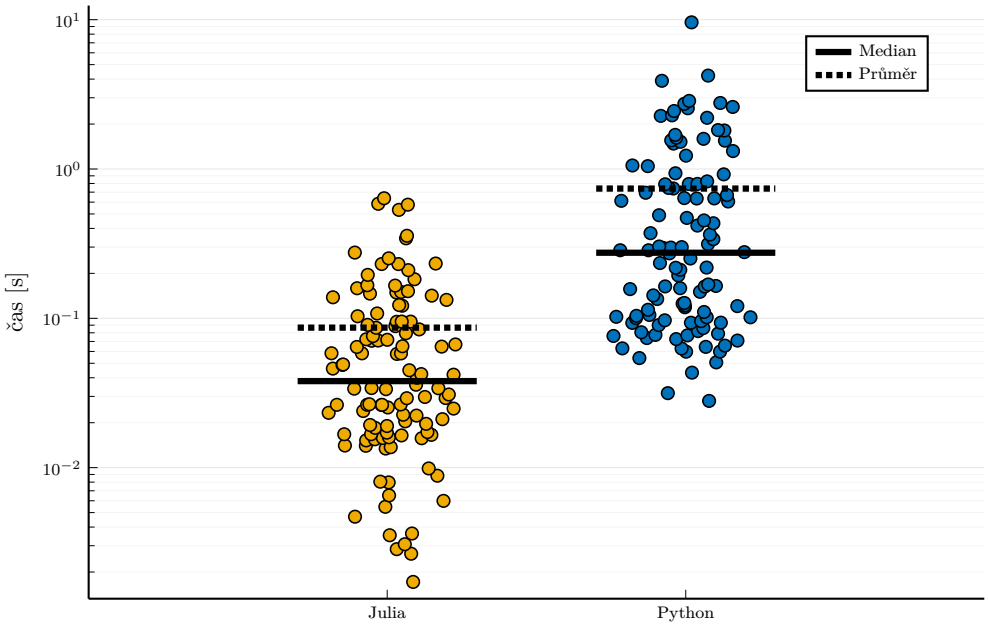
***Bake off* test** Ve tomto velkém „*bake off*“ testu, jinak také „rozstřelu“, bylo testováno všech 113 *datasetů* na všech jádrech procesoru. Podrobné výsledky tohoto testu je možné najít v Příloze A.1, výsledky na vybraných *datasetech* jsou v následující Tabulce 4.2. MiniRocket byl použit se svými původními parametry 9996 generovaných příznaků s maximálně 32 *dilatacemi* na *kernel*, zatím co *k-Nearest Neighbors* byl s $k = 1$ (ted 1NN) a *Dynamic Time Warping* byl omezen na Sakoe–Chiba pásmo o poloměru 10.

Dataset / čas $[s]$	knn.jl	minirocket.jl	minirocket.py	knn.py
Chinatown	0.0032490	0.0017170	0.0315433	0.0236043
CricketX	0.7813460	0.0650170	0.2724323	21.371315
DistalPhalanxOutlineCorrect	0.2441630	0.0263690	0.1102595	2.2982366
FacesUCR	0.7501890	0.0223150	0.3638622	12.900559
GunPointMaleVersusFemale	0.1118310	0.0167400	0.1039046	1.9539508
Lightning2	0.0996130	0.0448910	0.1591657	2.4009832

Tabulka 4.2 Rychlost jednotlivých algoritmů na vybraných *datasetech*.

V případě MiniRocket zpracovala implementace v jazyku Julia z této práce jeden *dataset* v průměru za 0,0866 vteřiny se střední hodnotou 0,0380 vteřiny. Proti tom implementace z balíčku sktime trval trénink a predikce jednoho *datasetu* v průměru

0,7396 vteřiny se střední hodnotou 0,27496 vteřiny. Ve výsledku je tedy implementace algoritmu MiniRocket z této práce v průměru 8,5krát rychlejší, než její protějšek z balíčku sktime. Graf ukazující rychlost na všech 113 *bake off datasetech* lze najít na obrázku 4.1.

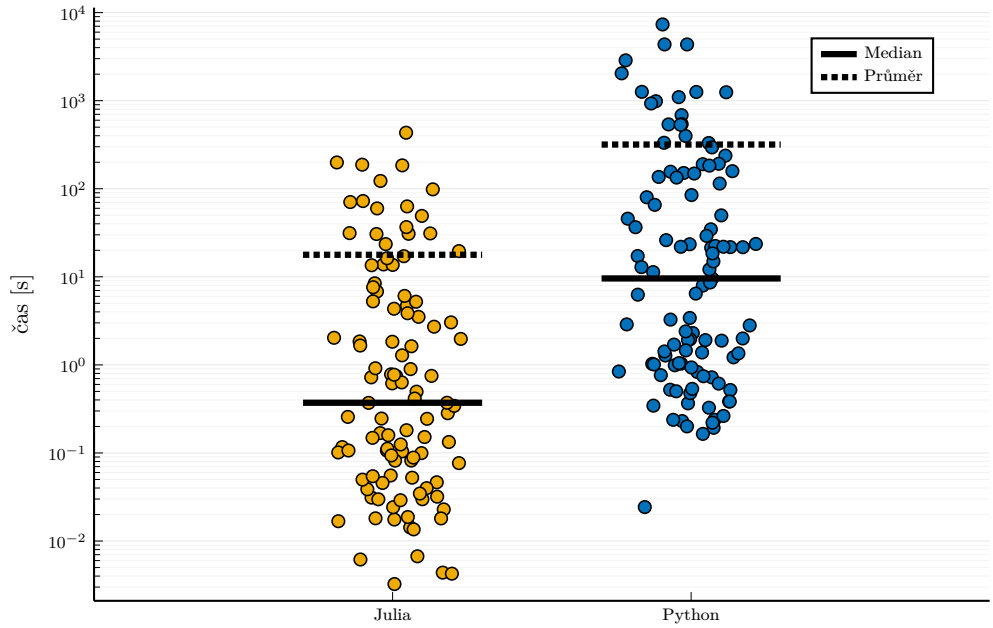


Obrázek 4.1 Porovnání rychlosti algoritmu MiniRocket v *bake off* testu

Implementace *k-Nearest Neighbors* a *Dynamic Time Warping*, pro tento test se Sakoe–Chiba pásmem o poloměru 10, si vedla ještě lépe. Implementace z balíčku sktime nebyla schopna v čase pod deset minut poradit s pěti *datasety* a po spodní hranici deseti minut běhu, zpravidla však i delší době, byl běh nad těmito *datasety* ukončen. Výpis takových *datasetů* je možné najít v tabulce v Příloze A.2, kde jsou daným algoritmem nedokončené *datasety* značeny čárou místo času v sekundách. Po odebrání výsledků, kde sktime implementace nedokončila svůj běh, byla průměrná doba zpracování *datasetu* 317,676, respektive 17,98 vteřiny v neprospěch implementace z balíčku sktime v jazyce Python, střední doba pak byla 9,5861, respektive 0,3716 vteřiny, opět v neprospěch balíčku sktime. Implementace KNN s DTW se Sakoe-Chiba pásmem je v této implementaci tedy 17,8krát rychlejší.

Jedním z důvodů, proč tomu tak je, může být chybějící podpora pro vícevláknový výpočet vzdáleností mezi jednotlivými vzorky, kdy implementace *k-Nearest Neighbors* z balíčku sktime prvně pouze jedním vláknem vypočítá vzdálenost mezi všemi páry vzorků z trénovacího i testovacího *datasetu* v čistém Pythonu[7], kterou následně předá již efektivnímu algoritmu pro nalezení k nejbližších sousedů. Bohužel výpočet vzdáleností mezi všemi páry vzorků z testovacího plus trénovacího *datasetu* je zdaleka nejnáročnější součást celého algoritmu.

[7] https://github.com/sktime/sktime/blob/v0.18.0/sktime/distances/_numba_utils.py#L52

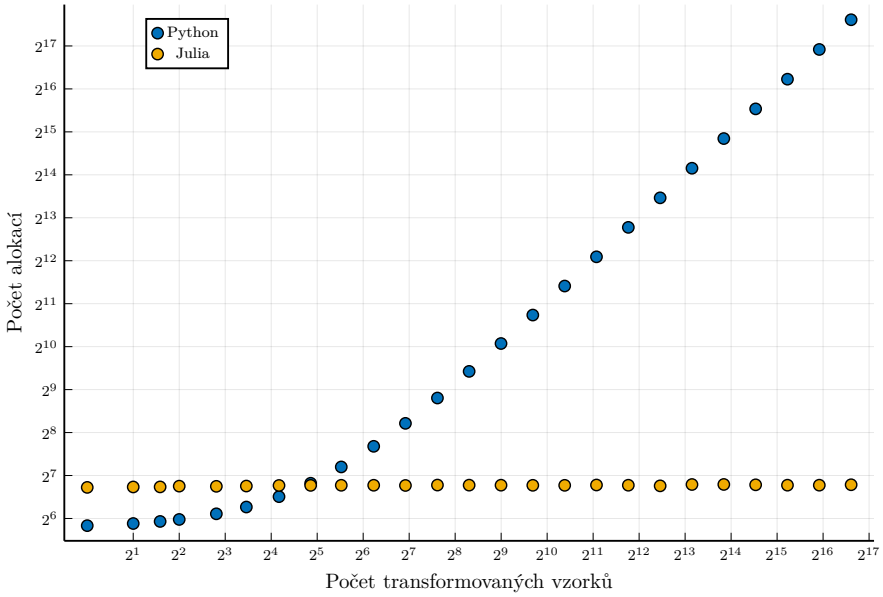


Obrázek 4.2 Porovnání rychlosti algoritmu *k-Nearest Neighbors* a *Dynamic Time Warping* se Sakoe–Chiba pásmem poloměru 10 v *bake off* testu

Omezení počtu alokací paměti Oproti implementaci *k-Nearest Neighbors* je implementace MiniRocket v balíčku sktime velmi efektivní. Dokáže totiž využít všechna jádra procesoru po celou dobu běhu transformace a samotný kód je *just-in-time* kompilovaný pomocí knihovny Numba (viz 2.3.1.2). I přes to je však v průměru 8,5krát pomalejší než implementace v jazyku Julia v této práci. Jedním z důvodů může být množství alokací, které MiniRocket z balíčku sktime provádí. Aby bylo možné *debuggovat* samotný balíček Numba a kód jím zkompilovaný, tak Numba obsahuje funkci `numba.core.runtime.rtsys.get\_allocation_stats()`, která vrací informace o počtu interních alokací a dalších operací s pamětí. Tuto funkci zle využít i pro monitorování počtu alokací provedených sktime implementací algoritmu MiniRocket. Díky tomu ní být proveden i následující test, viz Obrázek 4.3, kde lze spatřit, že počet alokací roste úměrně vůči počtu transformovaných vzorků.

Naopak implementace, která vznikla v rámci této práce, provádí pouze konstantní počet alokací paměti, kde mírné rozdíly v rádu jednotek alokací (celkový počet se pohybuje mezi 104 až 114 alokacemi) mezi jednotlivými běhy jsou interní pomocné struktury knihoven Julia, které vznikají spouštěním více vláken. Tento přístup k omezení počtu alokací je jeden z faktorů, proč implementace v Julia v této práci dosahuje výrazně vyšší rychlosti. Podobný přístup je implementován i v rámci implementace *k-Nearest Neighbors*, respektive výpočtu vzdáleností DTW, DTW se Sakoe–Chiba pásmem a DTW s Itakurovým parallelogramem, kdy je matice vzdáleností sdílená, respektive alokovaná paměť je „kešovaná“ mezi jednotlivými voláními funkce `dtw!(...)`.

„Kvazi-determinizace“ MiniRocket, jak bylo již zmíněno v Sekci 3.2.2, byl obohacen o nový *hyperparametr* `shuffled`, který značí, že trénovací *dataset*, který je na



Obrázek 4.3 Počet provedených alokací algoritmem MiniRocket z balíčku sktime v závislosti na počtu transformovaných časových řad

vstupu, je již náhodně seřazen. V takovém případě není nutné vybírat vzorky z trénovacího *datasetu* náhodně, ale je možné ja procházet postupně v pořadí, ve kterém již jsou. Myšlenka, že nenáhodný přístup do paměti bude podávat lepší výsledky, byla bohužel mylná. Při testování na náhodných datech, nehledě na velikost *datasetu* i délku časových řad byla vždy rychlost v podstatě totožná. Samotné trénování je navíc výrazně rychlejší proces než transformace a zisk z této úpravy by byl nejspíše minimální.

Porovnání výkonu variant DTW Varianta se Sakoe–Chiba pásmem byla testována již dříve v rámci *bake off* testu. Zbývá tedy otestovat *Dynamic Time Warping* ve variantě s Itakurovým parallelogramem a čisté, neupravené DTW spolu s 1NN klasifikátorem. Tentokrát pro změnu s omezením pouze na 1 jádro procesoru, aby byla dorovnána výhoda Julia implementace s plnohodnotnou podporu paralelizace výpočtů. Výsledné časy jsou vidět na Tabulce 4.3. I v tomto případě implementace v této práci poráží tu z balíčku sktime, ačkoliv nyní už „jen“ ∼1,4krát v případě DTW a ∼3,3krát.

Dataset / doba běhu [*s*]	DTW.jl	DTW.py	DTW Itak.jl	DTW Itak.py
`Chinatown`	0.02386	0.03112	0.00628	0.02816
`CricketX`	44.10731	61.52051	18.49908	61.64181
`DistalPhalanxOutlineCorrect`	4.42095	4.89648	1.44837	4.66925
`FacesUCR`	22.88038	31.71154	9.45112	30.67438
`GunPointMaleVersusFemale`	3.8535	4.42965	1.28045	4.31253
`Lightning2`	4.74042	6.74399	1.65544	6.80855

Tabulka 4.3 Porovnání DTW a DTW s Itakurovým parallelogramem na vybraných *datasetech*.

Kapitola 5

Závěr

Tato práce se zaměřuje na problematiku časových řad a jejich efektivní klasifikace. Po teoretické stránce v rámci Kapitoly 1 proběhlo uvedení do problematiky, byly popsány nyní dostupné principy a metody pro klasifikaci časových řad, jako jsou například metody založené na vzdálenostech, *shapeletech*, slovníkové metody a metody využívající hluboké strojové učení a neuronové sítě. V následující kapitole byly podrobně rozebrány algoritmy *k-Nearest Neighbors* a *Dynamic Time Warping*, a to včetně svých variant omezujících prohledávaný prostor jako je DTW omezené na Sakoe–Chiba pásmo či na Itakurův parallelogram a navíc byla představena a následně i implementována funkce LB_Keogh pro nalezení spodní meze vzájemné vzdálenosti dvou časových řad. Byl podrobně rozebrán algoritmus MiniRocket včetně svého předchůdce Rocket, z čehož vyústily optimalizace implementace těchto algoritmů v praktické části této práce. Kapitola obsahuje i popis rozdílů mezi programovacími jazyky Python a Julia, včetně praktických zdrojového kódu znázorňujících některé rozdíly, a nástroje s těmito a tématem této jazyky spojené, jako je sktime či *framewokr* Machine Learning Julia nebo archiv časových řad spravovaný Kalifornskou univerzitou v Riverside.

V praktické části této práce byl implementován, popsán a zdokumentován balíček pro klasifikaci časových řad implementující nejen efektivní verze algoritmů MiniRocket, *k-Nearest Neighbors* a *Dynamic Time Warping* včetně již zmíněných variant implementujících Sakoe–Chiba pásmo a Itakurův parallelogram, ale i nástroj pro automatické stahování a načítání *datasetů* z archivu UCR. Balíček byl vybaven *unit testy*, které ověřují nejen jeho správný chod, ale i shodu s implementacemi z balíčku sktime. Mimo to je možné najít v projektu také nástroje a skripty pro usnadnění testování výkonu jak implementace z této práce, tak i implementace těchto algoritmů z balíčku sktime, a také ukázky použití ve formě Pluto.jl notebooků.

Na konec byla výkonnost implementace algoritmů MiniRocket, *k-Nearest Neighbors* a *Dynamic Time Warping*, včetně svých již zmíněných variant, potvrzena na 113 *datasetech* z archivu časových řad UCR, kdy algoritmy implementované v jazyku Julia v této práci porážely své protějšky z balíčku sktime s více jak trojnásobnou, v některých testech více jak sedmnáctinásobnou rychlostí. Zároveň byly nastíněny možné příčiny problémů omezujících výkon implementace v balíčku sktime.

Všechny cíle této bakalářské práce byly splněny. Tato práce pokládá základní kámen

pro vznik komunity zabývající se výzkumem a klasifikací časových řad v jazyku Julia. Praktická část práce je snadno rozšiřitelná o další modely a funkce a do budoucna lze uvažovat o rozšíření tohoto balíčku až na srovnatelnou úroveň se svými protějšky jako je například sktime, či využít tento balíček jako podporu pro vědecké výpočty jiného výzkumu.

Dodatek A

Tabulka výsledků *bake off* testu

A.1 Tabulka výsledků *bake off* testu seřazena dle času

Dataset / doba běhu [s]	knn.jl	minirocket.jl	minirocket.py	knn.py
ACSF1	3.5124060	34.6290801	0.1227210	0.4332611
Adiac	0.4965120	7.9561813	0.0309010	0.1938193
ArrowHead	0.1042180	0.7223939	0.0137110	0.0934526
BME	0.0142840	0.2302330	0.0054670	0.0507012
Beef	0.0818030	0.3658179	0.0252940	0.0932112
BeetleFly	0.0067130	0.2380093	0.0264930	0.0737723
BirdChicken	0.0061710	0.2378726	0.0248640	0.0768564
CBF	0.0554960	0.8304981	0.0059910	0.1632620
Car	0.1331900	1.8847464	0.0359240	0.1267132
Chinatown	0.0032490	0.0242966	0.0017170	0.0315433
ChlorineConcentration	4.3395410	84.8223516	0.0490700	0.8258805
CinCECGTorso	23.5262390	237.6389432	0.1077880	2.2874450
Coffee	0.0043790	0.2003549	0.0140640	0.0541807
Computers	2.7134570	45.7185578	0.0643040	0.4523194
CricketX	0.7813460	21.3723815	0.0650170	0.2724323
CricketY	0.7481770	21.7342668	0.0710270	0.2774871
CricketZ	0.7224100	21.6920926	0.0583620	0.2857232
Crop	70.6336810	684.9721217	0.2756030	1.5925969
DiatomSizeReduction	0.0311380	1.0285799	0.0152250	0.1569221
DistalPhalanxOutlineAgeGroup	0.1680750	0.8412639	0.0190050	0.0821407
DistalPhalanxOutlineCorrect	0.2441630	2.2994299	0.0263690	0.1102595
DistalPhalanxTW	0.1590560	0.7648275	0.0184840	0.0900523

Dataset / doba běhu [*s*]	knn.jl	minirocket.jl	minirocket.py	knn.py
ECG200	0.0175520	0.2630450	0.0080530	0.0432683
ECG5000	4.7017010	80.0669037	0.0400170	0.7946411
ECGFiveDays	0.0464720	0.7435213	0.0065170	0.1635468
EOGHorizontalSignal	31.3505250	330.6229032	0.1656600	1.0559549
EOGVerticalSignal	30.6231810	332.6105553	0.1665430	1.0452767
Earthquakes	0.6166770	17.2224010	0.0646500	0.3026572
ElectricDevices	98.4185830	1256.4592072	0.5325380	1.8224823
EthanolLevel	122.6290690	1246.6431447	0.2522060	1.8108612
FaceAll	1.8314920	29.1897284	0.0582380	0.3718435
FaceFour	0.0181120	0.5222983	0.0204160	0.0806532
FacesUCR	0.7501890	12.9013850	0.0223150	0.3638622
FiftyWords	0.8959500	23.6229899	0.0707720	0.2855508
Fish	0.2826530	9.5861377	0.0487770	0.2187498
FordA	49.1454420	2043.7063263	0.5847670	2.5516205
FordB	30.7972350	1259.7162868	0.5769450	2.2686895
FreezerRegularTrain	2.0309580	65.6520535	0.0338010	0.9362636
FreezerSmallTrain	0.3704340	12.1326233	0.0171560	0.9207259
Fungi	0.0767040	0.3444685	0.0088300	0.0789578
GunPoint	0.0242130	0.4779612	0.0165820	0.0643897
GunPointAgeSpan	0.1065340	1.9959429	0.0154860	0.1015901
GunPointMaleVersusFemale	0.1118310	1.9551212	0.0167400	0.1039046
GunPointOldVersusYoung	0.1165360	1.9120903	0.0157980	0.1016863
Ham	0.1484200	3.4114439	0.0341290	0.1505123
HandOutlines	432.6999970	7337.3339259	0.6369420	3.8977160
Haptics	8.4424580	136.2312047	0.0904010	0.6061967
Herring	0.0523840	1.9116560	0.0291730	0.1206887
HouseTwenty	3.0388890	36.5872043	0.1383280	0.4902695
InlineSkate	31.1100610	397.5670027	0.1463930	1.3186267
InsectEPGRegularTrain	0.3426960	8.6188653	0.0423180	0.2515971
InsectEPGSmallTrain	0.0937930	3.2686930	0.0340230	0.2180644
InsectWingbeatSound	1.8459570	49.8353464	0.0292360	0.6122943
ItalyPowerDemand	0.0228630	0.1930116	0.0026520	0.0627793
LargeKitchenAppliances	6.8057760	150.8791692	0.0760700	0.6386808
Lightning2	0.0996130	2.4018217	0.0448910	0.1591657
Lightning7	0.0298360	1.0337112	0.0239520	0.0969771
Mallat	19.5916850	294.7119757	0.0582900	2.4454727
Meat	0.0398610	1.2778811	0.0232950	0.1023641
MedicalImages	0.4147070	6.4487977	0.0263470	0.1424132
MiddlePhalanxOutlineAgeGroup	0.1013660	0.9908525	0.0167470	0.0725949
MiddlePhalanxOutlineCorrect	0.2567250	2.8061616	0.0262730	0.1053139
MiddlePhalanxTW	0.1065940	1.0109725	0.0157340	0.0709588
MixedShapesRegularTrain	184.2632940	2868.9787878	0.1328620	2.8584144
MixedShapesSmallTrain	36.7054600	537.7078236	0.0668430	2.6040415
MoteStrain	0.0387070	0.5015874	0.0036220	0.1685440

Dataset / doba běhu [*s*]	knn.jl	minirocket.jl	minirocket.py	knn.py
NonInvasiveFetalECGThorax1	198.9212820	4350.0872084	0.3570980	2.7321988
NonInvasiveFetalECGThorax2	187.6074810	4351.9137488	0.3434250	2.7675519
OSULeaf	0.3716870	14.9524090	0.0460600	0.2344929
OliveOil	0.0187420	0.6128975	0.0297350	0.0957615
PhalangesOutlinesCorrect	1.9723890	22.3888015	0.0723730	0.2959200
Phoneme	59.8018600	985.9801570	0.1032420	2.2029288
PigAirwayPressure	13.8260430	189.9351160	0.1490060	0.7470454
PigArtPressure	13.6520190	192.0954358	0.1501070	0.7937536
PigCVP	13.5352940	183.0314968	0.1519380	0.7406125
Plane	0.0297910	0.5343121	0.0098790	0.0629067
PowerCons	0.0820140	1.4210362	0.0134230	0.0935518
ProximalPhalanxOutlineAgeGroup	0.1516850	1.2188252	0.0226160	0.0761960
ProximalPhalanxOutlineCorrect	0.2459460	2.8828157	0.0211200	0.1185955
ProximalPhalanxTW	0.1249470	1.3828402	0.0159670	0.0774672
RefrigerationDevices	5.2635180	114.6339802	0.0947710	0.6345766
Rock	1.2839880	23.4285085	0.1827030	0.4180366
ScreenType	5.2136270	158.3470142	0.0843090	0.6932221
SemgHandGenderCh2	63.0198960	929.6702271	0.1954220	1.6218216
SemgHandMovementCh2	72.5121320	1097.0286711	0.2309540	1.5214046
SemgHandSubjectCh2	72.2227910	—	0.2327770	1.5483560
ShapeletSim	0.0456630	1.6996504	0.0335500	0.1647131
ShapesAll	3.8599800	156.1104312	0.0951180	0.6693989
SmallKitchenAppliances	6.0705420	148.9203418	0.0718170	0.6353679
SmoothSubspace	0.0042610	0.1646525	0.0046930	0.0279870
SonyAIBORobotSurface1	0.0167990	0.2206865	0.0028450	0.0860554
SonyAIBORobotSurface2	0.0318930	0.3846250	0.0030700	0.1140658
StarLightCurves	1232.7551970	—	0.2101550	9.5970638
Strawberry	0.9160960	21.9216792	0.0864300	0.2976729
SwedishLeaf	0.6326300	11.3107808	0.0265840	0.2112808
Symbols	0.1812890	6.2696706	0.0196520	0.4703896
SyntheticControl	0.0885700	0.9355330	0.0140110	0.0598306
ToeSegmentation1	0.0497730	1.3517774	0.0173190	0.1252900
ToeSegmentation2	0.0290670	1.0465392	0.0193080	0.1217856
Trace	0.0544230	1.4610394	0.0263560	0.1002572
TwoLeadECG	0.0345970	0.5187592	0.0035260	0.1347003
TwoPatterns	7.6181640	133.9902002	0.0577470	0.7900410
UMD	0.0135660	0.3262645	0.0079640	0.0597752
UWaveGestureLibraryAll	394.7505500	—	0.2308210	4.2215589
UWaveGestureLibraryX	17.1548640	541.6563766	0.1589820	1.4798226
UWaveGestureLibraryY	16.0871090	—	0.1418900	1.5558431
UWaveGestureLibraryZ	16.2497830	532.3892986	0.1214750	1.6925875
Wafer	13.4080350	—	0.0794620	1.2285306
Wine	0.0180560	0.3838367	0.0164370	0.0656436
WordSynonyms	0.7732720	21.9461508	0.0419300	0.2997964

Dataset / *doba běhu* [*s*]	knn.jl	minirocket.jl	minirocket.py	knn.py
Worms	1.6585240	18.4439780	0.0886370	0.3388049
WormsTwoClass	1.6280060	26.0922206	0.0950460	0.3138163

A.2 Tabulka výsledků *bake off* testu seřazena dle času

Dataset / *doba běhu* [*s*]	knn.jl	knn.py	minirocket.jl	minirocket.py
StarLightCurves	1232.7551970	—	0.2101550	9.5970638
HandOutlines	432.6999970	7337.3339259	0.6369420	3.8977160
UWaveGestureLibraryAll	394.7505500	—	0.2308210	4.2215589
NonInvasiveFetalECGThorax1	198.9212820	4350.0872084	0.3570980	2.7321988
NonInvasiveFetalECGThorax2	187.6074810	4351.9137488	0.3434250	2.7675519
MixedShapesRegularTrain	184.2632940	2868.9787878	0.1328620	2.8584144
EthanolLevel	122.6290690	1246.6431447	0.2522060	1.8108612
ElectricDevices	98.4185830	1256.4592072	0.5325380	1.8224823
SemgHandMovementCh2	72.5121320	1097.0286711	0.2309540	1.5214046
SemgHandSubjectCh2	72.2227910	—	0.2327770	1.5483560
Crop	70.6336810	684.9721217	0.2756030	1.5925969
SemgHandGenderCh2	63.0198960	929.6702271	0.1954220	1.6218216
Phoneme	59.8018600	985.9801570	0.1032420	2.2029288
FordA	49.1454420	2043.7063263	0.5847670	2.5516205
MixedShapesSmallTrain	36.7054600	537.7078236	0.0668430	2.6040415
EOGHorizontalSignal	31.3505250	330.6229032	0.1656600	1.0559549
InlineSkate	31.1100610	397.5670027	0.1463930	1.3186267
FordB	30.7972350	1259.7162868	0.5769450	2.2686895
EOGVerticalSignal	30.6231810	332.6105553	0.1665430	1.0452767
CinCECGTorso	23.5262390	237.6389432	0.1077880	2.2874450
Mallat	19.5916850	294.7119757	0.0582900	2.4454727
UWaveGestureLibraryX	17.1548640	541.6563766	0.1589820	1.4798226
UWaveGestureLibraryZ	16.2497830	532.3892986	0.1214750	1.6925875
UWaveGestureLibraryY	16.0871090	—	0.1418900	1.5558431
PigAirwayPressure	13.8260430	189.9351160	0.1490060	0.7470454
PigArtPressure	13.6520190	192.0954358	0.1501070	0.7937536
PigCVP	13.5352940	183.0314968	0.1519380	0.7406125
Wafer	13.4080350	—	0.0794620	1.2285306
Haptics	8.4424580	136.2312047	0.0904010	0.6061967
TwoPatterns	7.6181640	133.9902002	0.0577470	0.7900410
LargeKitchenAppliances	6.8057760	150.8791692	0.0760700	0.6386808
SmallKitchenAppliances	6.0705420	148.9203418	0.0718170	0.6353679
RefrigerationDevices	5.2635180	114.6339802	0.0947710	0.6345766
ScreenType	5.2136270	158.3470142	0.0843090	0.6932221
ECG5000	4.7017010	80.0669037	0.0400170	0.7946411

Dataset / doba běhu [*s*]	knn.jl	minirocket.jl	minirocket.py	knn.py
ChlorineConcentration	4.3395410	84.8223516	0.0490700	0.8258805
ShapesAll	3.8599800	156.1104312	0.0951180	0.6693989
ACSF1	3.5124060	34.6290801	0.1227210	0.4332611
HouseTwenty	3.0388890	36.5872043	0.1383280	0.4902695
Computers	2.7134570	45.7185578	0.0643040	0.4523194
FreezerRegularTrain	2.0309580	65.6520535	0.0338010	0.9362636
PhalangesOutlinesCorrect	1.9723890	22.3888015	0.0723730	0.2959200
InsectWingbeatSound	1.8459570	49.8353464	0.0292360	0.6122943
FaceAll	1.8314920	29.1897284	0.0582380	0.3718435
Worms	1.6585240	18.4439780	0.0886370	0.3388049
WormsTwoClass	1.6280060	26.0922206	0.0950460	0.3138163
Rock	1.2839880	23.4285085	0.1827030	0.4180366
Strawberry	0.9160960	21.9216792	0.0864300	0.2976729
FiftyWords	0.8959500	23.6229899	0.0707720	0.2855508
CricketX	0.7813460	21.3723815	0.0650170	0.2724323
WordSynonyms	0.7732720	21.9461508	0.0419300	0.2997964
FacesUCR	0.7501890	12.9013850	0.0223150	0.3638622
CricketY	0.7481770	21.7342668	0.0710270	0.2774871
CricketZ	0.7224100	21.6920926	0.0583620	0.2857232
SwedishLeaf	0.6326300	11.3107808	0.0265840	0.2112808
Earthquakes	0.6166770	17.2224010	0.0646500	0.3026572
Adiac	0.4965120	7.9561813	0.0309010	0.1938193
MedicalImages	0.4147070	6.4487977	0.0263470	0.1424132
OSULeaf	0.3716870	14.9524090	0.0460600	0.2344929
FreezerSmallTrain	0.3704340	12.1326233	0.0171560	0.9207259
InsectEPGRegularTrain	0.3426960	8.6188653	0.0423180	0.2515971
Fish	0.2826530	9.5861377	0.0487770	0.2187498
MiddlePhalanxOutlineCorrect	0.2567250	2.8061616	0.0262730	0.1053139
ProximalPhalanxOutlineCorrect	0.2459460	2.8828157	0.0211200	0.1185955
DistalPhalanxOutlineCorrect	0.2441630	2.2994299	0.0263690	0.1102595
Symbols	0.1812890	6.2696706	0.0196520	0.4703896
DistalPhalanxOutlineAgeGroup	0.1680750	0.8412639	0.0190050	0.0821407
DistalPhalanxTW	0.1590560	0.7648275	0.0184840	0.0900523
ProximalPhalanxOutlineAgeGroup	0.1516850	1.2188252	0.0226160	0.0761960
Ham	0.1484200	3.4114439	0.0341290	0.1505123
Car	0.1331900	1.8847464	0.0359240	0.1267132
ProximalPhalanxTW	0.1249470	1.3828402	0.0159670	0.0774672
GunPointOldVersusYoung	0.1165360	1.9120903	0.0157980	0.1016863
GunPointMaleVersusFemale	0.1118310	1.9551212	0.0167400	0.1039046
MiddlePhalanxTW	0.1065940	1.0109725	0.0157340	0.0709588
GunPointAgeSpan	0.1065340	1.9959429	0.0154860	0.1015901
ArrowHead	0.1042180	0.7223939	0.0137110	0.0934526
MiddlePhalanxOutlineAgeGroup	0.1013660	0.9908525	0.0167470	0.0725949
Lightning2	0.0996130	2.4018217	0.0448910	0.1591657

Dataset / doba běhu $[s]$	knn.jl	minirocket.jl	minirocket.py	knn.py
InsectEPGSmallTrain	0.0937930	3.2686930	0.0340230	0.2180644
SyntheticControl	0.0885700	0.9355330	0.0140110	0.0598306
PowerCons	0.0820140	1.4210362	0.0134230	0.0935518
Beef	0.0818030	0.3658179	0.0252940	0.0932112
Fungi	0.0767040	0.3444685	0.0088300	0.0789578
CBF	0.0554960	0.8304981	0.0059910	0.1632620
Trace	0.0544230	1.4610394	0.0263560	0.1002572
Herring	0.0523840	1.9116560	0.0291730	0.1206887
ToeSegmentation1	0.0497730	1.3517774	0.0173190	0.1252900
ECGFiveDays	0.0464720	0.7435213	0.0065170	0.1635468
ShapeletSim	0.0456630	1.6996504	0.0335500	0.1647131
Meat	0.0398610	1.2778811	0.0232950	0.1023641
MoteStrain	0.0387070	0.5015874	0.0036220	0.1685440
TwoLeadECG	0.0345970	0.5187592	0.0035260	0.1347003
SonyAIBORobotSurface2	0.0318930	0.3846250	0.0030700	0.1140658
DiatomSizeReduction	0.0311380	1.0285799	0.0152250	0.1569221
Lightning7	0.0298360	1.0337112	0.0239520	0.0969771
Plane	0.0297910	0.5343121	0.0098790	0.0629067
ToeSegmentation2	0.0290670	1.0465392	0.0193080	0.1217856
GunPoint	0.0242130	0.4779612	0.0165820	0.0643897
ItalyPowerDemand	0.0228630	0.1930116	0.0026520	0.0627793
OliveOil	0.0187420	0.6128975	0.0297350	0.0957615
FaceFour	0.0181120	0.5222983	0.0204160	0.0806532
Wine	0.0180560	0.3838367	0.0164370	0.0656436
ECG200	0.0175520	0.2630450	0.0080530	0.0432683
SonyAIBORobotSurface1	0.0167990	0.2206865	0.0028450	0.0860554
BME	0.0142840	0.2302330	0.0054670	0.0507012
UMD	0.0135660	0.3262645	0.0079640	0.0597752
BeetleFly	0.0067130	0.2380093	0.0264930	0.0737723
BirdChicken	0.0061710	0.2378726	0.0248640	0.0768564
Coffee	0.0043790	0.2003549	0.0140640	0.0541807
SmoothSubspace	0.0042610	0.1646525	0.0046930	0.0279870
Chinatown	0.0032490	0.0242966	0.0017170	0.0315433

Bibliografie

1. KAIST; UNIST; POSTECH. 데이터 사이언스 경진대회 [online]. 2022. [cit. 2023-03-12]. Dostupné z: https://www.datascience-contest.com/.
2. KAGGLE. *How to Use Kaggle - Simple Competitions* [online]. 2023. [cit. 2023-03-12]. Dostupné z: https://www.kaggle.com/docs/competitions.
3. LÖNING, Markus; BAGNALL, Anthony; GANESH, Sajaysurya; KAZAKOV, Viktor; LINES, Jason; KIRÁLY, Franz J. *sktime: A Unified Interface for Machine Learning with Time Series*. 2019. Dostupné z arXiv: 1909.07872 [cs.LG].
4. LÖNING, Markus; KIRÁLY, Franz; BAGNALL, Tony; MIDDLEHURST, Matthew; GANESH, Sajaysurya; OASTLER, George; LINES, Jason; WALTER, Martin; VIKTORKAZ; MENTEL, Lukasz; CHRISHOLDER; TSAPROUNIS, Leonidas; RNKUHNS; PARKER, Mirae; OWOSENI, Taiwo; ROCKENSCHAUB, Patrick; DANBARTL; JESELLIER; EENTICOTT-SHELL; GILBERT, Ciaran; BULATOVA, Guzal; LOVKUSH; SCHÄFER, Patrick; KHRAPOV, Stanislav; BUCHHORN, Katie; TAKE, Kejsi; SUBRAMANIAN, Shivansh; MEYER, Svea Marie; AIDENRUSHBROOKE; RICE, Beth. *sktime/sktime: v0.13.4*. Zenodo, 2022. Ver. v0.13.4. Dostupné z DOI: 10.5281/zenodo.7117735.
5. FOUNDATION, Python Software. *Python 3.11.3 documentation* [online]. 2023. [cit. 2023-04-23]. Dostupné z: https://docs.python.org/3.11/.
6. HARRIS, Charles R.; MILLMAN, K. Jarrod; WALT, Stéfan J. van der; GOMMERS, Ralf; VIRTANEN, Pauli; COURNAPEAU, David; WIESER, Eric; TAYLOR, Julian; BERG, Sebastian; SMITH, Nathaniel J.; KERN, Robert; PICUS, Matti; HOYER, Stephan; KERKWIJK, Marten H. van; BRETT, Matthew; HALDANE, Allan; RÍO, Jaime Fernández del; WIEBE, Mark; PETERSON, Pearu; GÉRARD-MARCHANT, Pierre; SHEPPARD, Kevin; REDDY, Tyler; WECKESSER, Warren; ABBASI, Hameer; GOHLKE, Christoph; OLIPHANT, Travis E. Array programming with NumPy. *Nature*. 2020, roč. 585, č. 7825, s. 357–362. ISSN 1476-4687. Dostupné z DOI: 10.1038/s41586-020-2649-2.
7. LAM, Siu Kwan; PITROU, Antoine; SEIBERT, Stanley. Numba: A LLVM-Based Python JIT Compiler. In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. Austin, Texas: Association for Computing Machine-

ry, 2015. LLVM '15. ISBN 9781450340052. Dostupné z DOI: 10.1145/2833157.2833162.

8. MAAS, Martin D. *Julia vs Numba and Cython: Looking Beyond Microbenchmarks* [online]. 2022. [cit. 2023-03-12]. Dostupné z: https://www.matecdev.com/posts/julia-python-numba-cython.html.
9. BEZANSON, Jeff; EDELMAN, Alan; KARPINSKI, Stefan; SHAH, Viral B. Julia: A Fresh Approach to Numerical Computing. *SIAM Review*. 2017, roč. 59, č. 1, s. 65–98. Dostupné z DOI: 10.1137/141000671.
10. HECKERT, N. Alan; FILLIBEN, James; CROARKIN, Carroll; HEMBREE, Barry; GUTHRIE, William; TOBIAS, Paul; PRINS, Jack. *Handbook 151: NIST/SEMATECH e-Handbook of Statistical Methods*. NIST Interagency/Internal Report (NISTIR), National Institute of Standards a Technology, Gaithersburg, MD, 2002. Dostupné z DOI: 10.18434/M32189.
11. CHATFIELD, Chris; XING, Haipeng. *The analysis of time series: an introduction with R*. CRC press, 2019. Dostupné z DOI: 10.1201/9781351259446.
12. HAYES, Adam. *What is a time series and how is it used to analyze data?* [online]. Investopedia, 2022 [cit. 2023-03-19]. Dostupné z: https://www.investopedia.com/terms/t/timeseries.asp.
13. BLOOMFIELD, Peter. *Fourier Analysis of Time Series: An Introduction, Second Edition*. 2000. Dostupné z DOI: 10.1137/1019120.
14. BROCKWELL Peter J.and Davis, Richard A. *Introduction to Time Series and Forecasting*. Cham: Springer International Publishing, 2016. ISBN 978-3-319-29854-2. Dostupné z DOI: 10.1007/978-3-319-29854-2.
15. ODHIAMBO, Nicholas M. Inflation Dynamics And Economic Growth In Tanzania: A Multivariate Time Series Model. *Journal of Applied Business Research (JABR)*. 2012, roč. 28, č. 3, s. 317–324. Dostupné z DOI: 10.19030/jabr.v28i3.6951.
16. PEIRIS, D. Ramanee; MCNICOL, James W. Modelling daily weather with multivariate time series. *Agricultural and Forest Meteorology*. 1996, roč. 79, č. 4, s. 219–231. ISSN 0168-1923. Dostupné z DOI: 10.1016/0168-1923(95)02282-1.
17. SHUMWAY, Robert H.; STOFFER, David S. *Time Series Analysis and Its Applications: With R Examples*. Springer International Publishing, 2017. ISBN 978-3-319-52452-8. Dostupné z DOI: 10.1007/978-3-319-52452-8_1.
18. IDREES, Sheikh Mohammad; ALAM, M. Afshar; AGARWAL, Parul. A Prediction Approach for Stock Market Volatility Based on Time Series Data. *IEEE Access*. 2019, roč. 7, s. 17287–17298. Dostupné z DOI: 10.1109/ACCESS.2019.2895252.
19. TANG, Hewei; ZHANG, Shang; ZHANG, Feifei; VENUGOPAL, Suresh. Time series data analysis for automatic flow influx detection during drilling. *Journal of Petroleum Science and Engineering*. 2019, roč. 172, s. 1103–1111. ISSN 0920-4105. Dostupné z DOI: 10.1016/j.petrol.2018.09.018.
20. OMENZETTER, Piotr; BROWNJOHN, James M. W. Application of time series analysis for bridge monitoring. *Smart Materials and Structures*. 2006, roč. 15, č. 1, s. 129. Dostupné z DOI: 10.1088/0964-1726/15/1/041.

21. CABALLERO BARAJAS, Karla L.; AKELLA, Ram. Dynamically Modeling Patient's Health State from Electronic Medical Records: A Time Series Approach. In: *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. Sydney, NSW, Australia: Association for Computing Machinery, 2015, s. 69–78. KDD '15. ISBN 9781450336642. Dostupné z DOI: 10.1145/2783258.2783289.

22. CHUAH, Mooi Choo; FU, Fen. ECG Anomaly Detection via Time Series Analysis. In: THULASIRAMAN, Parimala; HE, Xubin; XU, Tony Li; DENKO, Mieso K.; THULASIRAM, Ruppa K.; YANG, Laurence T. (ed.). *Frontiers of High Performance Computing and Networking ISPA 2007 Workshops*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, s. 123–135. ISBN 978-3-540-74767-3. Dostupné z DOI: 10.1007/978-3-540-74767-3_14.

23. WAH, Win; DAS, Sourav; EARNEST, Arul; LIM, Leo Kang Yang; CHEE, Cynthia Bin Eng; COOK, Alex Richard; WANG, Yee Tang; WIN, Khin Mar Kyi; ONG, Marcus Eng Hock; HSU, Li Yang. Time series analysis of demographic and temporal trends of tuberculosis in Singapore. *BMC Public Health*. 2014, roč. 14, č. 1, s. 1–10. Dostupné z DOI: 10.1186/1471-2458-14-1121.

24. KAREVAN, Zahra; SUYKENS, Johan A.K. Transductive LSTM for time-series prediction: An application to weather forecasting. *Neural Networks*. 2020, roč. 125, s. 1–9. ISSN 0893-6080. Dostupné z DOI: 10.1016/j.neunet.2019.12.030.

25. STELLMES, Marion; RÖDER, Achim; UDELHOVEN, Thomas; HILL, Joachim. Mapping syndromes of land change in Spain with remote sensing time series, demographic and climatic data. *Land Use Policy*. 2013, roč. 30, č. 1, s. 685–702. ISSN 0264-8377. Dostupné z DOI: 10.1016/j.landusepol.2012.05.007.

26. MITCHELL, Thomas M. *Machine Learning*. 1. vyd. USA: McGraw-Hill, Inc., 1997. ISBN 0070428077. Dostupné z DOI: 10.5555/541177.

27. HASTIE, Trevor; FRIEDMAN, Jerome; TIBSHIRANI, Robert. The Elements of Statistical Learning: Data Mining, Inference, and Prediction. In: Springer New York, 2001. ISBN 978-0-387-21606-5. Dostupné z DOI: 10.1007/978-0-387-21606-5.

28. KOTSIANTIS, SSotiris B.; ZAHARAKIS, Ioannis D.; PINTELAS, Panagiotis E. Machine Learning: A Review of Classification and Combining Techniques. *Artif. Intell. Rev.* 2006, roč. 26, č. 3, s. 159–190. ISSN 0269-2821. Dostupné z DOI: 10.1007/s10462-007-9052-3.

29. GAMBOA, John C. Borges. Deep Learning for Time-Series Analysis. *CoRR*. 2017, roč. abs/1701.01887. Dostupné z arXiv: 1701.01887.

30. BAGNALL, Anthony; LINES, Jason; BOSTROM, Aaron; LARGE, James; KEOGH, Eamonn. The great time series classification bake off: a review and experimental evaluation of recent algorithmic advances. *Data Mining and Knowledge Discovery*. 2017, roč. 31, č. 3, s. 606–660. ISSN 1573-756X. Dostupné z DOI: 10.1007/s10618-016-0483-9.

31. DEMPSTER, Angus; SCHMIDT, Daniel F.; WEBB, Geoffrey I. MiniRocket. In: *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. New York: ACM, 2021, s. 248–257. Dostupné z DOI: 10.1145/3447548.3467231.

32. DEMPSTER, Angus; PETITJEAN, François; WEBB, Geoffrey I. ROCKET: exceptionally fast and accurate time series classification using random convolutional kernels. *Data Mining and Knowledge Discovery*. 2020, roč. 34, č. 5, s. 1454–1495. Dostupné z DOI: 10.1007/s10618-020-00701-z.

33. MIDDLEHURST, Matthew; VICKERS, William; BAGNALL, Anthony. Scalable Dictionary Classifiers for Time Series Classification. In: *Intelligent Data Engineering and Automated Learning – IDEAL 2019*. Springer International Publishing, 2019, s. 11–19. Dostupné z DOI: 10.1007/978-3-030-33607-3_2.

34. LINES, Jason; TAYLOR, Sarah; BAGNALL, Anthony. HIVE-COTE: The Hierarchical Vote Collective of Transformation-Based Ensembles for Time Series Classification. In: *2016 IEEE 16th International Conference on Data Mining (ICDM)*. 2016, s. 1041–1046. ISSN 2374-8486. Dostupné z DOI: 10.1109/ICDM.2016.0133.

35. SHIFAZ, Ahmed; PELLETIER, Charlotte; PETITJEAN, François; WEBB, Geoffrey I. TS-CHIEF: a scalable and accurate forest algorithm for time series classification. *Data Mining and Knowledge Discovery*. 2020, roč. 34, č. 3, s. 742–775. Dostupné z DOI: 10.1007/s10618-020-00679-8.

36. DINGER, Timothy R.; CHANG, Yuan-chi; PAVULURI, Raju; SUBRAMANIAN, Shankar. *What is time series classification?* [online]. IBM, 2022 [cit. 2023-03-20]. Dostupné z: https://developer.ibm.com/learningpaths/get-started-time-series-classification-api/what-is-time-series-classification/.

37. BAGNALL, Anthony; LINES, Jason; BOSTROM, Aaron; LARGE, James; KEOGH, Eamonn. The great time series classification bake off: a review and experimental evaluation of recent algorithmic advances. *Data Mining and Knowledge Discovery*. 2017, roč. 31, č. 3, s. 606–660. ISSN 1573-756X. Dostupné z DOI: 10.1007/s10618-016-0483-9.

38. EBRAHIMI, Zahra; LONI, Mohammad; DANESHTALAB, Masoud; GHAREHBAGHI, Arash. A review on deep learning methods for ECG arrhythmia classification. *Expert Systems with Applications: X*. 2020, roč. 7, s. 100033. ISSN 2590-1885. Dostupné z DOI: 10.1016/j.eswax.2020.100033.

39. FAWAZ, Hassan Ismail; FORESTIER, Germain; WEBER, Jonathan; IDOUMGHAR, Lhassane; MULLER, Pierre-Alain. Deep learning for time series classification: a review. *Data Mining and Knowledge Discovery*. 2019, roč. 33, č. 4, s. 917–963. Dostupné z DOI: 10.1007/s10618-019-00619-1.

40. FIX, Evelyn; HODGES, Joseph Lawson. Nonparametric discrimination: consistency properties. *Randolph Field, Texas, Project 21-49-004*. 1951, s. 21–49. Dostupné také z: https://apps.dtic.mil/sti/citations/ADA800276. Accession Number ADA800276.

41. COVER, T.; HART, P. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*. 1967, roč. 13, č. 1, s. 21–27. Dostupné z DOI: 10.1109/TIT.1967.1053964.

42. WANG, Yuxiang; WANG, Ruijin; LI, Dongfen; ADU-GYAMFI, D.; TIAN, Kaibin; ZHU, Yixin. Improved Handwritten Digit Recognition using Quantum K-Nearest Neighbor Algorithm. *International Journal of Theoretical Physics*. 2019, roč. 58, s. 1–10. Dostupné z DOI: 10.1007/s10773-019-04124-5.

43. ADENIYI, Adedayo; WAI, Zune; YONGQUAN, Y. Automated Web Usage Data Mining and Recommendation System using K-Nearest Neighbor (KNN) Classification Method. *Applied Computing and Informatics*. 2014, roč. 36. Dostupné z DOI: 10.1016/j.aci.2014.10.001.

44. VAŠATA, D.; KOVALENKO, A.; TICHÝ, O. *BI-VZD* [online]. FIT ČVUT, 2023 [cit. 2023-04-14]. Dostupné z: https://courses.fit.cvut.cz/BI-VZD/lectures/files/05prednaska.pdf.

45. SAKOE, Hirokai; CHIBA, Seibi. Dynamic programming algorithm optimization for spoken word recognition. *IEEE Transactions on Acoustics, Speech, and Signal Processing*. 1978, roč. 26, č. 1, s. 43–49. Dostupné z DOI: 10.1109/TASSP.1978.1163055.

46. MÜLLER, Meinard. Dynamic Time Warping. In: *Information Retrieval for Music and Motion*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, s. 69–84. ISBN 978-3-540-74048-3. Dostupné z DOI: 10.1007/978-3-540-74048-3_4.

47. RATANAMAHATANA, Chotirat Ann; KEOGH, Eamonn. Three Myths about Dynamic Time Warping Data Mining. In: *Proceedings of the 2005 SIAM International Conference on Data Mining (SDM)*. [B.r.], s. 506–510. Dostupné z DOI: 10.1137/1.9781611972757.50.

48. ALIZADEH, Esmaeil. *An Introduction to Dynamic Time Warping* [online]. 2022. [cit. 2023-04-03]. Dostupné z: https://builtin.com/data-science/dynamic-time-warping.

49. ITAKURA, F. Minimum prediction residual principle applied to speech recognition. *IEEE Transactions on Acoustics, Speech, and Signal Processing*. 1975, roč. 23, č. 1, s. 67–72. Dostupné z DOI: 10.1109/TASSP.1975.1162641.

50. KEOGH, Eamonn; RATANAMAHATANA, Chotirat Ann. Exact indexing of dynamic time warping. *Knowledge and Information Systems*. 2005, roč. 7, č. 3. ISSN 0219-3116. Dostupné z DOI: 10.1007/s10115-004-0154-9.

51. YU, Fisher; KOLTUN, Vladlen. *Multi-Scale Context Aggregation by Dilated Convolutions*. 2016. Dostupné z DOI: 10.48550/arXiv.1511.07122.

52. DAU, Hoang Anh; BAGNALL, Anthony; KAMGAR, Kaveh; YEH, Chin-Chia Michael; ZHU, Yan; GHARGHABI, Shaghayegh; RATANAMAHATANA, Chotirat Ann; KEOGH, Eamonn. *The UCR Time Series Archive* [online]. 2019. [cit. 2023-04-10]. Dostupné z DOI: 10.48550/arXiv.1810.07758.

53. DEMŠAR, Janez. Statistical Comparisons of Classifiers over Multiple Data Sets. *J. Mach. Learn. Res.* 2006, roč. 7, s. 1–30. ISSN 1532-4435. Dostupné z DOI: 10.5555/1248547.1248548.

54. BENAVOLI, Alessio; CORANI, Giorgio; MANGILI, Francesca. *Should we really use post-hoc tests based on mean-ranks?* [online]. 2015. [cit. 2023-05-04]. Dostupné z DOI: 10.48550/arXiv.1505.02288.

55. BUNSE, Mirko. *CriticalDifferenceDiagrams.jl* [online]. 2022. [cit. 2023-05-04]. Dostupné z: https://mirkobunse.github.io/CriticalDifferenceDiagrams.jl/stable/.

56. NITIN, Verma. *Multiples of Golden-Ratio Modulo 1* [online]. 2021. [cit. 2023-04-21]. Dostupné z: https://mathsanew.com/articles/golden_ratio_multiples_mod_1.pdf.

57. KINGMA, Diederik P.; BA, Jimmy. *Adam: A Method for Stochastic Optimization.* 2017. Dostupné z DOI: 10.48550/arXiv.1412.6980.

58. VAN ROSSUM, Guido. *Python tutorial* [online]. Amsterdam, Netherlands, 1995-05 [cit. 2023-04-23]. Tech. zpr., CS-R9526. Centrum voor Wiskunde en Informatica (CWI). Dostupné z: https://ir.cwi.nl/pub/5007/05007D.pdf.

59. OSTROWSKA, Kamila. *A Brief History of Python* [online]. LearnPython.com, 2022 [cit. 2023-04-23]. Dostupné z: https://learnpython.com/blog/history-of-python/.

60. OVERFLOW, Stack. *Stack Overflow Developer Survey 2022* [online]. 2022. [cit. 2023-04-23]. Dostupné z: https://survey.stackoverflow.co/2022/#most-popular-technologies-language.

61. HULS, Mike. *Why Python is so slow and how to speed it up* [online]. towardsdatascience.com, 2021 [cit. 2023-04-23]. Dostupné z: https://towardsdatascience.com/why-is-python-so-slow-and-how-to-speed-it-up-485b5a84154e.

62. TEAM, Benchmarks Game. *The Computer Language Benchmarks Game - box plot charts* [online]. 2023. [cit. 2023-04-23]. Dostupné z: https://benchmarksgame-team.pages.debian.net/benchmarksgame/box-plot-summary-charts.html.

63. ZHANG, Qiang; XU, Lei; XU, Baowen. RegCPython: A Register-Based Python Interpreter for Better Performance. *ACM Trans. Archit. Code Optim.* 2022, roč. 20, č. 1. ISSN 1544-3566. Dostupné z DOI: 10.1145/3568973.

64. FOUNDATION, Python Software. *Extending Python with C or C++* [online]. 2023. [cit. 2023-04-23]. Dostupné z: https://docs.python.org/3/extending/extending.html.

65. SHAJII, Ariya; RAMIREZ, Gabriel; SMAJLOVIĆ, Haris; RAY, Jessica; BERGER, Bonnie; AMARASINGHE, Saman; NUMANAGIĆ, Ibrahim. Codon: A Compiler for High-Performance Pythonic Applications and DSLs. In: Montréal, QC, Canada: Association for Computing Machinery, 2023, s. 191–202. CC 2023. ISBN 9798400700880. Dostupné z DOI: 10.1145/3578360.3580275.

66. RIGO, Armin et al. *PyPy - A fast, compliant alternative implementation of Python* [online]. 2023. [cit. 2023-04-24]. Dostupné z: https://www.pypy.org/features.html.

67. BEHNEL, Stefan; BRADSHAW, Robert; CITRO, Craig; DALCIN, Lisandro; SELJEBOTN, Dag Sverre; SMITH, Kurt. Cython: The Best of Both Worlds. *Computing in Science & Engineering*. 2011, roč. 13, č. 2, s. 31–39. Dostupné z DOI: 10.1109/MCSE.2010.118.

68. BEZANSON, Jeff; EDELMAN, Alan; KARPINSKI, Stefan; SHAH, Viral B. et al. *Julia Micro-Benchmarks* [online]. 2019. [cit. 2023-05-05]. Dostupné z: https://julialang.org/benchmarks/.

69. KARPINSKI, Stefan. *The Unreasonable Effectiveness of Multiple Dispatch*. Sv. 2019 [online]. 2019. [cit. 2023-05-06]. Dostupné z: https://www.youtube.com/watch?v=kc9HwsxE1OY&t=424s.

70. BLAOM, Anthony D.; KIRALY, Franz; LIENART, Thibaut; SIMILLIDES, Yiannis; ARENAS, Diego; VOLLMER, Sebastian J. MLJ: A Julia package for composable machine learning. *Journal of Open Source Software*. 2020, roč. 5, č. 55, s. 2704. Dostupné z DOI: 10.21105/joss.02704.

71. BLAOM, Anthony D.; VOLLMER, Sebastian J. *Flexible model composition in machine learning and its implementation in MLJ*. 2020. Dostupné z DOI: 10.48550/arXiv.2012.15505.

72. PEDREGOSA, F.; VAROQUAUX, G.; GRAMFORT, A.; MICHEL, V.; THIRION, B.; GRISEL, O.; BLONDEL, M.; PRETTENHOFER, P.; WEISS, R.; DUBOURG, V.; VANDERPLAS, J.; PASSOS, A.; COURNAPEAU, D.; BRUCHER, M.; PERROT, M.; DUCHESNAY, E. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*. 2011, roč. 12, s. 2825–2830.

73. BLAOM, Anthony D.; LIENART, Thibaut; SAMUEL, Okon; SIMILLIDES, Yiannis; ALUTHGE, Dilum et al. *JuliaAI/MLJModels.jl* [online]. 2023. [cit. 2023-05-06]. Dostupné z: https://github.com/JuliaAI/MLJModels.jl.

74. BLAOM, Anthony D. et al. *MLJ Reference Manual - Adding Models for General Use* [online]. 2023. [cit. 2023-05-06]. Dostupné z: https://alan-turing-institute.github.io/MLJ.jl/dev/adding_models_for_general_use/.

75. STALLINGS, William. *Computer Organization and Architecture: Designing for Performance*. 10th. USA: Pearson Education, 2016. ISBN 9780134101613.

76. BEZANSON, Jeff; EDELMAN, Alan; KARPINSKI, Stefan; SHAH, Viral B. et al. *Julia Documentation* [online]. 2023. [cit. 2023-04-24]. Dostupné z: https://docs.julialang.org/en/v1/.

77. ELROD, Chris; HOLY, Tim; ALUTHGE, Dilum; RANOCHA, Hendrik; PROTTER, Mason; ELROD, Chris et al. *JuliaSIMD/LoopVectorization.jl* [online]. 2023. [cit. 2023-05-10]. Dostupné z: https://github.com/JuliaSIMD/LoopVectorization.jl.

78. KELLER, C. Brenhin; ELROD, Chris et al. *JuliaSIMD/VectorizedStatistics.jl* [online]. 2023. [cit. 2023-05-10]. Dostupné z: https://github.com/JuliaSIMD/VectorizedStatistics.jl.

79. PLAS, Fons van der; DRAL, Michiel; BERG, Paul; ΓΕΩΡΓΑΚΌΠΟΥΛΟΣ, Παναγιώτης; HUIJZER, Rik; BOCHEŃSKI, Mikołaj; MENGALI, Alberto; LUNGWITZ, Benjamin; BURNS, Connor; PRIYASHAN, Hirumal; LING, Jerry; ZHANG, Eric; SCHNEIDER, Felipe S. S.; WEAVER, Ian; LUO, Xiu-zhe (Roger); KADOWAKI, Shuhei; WU, Gabriel; GERRITSEN, Jelmar; NOVOSEL, Rok; SUPANAT; MOON, Zachary; MÜLLER, Luis; TECOSAUR; ABBOTT, Michael; BAUER, Nicholas; PBOUFFARD; TERASAKI, Satoshi; POLASA, Shashank. *fonsp/Pluto.jl: v0.19.25*. Zenodo, 2023. Ver. v0.19.25. Dostupné z DOI: 10.5281/zenodo.7841565.

Obsah přiloženého média

- `benchmarks/` ..složka obsahující nástroje pro benchmarkování a generování grafů
- `examples/` .....složka obsahující ukázky použití
- `test/` .....složka obsahující unit testy
- `src/` .....složka se zdrojovými kódy balíčku TsClassification
  - `DataSets` .....složka obsahující modul obstarávající práci s datasety
  - `KNNDTW` .....složka s modeulem s KNN + DTW
  - `MiniRocket` .....složka s modeulem s MINIROCKET
  - `TsClassification.jl` .....entrypoint projektu
- `src/` .....složka se zdrojovými kódy balíčku TsClassification