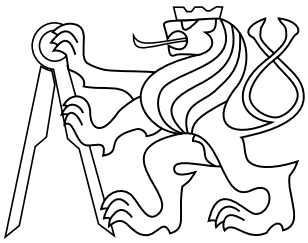
bakalářská práce

# Predikce selhání pro Model-Based integrační testování

*Konstantin Khokhlov*



Březen 2020

Ing. Lukáš Krejčí

České vysoké učení technické v Praze
Fakulta elektrotechnická, Katedra měření

# ZADÁNÍ BAKALÁŘSKÉ PRÁCE

## I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Khokhlov** Jméno: **Konstantin** Osobní číslo: **474681**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra měření**

Studijní program: **Kybernetika a robotika**

## II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

**Predikce selhání pro Model-Based integrační testování**

Název bakalářské práce anglicky:

**Failure prediction for Model-Based Integration Testing**

Pokyny pro vypracování:

1. Seznamte se s problematikou HiL integračního testování.
2. Analyzujte zdrojový kód a modelovací jazyk MBT nástroje Taster. Navrhněte rozšíření modelovacího jazyka o data použitelná pro predikci závad ve funkcionalitách testovaných systémů.
3. Seznamte se s frameworkem ML.NET pro strojové učení.
4. Navrhněte vhodnou architekturu pro začlenění predikce selhání do nástroje Taster s využitím frameworku ML.NET. Navržená architektura by měla být modulární a měla by umožnit snadnou výměnu či konfiguraci jednotlivých součástí, jako například použité algoritmy strojového učení, či modely. Navrženou architekturu implementujte do zdrojového kódu nástroje Taster.
5. Navrhněte a implementujte konkrétní metodu predikce selhání s využitím této architektury. Metodu experimentálně ověřte na případové studii.

Seznam doporučené literatury:

[1] Bishop, Christopher M.: Pattern Recognition and Machine Learning (Information Science and Statistics). Springer-Verlag, 2006.
[2] Zander, J., Schieferdecker, I. and Mosterman, P. J.: Model-Based Testing for Embedded. Systems. Taylor & Francis, 2011.
[3] Johan Bengtsson, and Wang Yi: Timed Automata: Semantics, Algorithms and Tools. Lecture Notes in Computer Science. 2004.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

**Ing. Lukáš Krejčí, katedra měření FEL**

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

Datum zadání bakalářské práce: **12.02.2020** Termín odevzdání bakalářské práce: **22.05.2020**

Platnost zadání bakalářské práce:
**do konce letního semestru 2020/2021**

| Ing. Lukáš Krejčí<br>podpis vedoucí(ho) práce | podpis vedoucí(ho) ústavu/katedry | prof. Mgr. Petr Páta, Ph.D.<br>podpis děkana(ky) |
|---|---|---|

 © ČVUT v Praze, Design: ČVUT v Praze, VIC

## III. PŘEVZETÍ ZADÁNÍ

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

. _____
Datum převzetí zadání

_____
Podpis studenta

 © ČVUT v Praze, Design: ČVUT v Praze, VIC

## Poděkování

Chtěl bych poděkovat vedoucímu své diplomové práce Ing. Lukášu Krejčímu za odborné vedení, za pomoc a rady při zpracování této práce.

## 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.

## Abstrakt

Cílem této bakalářské práce je rozšíření funkcionality nástroje Taster o možnost predikce selhání jednotlivých funkcionalit modelů a výpočtu jejich relevance pro účely Model-Based integračního testováni. Práce po úvodním seznámením se s problematikou integračního testováni nejprve zkoumá časované automaty, nástroje UPPAAL a Taster, a knihovnu ML.NET, dále analyzuje data a algoritmy strojového učení, poté se zaměřuje na návrh a implementaci konkretní metody predikce a následné validaci její věrohodnosti pomocí generovaných dat. Navržená architektura je modulární a umožňuje snadnou výměnu či konfiguraci jednotlivých součástí.



### Klíčová slova

HIL, Taster, ML.NET, Strojové učeni, UPPAAL

## Abstrakt

The main goal of this bachelor thesis is an extension of the functionality of the Taster tool by failure prediction and calculating relevances of individual functions in test models for Model-Based Integration Testing. The thesis first studies timed automata, UPPAAL, and Taster tools, as well as the framework ML.NET. Then analyzes data and Machine Learning algorithms. Thereafter focuses on the design and implementation of a specific prediction method and following validation of its effectiveness with generated data. Designed architecture is modular and allows easy replacement or configuration of individual components.



### Keywords

HIL, Taster, ML.NET, Machine Learning, UPPAAL

# Obsah

**1. Úvod** **1**

**2. Použité nástroje a software** **2**
2.1. UPPAAL a časované automaty . . . . . 2
2.2. Taster . . . . . 2
2.2.1. Architektura nástroje Taster . . . . . 3
Parser XML . . . . . 3
Algoritmus simulace modelu . . . . . 3
Trace Logger . . . . . 4
Testovací Adaptéry . . . . . 4
Uživatelské rozhraní . . . . . 4
2.3. Framework ML.NET . . . . . 5

**3. Teoretická část** **7**
3.1. Vstupní data . . . . . 7
3.2. Algoritmy pro predikci . . . . . 9
3.2.1. Lineární modely . . . . . 9
3.2.2. Rozhodovací stromy . . . . . 10

**4. Implementace** **12**
4.1. Rozbor XML . . . . . 12
4.2. Třída Relevance . . . . . 13
4.3. Vytvoření prediktorů . . . . . 15
4.4. Uživatelské rozhraní . . . . . 18

**5. Validace** **20**
5.1. Generování dat . . . . . 20
5.2. Ověření modelu . . . . . 20
5.3. Křížová validace . . . . . 21

**6. Závěr** **23**

**Přílohy**

**A. Generování dat** **24**

**Literatura** **25**

**Literatura** **25**

# Zkratky

| | |
|---|---|
| MBT | Model-Based Testing |
| HIL | Hardware In the Loop |
| LightGBM | Light Gradient Boosted Machine |
| GAM | Generalized Additive Model |
| ML | Machine Learning |
| SDCA | Stochastic Dual Coordinated Ascent |
| SSGD | Symbolic Stochastic Gradient Descent |
| RSS | Residual Sum of Squares |
| RMS | Root Mean Square |

# 1. Úvod

V současné době v automobilovém průmyslu prakticky nejde zaručit, aby každá součástka byla vyrobená jednou firmou. Integrování několika komponentů od různých výrobců do jednoho distribuovaného sytému může způsobit chyby, jež je potřeba otestovat. K odhalení těchto chyb slouží proces takzvaného integračního testování. Pro tyto účely je velice aktuální oblast tzv. "Model-Based" Testování (MBT). Pojem MBT označuje metodu testování na základě spustitelných modelů. Základním účelem MBT je automatické generování testů, nikoliv jejich ruční vytvoření.[1]

Pod vedením pana Ing. Jana Sobotky Ph.D. na Katedře měření Fakulty elektrotechnické ČVUT v Praze byl vyvinut nástroj Taster, jenž využívá metodu MBT a slouží k integračnímu testování automobilových řídících jednotek. Taster pracuje s modely, které jsou vytvořené na základě časovaných automatů a jsou kompatibilní s formátem softwarového prostředí UPPAAL, jež slouží k modelování a verifikaci systémů reálného času. Nástroj umožňuje propojení s reálným hardwarem pro účely tzv. "Harware-in-the-Loop"(HIL) testování. Základní myšlenkou HIL je začlenění reálného hardwaru do simulace a testování řídících algoritmů na reálných částech systému. [2]

Cílem této bakalářské práce je rozšíření funkcionality nástroje Taster o možnost predikce selhání jednotlivých komponentů testovaného systému. Tato predikce pomůže stanovit spolehlivost funkcionalit řídících jednotek a následně určit jejich hodnoty relevance. Tyto hodnoty se dále použijí v následujících algoritmech pro testování.

Predikce bude realizována na základě algoritmů strojového učení s využitím frameworku ML.NET pro programovací jazyk C#. Modely algoritmů se naučí na datech sebraných za běhu testů, která popisují charakter jednotlivých funkcionalit systému. V době napsání této bakalářské práce reálná data nejsou k dispozici, proto je funkčnost použitých algoritmů následně ověřena na generovaných datech.

Navržená architektura by měla být modulární a měla by umožnit snadnou výměnu či konfiguraci jednotlivých součástí, jako například použité algoritmy strojového učení, či modely predikce. Rozšíření modelovacího jazyka by také mělo zachovat kompatibilitu s formátem modelů nástroje UPPAAL.

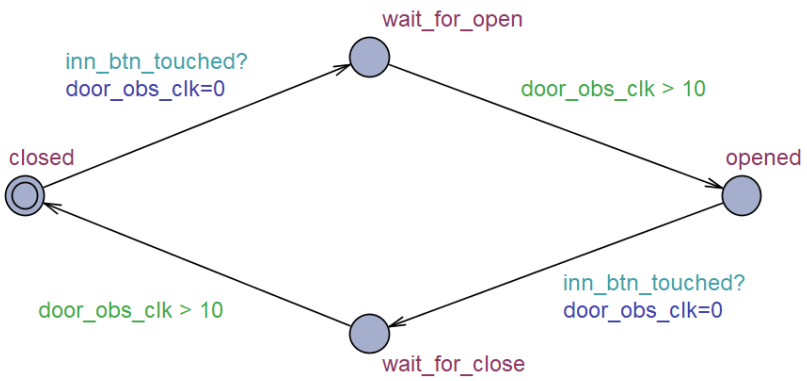
# 2. Použité nástroje a software

## 2.1. UPPAAL a časované automaty

UPPAAL je softwarové prostředí pro modelování a verifikaci systémů reálného času modelovaných jako sítě časovaných automatů, rozšířené o strukturované datové typy. Tento nástroj byl vyvinut spoluprací dvou univerzit Upsala a Alborgh. [3]

Časovaný automat je konečný automat rozšířený o časové omezení. Takový automat může být považován za abstraktní model systému reálného času. Každý automat lze reprezentovat grafem, jenž je určen množinou uzlů (reprezentující jednotlivé stavy automatu) a množinou orientovaných hran (jež reprezentují možné přechody), které spojují tyto uzly.[4] Proměnné automatu modelují závislosti v systému a logické hodiny v systému, které jsou inicializovány nulou při spuštění systému a poté se synchronně zvyšují se stejnou rychlostí. K omezení chování automatu slouží tzv. chrániče (angl. *guard*) a *invarianty*. Přechod mezi stavy může být proveden, pokud aktuální hodnoty proměnných vyhovují související podmínce *guardu* na příslušné hraně a také jsou splněny podmínky všech *invariantů* příslušného uzlu .[5]

Jednotlivé modely mohou obsahovat jeden nebo více časovaných automatů. Modely obsahující několik automatů jsou doplněny o *synchronizaci*. Synchronizace je mechanismus umožňující všem automatům přejít z jednoho stavu do druhého ve stejný čas.[6]




**Obrázek 1.** Příklad časovaného automatu v UPPAALu

Nástroj UPPAAL umožňuje provádět simulační kroky k ověření předpokládaného chování. Výsledné modely včetně deklarací jsou uloženy v *.xml souboru, který je později načten nástrojem Taster.[6]

## 2.2. Taster

Taster je testovací nástroj umožňující generování, provádění a vyhodnocování integračních testů z modelů v jazyce časovaných automatů.[6] Taster pracuje s modely kompa-

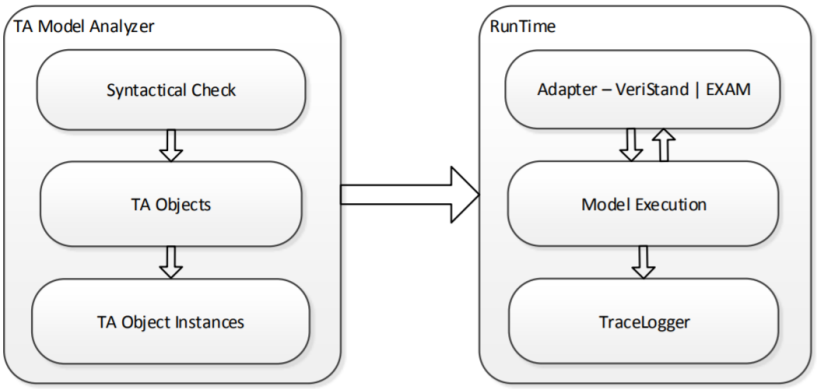
tibilními s formátem modelů pro program UPPAAL a umožňuje navázat komunikaci s reálným hardwarem pro účely "Hardware In Loop"(HIL) testování.

Nástroj poskytuje následující funkce:

- Ověření syntaxe modelů
- Prohlížení modelů v jazyce časovaných automatů
- On-line MBT testování
- Výběr algoritmu průchodu modelem
- Kompletní zaznamenáváni testů s vizualizací pro snadné ladění
- Vyhodnocování testů a uložení statistiky

### 2.2.1. Architektura nástroje Taster

Architekturu nástroje Taster lze rozdělit na 2 základní části, jež jsou vidět na obrázku 2. První část se zabývá načtením modelu z *.xml souboru a jeho následným převodem na vhodné datové struktury. Druhá část slouží k samotnému testování systému. [7] První část je implementována ve třídě *TASystem*. Druhé části odpovídá třída *TARuntime*.




**Obrázek 2.** Architektura Tasteru [7]

#### Parser XML

Taster načítá modely, které byly předem připravené v UPPAALu. Po načtení se nejdříve kontroluje syntaxe modelu. Potom Taster prochází jednotlivé XML elementy a postupně inicializuje instance třídy *Template*, kam se ukládají hrany a uzly spolu s jejich atributy (např. *guardy* a *invarianty*).

#### Algoritmus simulace modelu

Taster nabízí různé strategie průchodu modelem: náhodné, systematické nebo s použitím algoritmu strojového učení. Tyto strategie jsou neustále rozvíjeny. Některé z nich při rozhodování jakou hranou projít z aktuálního stavu používají hodnoty relevance uložené na příslušných hranách. Tato práce je zaměřena na výpočet těchto hodnot relevance.

### Trace Logger

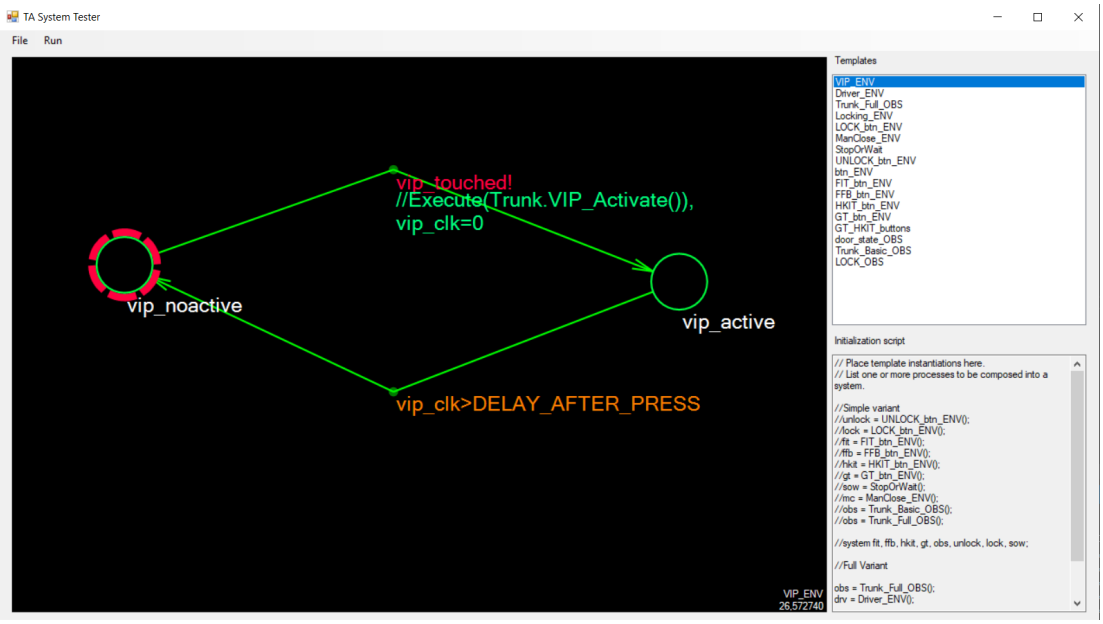
Pro účely analýzy, každý krok průchodu grafu je zaznamenáván v instanci třídy *TraceLogger*, která může být uložena do *.xml souboru. Následně lze nahrát uložené statistiky a znovu projít grafem s vizualizací postupu testu.

### Testovací Adaptéry

K účelům HIL testování slouží nástroje *EXAM* a *VeriStand*, jež umožňují komunikaci s reálným hardwarem. Ke komunikaci s těmito nástroji a provádění integračních testů v Tasteru slouží následující adaptéry. *EXAMAdaptor* přidává podporu pro nástroj *EXAM*. *VeriStandAdaptor* je určen pro podporu nástroje *VeriStand*. [4]

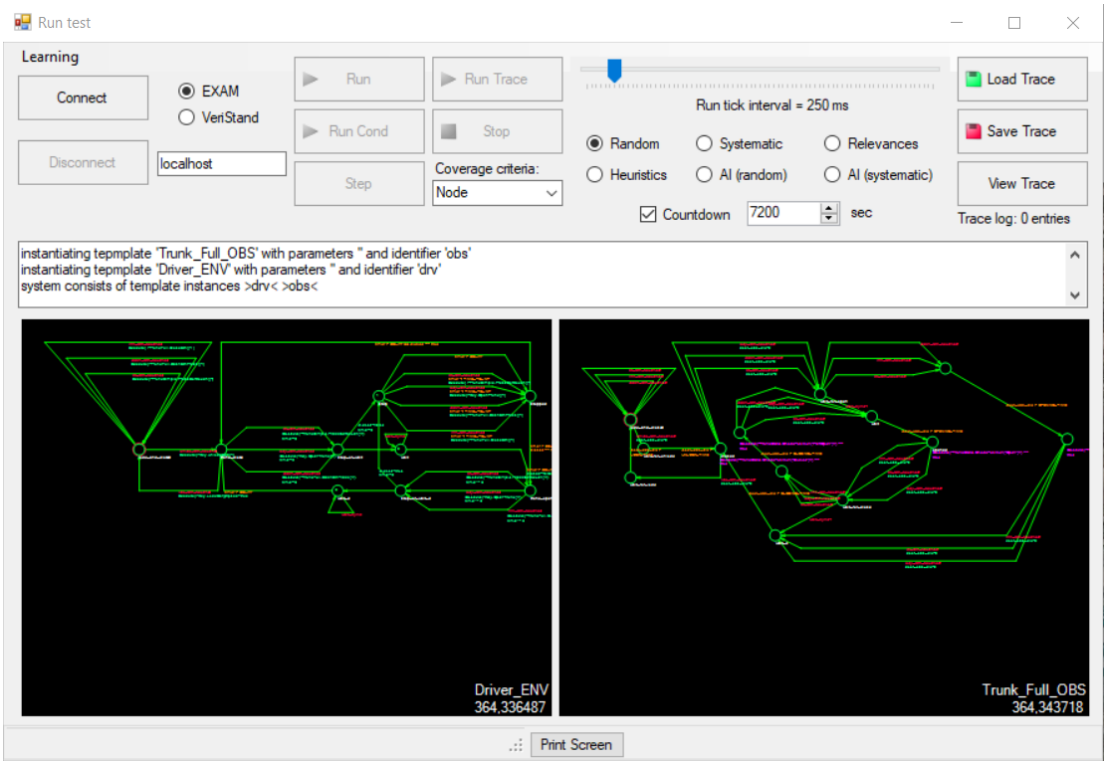
### Uživatelské rozhraní

Interakce uživatele s Tasterem je implementována ve formě grafického uživatelského rozhraní. Po spuštění programu se objeví dialogové okno *TA System*, kde lze nahrát a prohlédnout model a následně otevřít další dialogové okno *Run Test*.




**Obrázek 3.** Uživatelské rozhraní Tasteru. Dialogové okno *TA System*.

V okně *Run Test* lze provádět samotné testování, navazovat komunikaci s hardwarem, vybírat algoritmus průchodu, ukládat a nahrávat simulační kroky. Spolu s *Run Test* se otevře okno *LearningForm* pro výběr modelu strojového učení, který se použije v algoritmech průchodu modelem.




**Obrázek 4.** Uživatelské rozhraní Tasteru. Dialogové okno *Run Test*.

## 2.3. Framework ML.NET

Protože celá aplikace Taster je vyvinuta v programovacím jazyce C# pro realizaci algoritmů strojového učení lze použit framework ML.NET. Jedná se o populárni open-source knihovnu pro strojové učení od společnosti Microsoft obsahující řadu algoritmů a nástrojů pro řešení problémů klasifikace a regrese.

Hlavním instrumentem ML.NET je třída *MLContext*, jež nabízí sadu nástrojů pro transformaci a modifikaci vstupních dat, umožňuje naučení modelů pro predikci pomocí efektivních algoritmů a následnou validaci těchto modelů.

Proces sestavení modelu pro predikci (regresi) pomoci frameworku ML.NET je následující:

1 **Připravení dat a datových tříd:**

Data musí byt uložené v *.csv nebo *.txt souboru ve tvaru několika sloupců. Jeden sloupec musí odpovídat predikované hodnotě, ostatní musí odpovídat vstupním příznakům na jejichž základě bude model provádět predikce. Datová třída musí mít jednotlivé proměnné pro každý sloupec. Také se musí vytvořit třída s proměnnou pro uložení predikce.

2 **Načtení a transformace dat:**

ML.NET používá třídu *IDataView* jako flexibilní a efektivní způsob popisu číselných nebo textových tabulkových dat. Data se mohou načíst do instance této třídy pomocí metody *MLContext.Data.LoadFromTextFile()*. Algoritmy učení vyžadují jeden sloupec příznaků, proto je potřeba sloučit všechny příznakové sloupce do jednoho. Toto lze provést pomoci metody *MLContext.Transforms.Concatenate()*. Pomocí různých metod *MLContext.Transforms* (například *NormalizeMinMax()*) lze také provést

normalizaci jednotlivých sloupců. Všechny transformace se zapisují do jednoho transformačního řetěze.

3 **Výběr algoritmu a naučení modelu:**

Pro následné naučení modelu je potřeba vybrat nutný algoritmus. Třída *MLContext.Regression.Trainers* nabízí sadu lineárních algoritmů (např. Averaged perceptron, Stochastic dual coordinated ascent), algoritmy na základě rozhodovacích stromů (např. Fast tree, Light gradient boosted machine), meta algoritmy (např. One versus all) a jiné [8]. Vybraný algoritmus (tzv. *trainer*) se také zapíše do transformačního řetěze. Dále se vyvolá metoda *Fit()*, která naučí model na základě transformovaných dat.

4 **Validace modelu:**

Účinnost naučeného modelu lze ověřit pomocí testovacích dat. Pomocí metody *Transform()* lze vytvořit predikce pro vstupní příznaky testovacích dat. Metoda *MLContext.Regression.Evaluate()* umožňuje porovnání predikce s reálnými hodnotami testovacích dat. Výstupem metody jsou metriky vyhodnocující přesnost modelu. Tyto metriky jsou popsané v kapitole 5.2.

5 **Použití modelu:**

Pro následné použití modelu pro predikci je potřeba vytvořit takzvaný zdroj predikce (angl. *prediction engine*), jenž umožňuje provádět predikci pro jednotlivé vzorky dat. Zdroj predikce vytvoří metoda *MLContext.Model.CreatePredictionEngine()*.

# 3. Teoretická část

## 3.1. Vstupní data

Jak bylo zmíněno v úvodní části, cílem této práce je výpočet hodnot relevance jednotných funkcionalit v systému. Tato hodnota má rozdělit funkce dle jejich důležitosti, tedy funkcionalita, která má větší hodnotu relevance, má větší prioritu během testování, a algoritmus procházející stavovým automatem může vybrat hranu odpovídající této funkcionalitě jako první. Na určení relevance se podílejí tři hlavní parametry:

- **Index bezpečnosti** - je index vyjadřující vliv dané funkcionality na bezpečnost pasažérů automobilu. Rozsah indexu bezpečnosti je od 1 do 10.
- **Index komfortu** - tento parametr říká o tom, jako moc daná funkce ovlivňuje komfort pasažérů. Rozsah tohoto indexu je také od 1 do 10.
- **Koeficient nespolehlivosti** - tento koeficient zahrnuje informaci o tom jak často selhává činnost dané funkce.

Indexy bezpečnosti a komfortu byli určené manuálně a jsou k dispozici pro každou funkci. Tato práce je zaměřena na výpočet koeficientu nespolehlivosti. Tento koeficient nabývá hodnot od 0 do 1, a odpovídá normalizované pravděpodobnosti selhání funkce, tedy čím větší je koeficient, tím je nejspolehlivější daná funkcionalita, což vede k větší hodnotě relevance.

Pro výpočet koeficientu nespolehlivosti je potřeba umět predikovat pravděpodobnost selhání funkcionality na základě parametrů popisujících danou funkci. Tyto parametry jsou následující:

- Počet řídících jednotek podílejících se na vykonání dané funkce.
  Zkratka **CU_num** (angl. *Control Unit number*).
- Seznam identifikátorů těchto řídících jednotek.
  Zkratka **CU_IDs** (angl. *Control Unit IDs*)
- Počet dodavatelů podílejících na vyrábění těchto jednotek.
  Zkratka **S_num** (angl. *Suppliers number*)
- Seznam identifikátorů dodavatelů.
  Zkratka **S_IDs** (angl. *Suppliers IDs*)
- Identifikátor funkce.
  Zkratka **F_ID** (angl. *Function ID*)

Těmto parametrům se v následujícím textu bude říkat příznakový vektor.

Pro predikci selhání je potřeba naučit model strojového učení na základě sady dat. Každý řádek dat se musí skládat z příznakového vektoru a jemu přiřazené pravděpodobnosti selhání. Pokud se hodnoty pravděpodobnosti selhání normalizují, tedy se převedou na interval od 0 do 1, lze je považovat za hodnoty koeficientu nespolehlivosti.

Data musí být nejprve sebrána během testů, kde algoritmus prochází časovaným automatem a pokaždé když projde hranou (tedy se vykoná funkce), uloží se parametry popisující danou funkci a její aktuální stav; pokud dojde ke selhání zapíše se jednička, v opačném případě nula. Tím pádem každý řádek sebraných dat bude odpovídat příznakovému vektoru a přiřazenou nulou nebo jedničkou. Dále je potřeba vypočítat hash hodnoty z identifikátorů pomocí hašovací funkce. Příklad sebraných dat po výpočtu hash hodnot funkce je vidět v tabulce 1.

| CU_num | CU_IDs | S_num | S_IDs | F_ID | Failure |
|---|---|---|---|---|---|
| 2 | -1506833821 | 1 | 1074163616 | -1285657056 | 0 |
| 1 | 1636479509 | 1 | 1948168070 | -1819992976 | 0 |
| 4 | -552859147 | 4 | 2134790324 | -1433026027 | 1 |
| 4 | -552859147 | 4 | 2134790324 | -1433026027 | 0 |
| 1 | 1636479509 | 1 | 1948168070 | -1819992976 | 0 |
| 2 | 1799464071 | 2 | -1624288750 | -1448783338 | 0 |
| 2 | -1506833821 | 1 | 1074163616 | -1285657056 | 0 |
| 2 | 1799464071 | 2 | -1624288750 | -1448783338 | 1 |
| ... | | | | | ... |

**Tabulka 1.** Příklad sebraných dat

Použití sloupců se stavem funkce (0 nebo 1) k naučení modelů vede ke klasifikaci chybného a nechybného stavu. Proto je potřeba transformovat data pro řešení problému regrese. Z celé sady dat najdeme všechny unikátní příznakové vektory. Potom pro každý unikátní vektor vypočítáme střední hodnotu dle vztahu:

$$p_i = \frac{\sum_j^n f_{i,j}}{n}, \tag{1}$$

kde $n$ je počet výskytu $i$-tého příznakového vektoru v celé sadě dat, $f_{i,j} \in \{0, 1\}$ je stav $i$-tého vektoru v $j$-tém řádku. Pro účely této práce lze považovat danou hodnotu za skutečnou pravděpodobnost selhání pro $i$-tý unikátní vektor, pokud je celá datová sada dostatečně velká, tedy výskyty chybného stavu jsou statisticky relevantní a blízké k reálné distribuci.

Po této transformaci se vytvoří tzv. koncentrát z dat obsahující jen unikátní příznakové vektory a přiřazené hodnoty pravděpodobnosti selhání.

| CU_num | CU_IDs | S_num | S_IDs | F_ID | Failure_prob |
|---|---|---|---|---|---|
| 2 | -1506833821 | 1 | 1074163616 | -1285657056 | 0.0120 |
| 1 | 1636479509 | 1 | 1948168070 | -1819992976 | 0.0054 |
| 2 | 1799464071 | 2 | -1624288750 | -1448783338 | 0.0272 |
| 4 | -552859147 | 4 | 2134790324 | -1433026027 | 0.0451 |
| ... | | | | | ... |

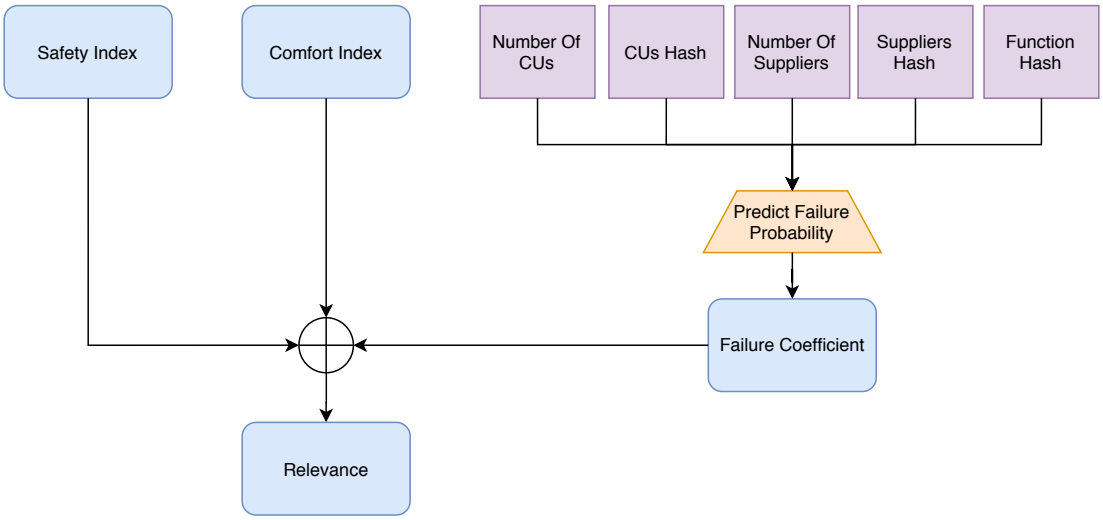
**Tabulka 2.** Příklad transformovaných dat - koncentrátu

Následně je potřeba normalizovat sloupec s pravděpodobnostmi selhání. To lze provést na základě maximální a minimální hodnoty. Tím pádem je ke každému příznakovému vektoru přiřazen koeficient nespolehlivosti, který bude následně predikován. Výtah několika řádků z výsledného koncentrátu je v tabulce 3.

| CU_num | CU_IDs | S_num | S_IDs | F_ID | Failure_coef |
|---|---|---|---|---|---|
| 2 | -1506833821 | 1 | 1074163616 | -1285657056 | 0.16 |
| 1 | 1636479509 | 1 | 1948168070 | -1819992976 | 0.07 |
| 2 | 1799464071 | 2 | -1624288750 | -1448783338 | 0.38 |
| 4 | -552859147 | 4 | 2134790324 | -1433026027 | 0.67 |
| ... | | | | | ... |

**Tabulka 3.** Příklad koncentrátu po normalizaci

Schéma celého vypočtu hodnoty relevance je ukázána na obrázku 5.




**Obrázek 5.** Schéma výpočtu hodnoty Relevance

## 3.2. Algoritmy pro predikci

V teto podkapitole jsou popsané algoritmy a modely, které lze využít pro predikci pravděpodobnosti selhání.

### 3.2.1. Lineární modely

Nejednodušším lineárním modelem pro regresi je lineární kombinace vstupních proměnných:

$$\hat{y}(\mathbf{x}, \mathbf{w}) = w_0 + w_1 x_1 + ... + w_D x_D, \tag{2}$$

kde $\mathbf{x} = (x_1, ..., x_D)^T$ je vektor vstupních proměnných dimenze $D$. Toto bývá také označováno jako lineární regrese. Klíčovou vlastností tohoto modelu je, že se jedná o lineární funkci s parametry $w_0, ..., w_D$. Pro rozšíření této třídy modelů lze zvažovat lineární kombinaci nelineárních funkcí:

$$\hat{y}(\mathbf{x}, \mathbf{w}) = w_0 + \sum_{j}^{D-1} w_j \phi_j(\mathbf{x}), \tag{3}$$

kde $\boldsymbol{\phi}(\mathbf{x})$ je takzvaná bázová funkce. [9] Například pro polynomiální regresi s jednou vstupní proměnnou je tato funkce definována jako:

$$\phi_j(x) = x^j. \tag{4}$$

Princip naučení modelu na vstupních datech je založen na minimalizaci účelové funkce. Hledají se takové parametry $\omega_0, ..., \omega_D$, aby výstupní hodnoty funkce (3) pro všechny vstupní příznakové vektory co nejvíce odpovídali hodnotám přiřazeným k vektorům v datové sadě. Účelová funkce může být definována, například, jako suma kvadratických odchylek predikovaných hodnot od reálných dělená počtem vzorků:

$$L(\mathbf{y}, \hat{y}) = \frac{1}{N} \sum_{i}^{N} (y_i - \hat{y}(\mathbf{x}_i, \mathbf{w}))^2, \tag{5}$$

kde $N$ je počet příznakových vektorů v datové sadě, $y_i$ je reálná hodnota přiřazena $i$-tému příznakovému vektoru $x_i$. ML.NET nabízí několik lineárních algoritmů, například:

- **Stochastic dual coordinated ascent (SDCA)** je moderní optimalizační algoritmus pro konvexní cílové funkce. Mezi vlastnosti metody SDCA patří schopnost provádět proudové učení (bez vložení celé sady dat do paměti) a schopnost dosáhnout rozumného výsledku s malým množstvím dat. [8]
- **Symbolic stochastic gradient descent (SSGD)** je algoritmus, jenž provádí predikce na základě rozdělení příznakového prostoru. Jedná se o jeden z nejrychlejších a zároveň nepřesnějších lineárních algoritmů. Velká výpočetní rychlost je stanovená tím, že SSGD učí modely paralelně v samostatných vláknech.[8]

Lineární modely fungují dobře pro lineární oddělitelné (angl. *linear separable)* data bez kategorických parametrů. Použití identifikátoru ve tvaru hash hodnoty v příznakových vektorech může vést k tomu, že nebude možné najít potřebné parametry $\omega_0, ..., \omega_D$. Podobné problémy se tedy lépe řeší pomocí rozhodovacích stromů. [10]

### 3.2.2. Rozhodovací stromy

Rozhodovací strom je modelem, který je jednoduchý na interpretaci a zároveň je obecně přesný a efektivní. Mezi výhody rozhodovacích stromů patří také rychlost predikce a nezávislost na normalizaci vstupních dat, protože každý jednotlivý parametr je použit zvlášť v rozhodovacím procesu. [11] Rozhodovací strom je možné použít pro řešení problémů klasifikace a regrese, z čehož vznikají pojmy klasifikačních a regresních stromů.

Existuje několik různých algoritmů pro vytvoření rozhodovacího stromu, například *CART*, *ID3*, *C5.0* a jiné.

Obecně regresní strom rozděluje příznakový prostor, t.j. prostor všech možných hodnot příznaků, na $M$ disjunktních oblastí $R_1, ..., R_M$. Každému pozorovaní, které spadne do některé oblasti $R_j$, je přiřazena hodnota rovná průměru hodnot získaných z trénovacích dat pro danou oblast $R_j$. Určení hranic těchto oblastí je založeno na minimalizaci regresní chyby, například sumy RSS (angl. *residual sum of squares*) definované vztahem

$$RSS = \sum_{j=1}^{M} \sum_{i \in R_j} (y_i - \hat{y}_{R_j}), \tag{6}$$

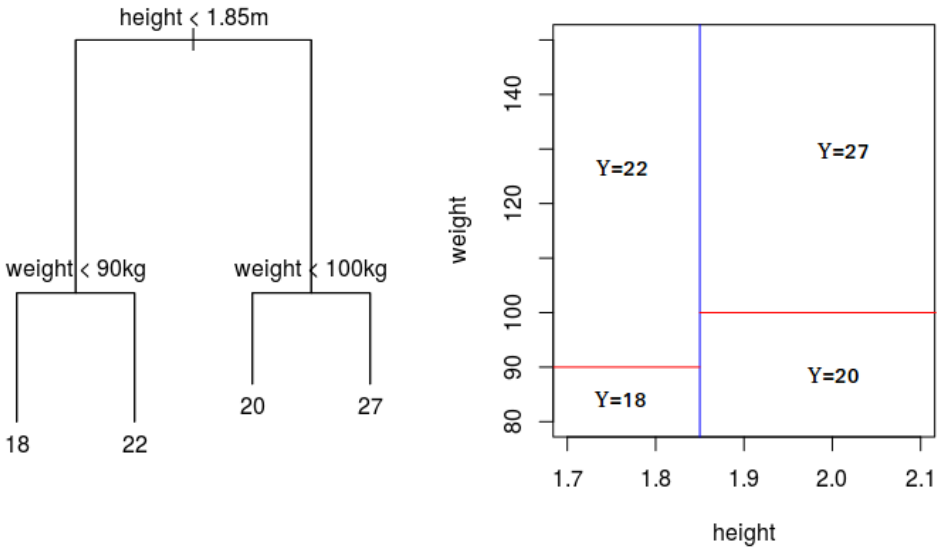
kde $\hat{y}_{R_j}$ je průměr vypočítaný z trénovacích dat pro oblast $R_j$, a $y_i$ je jednotlivá predikovaná hodnota. [12]

Model lze také popsat stromovým grafem sestávajícím se z uzlů a orientovaných hran. V každém neterminálním uzlu se strom větví – z uzlu vedou hrany do dvou nebo více dceřinných uzlů. Větvení je založeno na hodnotě jediného příznaku. Nejběžnější je binární větvení podle hodnoty podmínky nerovnosti ($x < c$), kde $x$ je hodnota příznaku

a $c$ je konstanta. Terminálnímu uzlu a zároveň pozorováním, která do něj patří, je přiřazena některá hodnota z oblasti $R_j$. [13]

Během samotné predikce algoritmus jen prochází stromem a na základě vstupních parametrů se nasměruje od kořenového uzlu k jednomu z terminálních stavů.

Na obrázku 6 je ukázána reprezentace regresního stromu pro predikci věku na základe výšky a váhy. Strom má 4 terminální stavy, a proto je příznakový prostor rozdělen na 4 části.



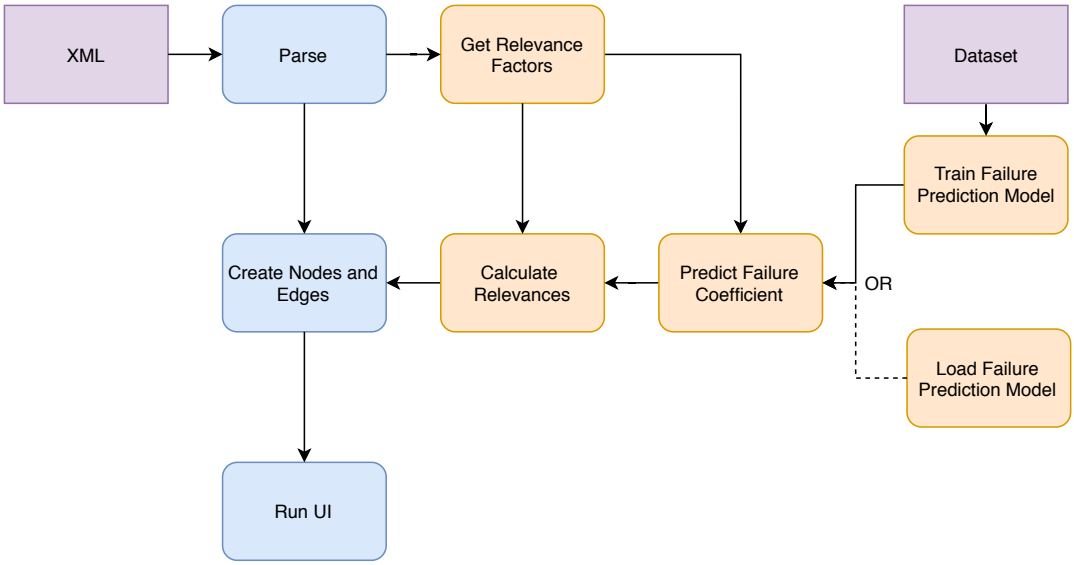

**Obrázek 6.** Příklad použití rozhodovacího stromu pro predikci věku (vlevo) a rozdělení prostoru predikce na 4 regiony, odpovídajícím 4 terminálním stavům (vpravo) [14]

V ML.NET je implementováno několik algoritmů pro predikci na základě rozhodovacích stromů:

- **Light gradient boosted machine (LightGMB)** je efektivní a přesný algoritmus s velkou výpočetní rychlostí a sníženou spotřebou paměti. Jedná se o implementaci algoritmu *Gradient boosting decision tree* (GBDT) rozšířenou o několik nových technik, umožnujících snadné zpracování velkého počtu vzorků dat a velkého množství příznaků. [15]
- **Fast Tree** je efektivní implementací *Multiple Additive Regression Trees* (MART). Fast Tree dobře zpracovává data z obrazu, je odolný vůči nenormalizovaným datům a je vysoce laditelný. [8]
- **Generalized additive model (GAM)** modeluje data jako sadu lineárně nezávislých příznaků podobně jako u lineárních modelům. Tento algoritmus využívá metodu GBDT k naučení takzvaných tvarových funkcí. Klíčovými vlastnostmi GAM jsou snadná interpretace, flexibilita a regularizace. [8]

# 4. Implementace

Jak bylo popsáno v kapitole 2, proces testování nástrojem Taster začíná rozborem XML souboru a inicializací instancí tříd modelu. Na obrázku 7 je zobrazen proces sestavení modelů po načtení XML souboru, kde je oranžovou barvou vyznačena část implementována v této práci.




**Obrázek 7.** Proces nahrávání modelu v Tasteru. Oranžovou barvou jsou označené části implementované během této práce

Pro každou hranu modelu se vytvoří instance třídy *Relevance*, do které se předají načtené parametry popisující funkcionalitu příslušnou dané hraně. Do této instance se také vloží prediktor koeficientu nespolehlivosti, který byl buďto naučen na nových datech nebo nahrán z uloženého souboru. Na základě vložených parametrů prediktor stanoví nespolehlivost dané funkcionality a následně se určí její relevance. Tento proces je popsán podrobněji v následujícím textu.

## 4.1. Rozbor XML

Na začátku je potřeba vytáhnout všechny parametry a indexy popisující jednotlivé funkcionality. Tyto parametry, jež jsou popsané v kapitole 3.1, jsou uložené v XML souboru v komentářích elementů příslušných hran. V komentářích jsou také uložené indexy bezpečnosti a komfortu.

**Výpis kódu 4.1 Příklad XML elementu pro popis hrany modelu**

```
<transition>
    <source ref="id2"/>
    <target ref="id2"/>
    <label kind="synchronisation" x="-1411" y="-603">
        dash_btn_touched!</label>
    <label kind="assignment" x="-1453" y="-620">
        Execute(/*TrunkFull.DashBtnPress()*/)</label>
    <label kind="comments" x="-1275" y="-603">
            S_IDX=4;
            C_IDX=6;
            CU_IDs=[ce702311-e622-e485-a86a-d6b688c6e4f7];
            S_IDs=[f800fba0-f297-d642-b7a4-b90ac3e32bb8];
            F_GUID=[d0db2337-f3db-e23f-a40d-ec8cb6f9b267];</label>
    <nail x="-1513" y="-620"/>
    <nail x="-1164" y="-620"/>
</transition>
```

Za proces rozboru XML v Tasteru odpovídá metoda *LoadTemplate()* třídy *TASystem*. Tato metoda inicializuje instance třídy hran (*Edge*) a třídy uzlů (*Node*) pro každý model. Tato metoda se rozšíří přidáním nové metody *ParseRelevanceStruct()* pro vytvoření instance třídy *Relevance*, která je následně vložena do každé hrany.

Nová metoda zkoumá komentáře XML elementů odpovídajících hranám. Každý parametr má přiřazený zápis s regulárními výrazy. Pokud se v komentáři najde řetězec shodný s tímto zápisem, hodnota parametru se uloží do proměnné. Pokud žádná shoda nebyla nalezena, do proměnné se zapíše nula.

**Výpis kódu 4.2 Příklad rozboru XML metodou *ParseRelevanceStruct()* pro index bezpečnosti**

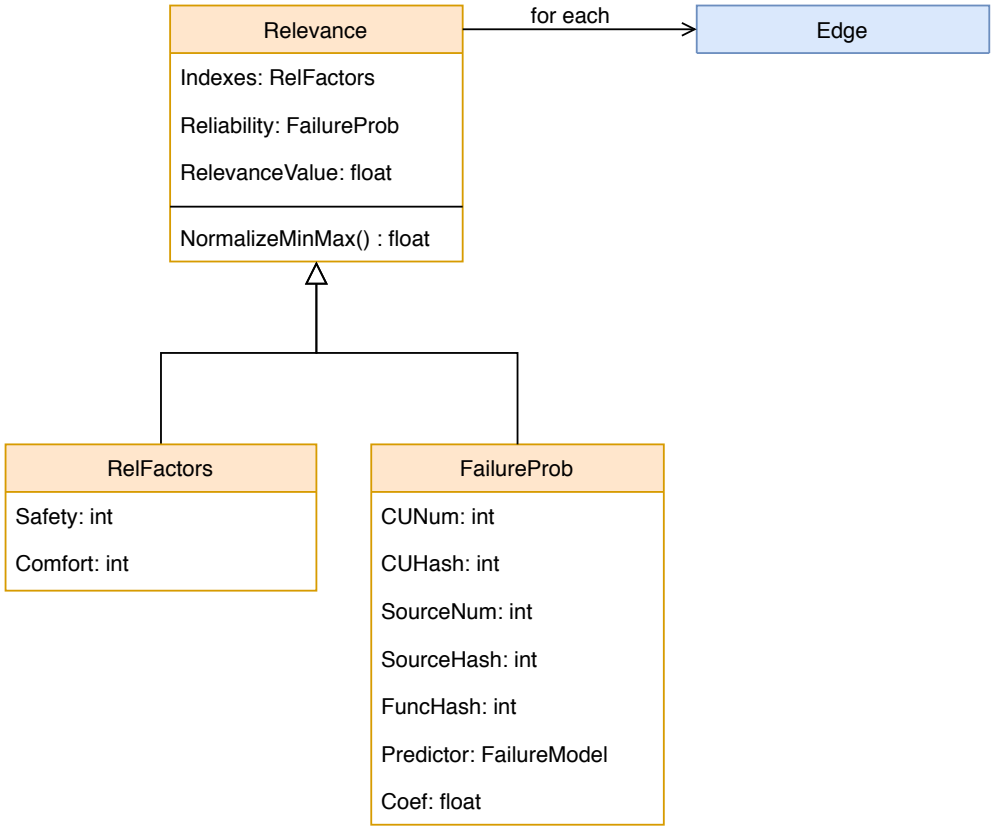
```
    var regexSafetyIdx = new Regex(@"S_IDX=\d+;");
    var regexInt = new Regex(@"\d+");
    var matchSafetyIdx = regexSafetyIdx.Match(comments);
    int safetyIdx = matchSafetyIdx.Success ?
            Int32.Parse(regexInt.Match(matchSafetyIdx.Value).Value)
            : 0;
```

Pokud byl načten seznam identifikátorů, například odpovídající ID dodavatelů, zapíšeme každý identifikátor do struktury *GUID* a následně pomocí hašovací funkce je vypočítána hash hodnota pro celý seznam. Implementace hašovací funkce je převzata z metody *GetHashCode()* třídy *string* z .NET Frameworku 4.7.1.[16]

Po načtení všech parametrů inicializujeme instanci třídy *Relevance*.

## 4.2. Třída Relevance

Třída Relevance se skládá z vnořené třídy *FailureProb* a vnořené struktury *RelFactors*, do nichž budou uložené vstupní parametry a prediktor koeficientu nespolehlivosti. Schéma teto třídy je vidět na obrázku 8.




**Obrázek 8.** Schéma třídy Relevance

Třída *FailureProb* zahrnuje parametry pro predikci nespolehlivosti příslušné funkcionality a samotný prediktor. Koeficient nespolehlivosti je implementován pomoci tzv. *property*, která se chová jako proměnná ale její hodnota se po přistoupení aktualizuje dle definovaného vzorce.

Pokud je vložen inicializovaný prediktor a je poprvé přistoupeno k *property Coef* odpovídající koeficientu nespolehlivosti, nejdříve se vyvolá metoda *Predict()* a poté se do koeficientu zapíše výsledek predikce. V případě, že prediktor je roven hodnotě *null*, zapíše se nula. Výsledek predikce se také uloží do proměnné *coefCache*, a pokud se znovu zavolá *property Coef*, pak se jen přečte hodnota z *coefCache* a nebude zbytečně volána funkce *Predict()*. Třída prediktoru je popsaná v následující kapitole.

**Výpis kódu 4.3 Definice *property Coef* odpovídající koeficientu nespolehlivosti**

```
1  public float coefCache = float.NaN;
2  public float Coef
3  {
4    get
5    {
6      if (float.IsNaN(coefCache))
7          coefCache = Predictor?
8              .Predict(CUNum, CUHash, SourceNum, SourceHash, FuncHash)
9              ?? 0.0f;
10     return coefCache;
11   }
12 }
```

Vnořená struktura *RelFactors* slouží k obalení indexů bezpečnosti a komfortu.

Výpočet relevance je také implementován pomocí *property*. Hodnota relevance se počítá z indexů a koeficientu nespolehlivosti jako lineární kombinace parametrů $\alpha, \beta, \gamma$, které jsou předem stanovené a vyjadřují vztahy mezi bezpečnosti, komfortem a nespolehlivostí, jejichž hodnoty jsou

$$\alpha = 0.5,\ \beta = 0.2,\ \gamma = 0.3.$$

Indexy bezpečnosti a komfortu nabývají hodnoty od 1 do 10, proto je potřebujeme normalizovat (t.j. zobrazit na interval $[0, 1]$) na základě maximální a minimální hodnoty v daném modelu. To lze provést za pomoci vztahu

$$X_{new} = \frac{X - X_{min}}{X_{max} - X_{min}}, \tag{7}$$

kde $X$ je hodnota indexu a $X_{min}, X_{max}$ jsou příslušné minimální, a maximální hodnoty. Minimum a maximum lze najít tak, že se ve třídě *Relevance* vytvoří statické proměnné, které budou globální pro každou inicializovanou instanci třídy, a každá nově načtená hodnota indexu bude porovnána se současnou hodnotou maxima nebo minima. Tím pádem po načtení celého modelu, každá instance třídy *Relevance* bude mít maximální a minimální hodnoty indexů.

**Výpis kódu 4.4 Výpočet hodnoty relevance**

```
1 public float RelValue =>   alpha * NormalizeMinMax(Indexes.Safety)
2                          + betta * NormalizeMinMax(Indexes.Comfort)
3                          + gamma * Failure.Coef;
```

Každá vytvořená instance třídy *Relevance* se předává do příslušné instance třídy *Edge*. Spolu s tím je implementována *property Relevance* ve třídě *Edge*, která vynásobí hodnotu relevance z instance třídy deseti a zaokrouhlí ji na celočíselnou hodnotu.

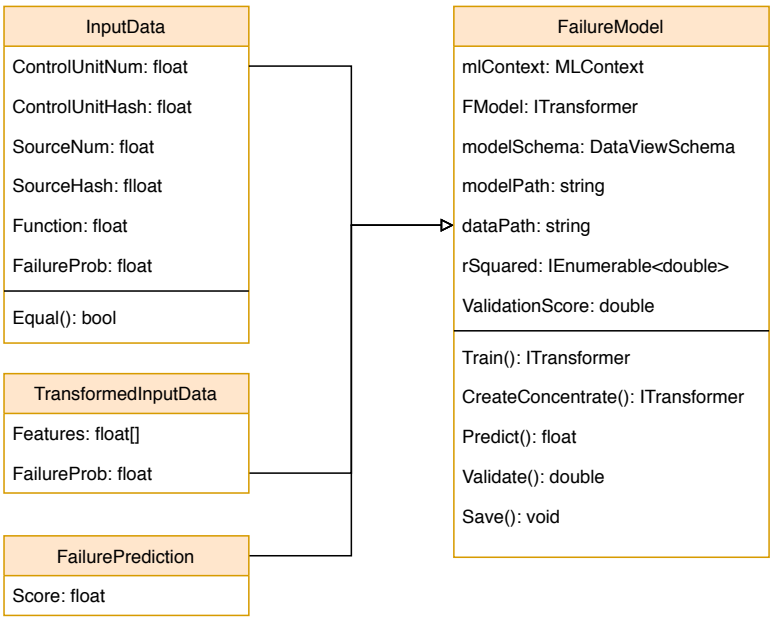
**Výpis kódu 4.5 *property Relevance* ve třídě *Edge***

```
1 public int Relevance => (int)(10 * (RelStruct?.RelValue ?? 0.0f));
```

Výhodou návrhu architektury této třídy je to, že se hodnota relevance počítá jen po její vyvolaní a také to, že prediktor se vytvoří jen jednou pro všechny hrany a tím pádem jej lze snadno vyměnit, což je v souladu s cílem práce.

## 4.3. Vytvoření prediktorů

Hlavním požadavkem návrhu architektury modelů pro predikci je snadná výměna a modifikace algoritmů. Proto je vytvořena abstraktní třída *FailureModel*, kterou lze používat jako šablonu pro všechny prediktory. Tato obecná třída obsahuje vnořené třídy *InputData* a *TransformedInputData* pro uložení trénovacích dat a třídu *FailurePrediction* pro uložení výsledku predikce. Tyto vnořené třídy jsou nezbytné pro využití framewroku ML.NET. Schéma třídy *FailureModel* je na obrázku 9.




**Obrázek 9.** Schema třídy *FailureModel*

Třída *InputData* má implementovanou metodu *Equals()*, která umožňuje porovnáni příznakových vektorů.

*FailureModel* obsahuje následující metody:

- Abstraktní metoda *Train()* pro naučení modelu.
- Metoda *CreateConcentrate()* pro vytvoření koncentrátu z dat.
- Metoda *Predict()*, jež provádí predikci pro vstupní příznakový vektor.
- Abstraktní metoda *Validate()*, jež provádí křížovou validaci.
- Metoda *Save()* umožňující uložení naučeného modelu.

Metody *Train()* a *Validate()* jsou abstraktní, tedy každý prediktor sestavený na základě *FailureModel* má svoji definici těchto metod. Ostatní metody jsou společné.

Pro použití nástrojů ML.NET třída *FailureModel* obsahuje proměnou *mlContext* pro instanci třídy *MLContext*. K uložení naučeného modelu a jeho schema slouží proměnné *FModel* a *modelSchema*. Proměnné *modelPath* a *dataPath* slouží k uložení cest k modelu a datům.

Jelikož každý prediktor má stejnou strukturu, lze popsat jen jeden. Příkladem může být *FastTreeFailurePredictor*, který predikuje spolehlivost na základě algoritmu *FastTree*. Při inicializaci instance třídy *FastTreeFailurePredictor* se zadávají následující parametry:

- *trainNewModel* určuje, jestli je potřeba vytvořit nový model nebo jej načíst ze souboru.
- *crossValidate* určuje, jestli je potřeba využit křížovou validaci pro naučení.
- *modelPath* obsahuje cestu k trénovacím datům
- *modelPath* obsahuje cestu k souboru s naučeným modelem

Pro naučení nového modelu se vyvolá metoda *Train()*. Proces naučení začíná načtením dat a jejich následnou transformací. Nejprve se pomocí metody *CreateConcentrate()* vytvoří koncentrát z dat, t.j. vypočítá se pravděpodobnost selhání pro každou unikátní sadu parametru, způsobem popsáným v kapitole 3.1.

**Výpis kódu 4.6 Vytvoření koncentrátu metodou *CreateConcentrate()***

```
var distinctElements = trainingDataEnumerable.Distinct();
var trainingDataConcentrate =
        distinctElements.Select(element =>
        {
            element.FailureProb = trainingDataEnumerable
            .Where(row => row.Equals(element))
            .Average(row => row.FailureProb);

            return element;
        });
```

Dále je potřeba připravit data k aplikaci algoritmu. Všechny parametry se sloučí do jednoho sloupce *Features* a pak se normalizuje sloupec s pravděpodobnostmi *FailureProb*. Potom se inicializuje instance algoritmu *FastTree*, tzv. *trainer*.

**Výpis kódu 4.7 Připravení dat k aplikaci algoritmu**

```
var dataPrep = mlContext.Transforms.Concatenate("Features")
       .Append(mlContext.Transforms.NormalizeMinMax("FailureProb"));

var dataPrepTransformer = dataPrep.Fit(dataConcentrate);
var transformedData = dataPrep.Transform(dataConcentrate);

var trainer = mlContext.Regression.Trainers.FastTree();
```

Pokud při inicializaci prediktoru byla zvolena křížová validace, transformovaná data a *trainer* se předají do metody *CrossValidate()*, která rozdělí vstupní data, naučí několik modelů a vybere z nich nejlepší. Tato metoda je popsána v kapitole 5.3. Následně se uloží metriky vyhodnocující účinnost modelů a schéma nejlepšího modelu.

**Výpis kódu 4.8 Naučení modelu s využitím křížové validace**

```
var cvResults = mlContext.Regression.CrossValidate(transformedData,
    trainer);

FModel = cvResults.OrderByDescending(fold => fold.Metrics.RSquared)
                  .Select(fold => fold.Model).ToArray()[0];
rSquared = cvResults.OrderByDescending(fold => fold.Metrics.RSquared)
                    .Select(fold => fold.Metrics.RSquared);
rms = cvResults.OrderByDescending(fold => fold.Metrics.RSquared)
               .Select(fold => fold.Metrics.RootMeanSquaredError);

modelSchema = transformedData.Schema;
```

Pokud křížová validace nebyla zvolena, model se naučí na celých datech pomocí metody *Fit()*. Potom se také uloží schéma modelu.

**Výpis kódu 4.9 Naučení modelu metodou *Fit()***

```
var trainingPipeline = dataPrep.Append(trainer);

FModel = trainingPipeline.Fit(trainingDataConcentrate);
modelSchema = trainingDataConcentrate.Schema;
```

Dále může být naučený model a jeho schéma uloženy pomocí metody *Save()*.

Pokud bylo zvoleno načíst model ze souboru, vyvolá se metoda *Load()*, jež načítá model a jeho schéma ze vstupní cesty.

K následující predikci slouží metoda *Predict()*, která bere na vstup parametry jednotlivé funkcionality. Z naučeného modelu se vytvoří zdroj predikce, který na základě vstupních parametrů sloučených do jednoho vektoru predikuje hodnotu nespolehlivosti funkce.

**Výpis kódu 4.10 Predikce pomocí modelu naučeném bez využití křížové validace**

```
var predEngine = mlContext.Model.CreatePredictionEngine
                 <InputData, FailurePrediction>(FModel);

InputData sampleData = new InputData()
{
    ControlUnitNum = ControlUnitNum,
    ControlUnitHash = ControlUnitHash,
    SourceNum = SourceNum,
    SourceHash = SourceHash,
    Function = Function,
    FailureProb = 0
};
predictionResult = predEngine.Predict(sampleData);
```

Model naučený s použitím křížové validace má jiné schéma, proto je pro vytvoření zdroje predikce navrhnuta třída *TransformedInputData*, která má místo jednotlivých parametrů jeden příznakový vektor. Na základě uloženého schématu metoda *Predict()* vybírá, jak vytvořit zdroj predikce.

Metoda *Validate()* slouží k validaci prediktoru. Tato metoda transformuje vstupní data a volá metodu *CrossValidate()*, stejně jako v metodě *Train()* s tím rozdílem, že se ukládá jen metrika $R^2$ ("R kvadrát"). Tato metrika je popsána v kapitole 5.2. Na základě průměrné hodnoty $R^2$ lze porovnávat účinnost prediktorů.

Třída *FailureModel* také obsahuje *property ValidationScore*. Pokud model příslušného prediktoru byl naučen s využitím křížové validace, hodnoty metriky $R^2$ jsou zapsány do proměnné *rSquared* a *ValidationScore* vrátí průměr těchto hodnot. V případě, že křížová validace nebyla provedena, *ValidationScore* vyvolá metodu *Validate()*.

**Výpis kódu 4.11 *property ValidationScore***

```
public double ValidationScore => rSquared?.ToArray()[0] ??
                                 Validate(dataPath);
```

Kromě *FastTreeFailurePredictor* jsou také navrhnuty prediktory *LightGBMFailurePredictor* a *GamFailurePredictor*, jež využívají pro naučení modelu algoritmy *LightGBM* a *GAM*.

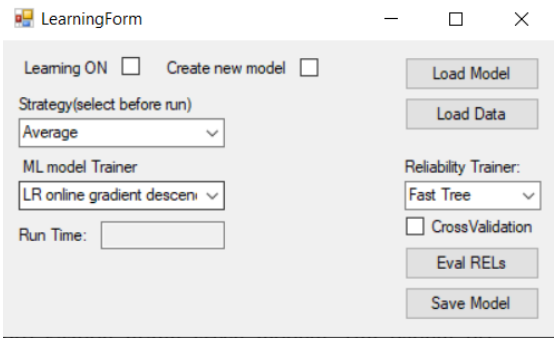
## 4.4. Uživatelské rozhraní

Pro výměnu algoritmu a výpočet relevance za běhu programu Taster, je potřeba rozšířit uživatelské rozhraní. Proto jsou do dialogového okna *Learning Form* přidané následující elementy:

- Tlačítko **Load Model** je určeno pro nahrávaní naučeného modelu pro predikci nespolehlivosti. Po stisknutí se vyvolá metoda *Load_Model_Click()*, a před uživatelem

se otevře další dialogové okno, kde lze zadat cestu k naučenému modelu. Tato cesta se uloží do struktury objektu *_runtime* třídy *TARuntime* do proměnné *fModelPath*. Také se změní stav proměnné *fCreateNewModel* na *false* a tím pádem se během dalšího výpočtu hodnot relevance nový model pro predikci jen načte z vybraného souboru.

- Tlačítko **Load Data** funguje podobně a je určeno pro nahrávání nové sady dat. Po stisknutí se vyvolá metoda *Load_Data_Click()* a uživatel v novém dialogovém okně může zadat cestu k novým datům. Cesta se opět uloží do struktury objektu třídy *TARuntime* do proměnné *fDataPath*. Stav proměnné *fCreateNewModel* se změní na *true* a tehdy, pokud bude inicializován nový prediktor, se vytvoří nový model na základě vložených dat.




**Obrázek 10.** Rozšířené uživatelské rozhraní Learning Form

- Vyskakovací menu **Reliability Trainer**. V tomto menu lze vybrat jaký algoritmus použít pro následující predikci. Typ prediktoru se uloží do proměnné *FPrediktorType*, která také patří do instance objektu třídy *TARuntime*.
- Tlačítko **Eval RELs**. Po stisku tohoto tlačítka se vyvolá funkce *Eval_RELs_Click()*, kde se vytvoří nový prediktor na základě výběru z vyskakovacího menu. Pokud nebyl načten nový model, nebo-li nová data, prediktor se načte z implicitně uvedené cesty (nebo-li se vytvoří nový prediktor na základě implicitní cesty k datům). Potom se do instance *Relevance* vloží nový prediktor a vynuluje se hodnota *coefCahce* pro každou hranu všech modelů, tím pádem se při následujícím vyvolání hodnoty relevance hodnota vypočítá za použití nového prediktoru.

**Výpis kódu 4.12 Proces obnovení prediktoru v Eval_RELs_Click()**

```
1 foreach (Template t in _runtime.TemplateInstances)
2 {
3   foreach (Edge edge in t.Edges)
4   {
5     edge.RelStruct.Failure.coefCache = float.NaN;
6     edge.RelStruct.Failure.Predictor = _runtime.FPredictor;
7   }
8 }
```

- Zaškrtávací políčko **Cross Validation** určuje, jestli je potřeba naučit model pomoci křížové validace.
- Tlačítko **Save Model** je určeno k uložení modelu použitého prediktoru. Po stisku se vyvolá metoda *Save_Model_Click()* a otevře se dialogové okno, kde lze zadat cestu pro uložení modelu. Následně se vyvolá metoda *Save()* příslušného prediktoru.

# 5. Validace

## 5.1. Generování dat

V době vzniku této bakalářské práce nejsou reálná data, získaná testováním reálných systému, k dispozici, proto je pro částečné ověření ML modelů potřeba trénovací a testovací sadu dat generovat.

Nejdříve je potřeba generovat pole unikátních příznakových vektorů, t.j. vektorů obsahujících parametry funkce (počet řídících jednotek, počet dodavatelů, jejich ID, a ID funkcionalit). To lze provést tak, že se náhodně vygeneruje 100 unikátních vektorů, s tím pravidlem, že čím větší je index v poli, tím větší jsou hodnoty počtů řídicích jednotek a dodavatelů, a to v rozmezí od 1 do 7.

Potom se náhodně vybere příznakový vektor z pole vektorů. Na základě pořadí vybraného vektoru v poli mu je přidělena hodnota pravděpodobnosti selhání tak, že jeho index se vydělí tisícem, tedy se vytvoří hodnoty od 0.1% do 10% a vektor s větším počtem řídících jednotek a dodavatelů má větší pravděpodobnost selhání. Pak se na základě vypočítané pravděpodobnosti přiřadí danému příznakovému vektoru jednička nebo nula. Tedy například - vektor má následující parametry:

| CU_num | CU_IDs | S_num | S_IDs | F_ID |
|---|---|---|---|---|
| 3 | 772317860 | 3 | 1169705007 | -1189174851 |

a má index v poli rovný 34, a tedy s pravděpodobností $\frac{34 \times 100\%}{1000} = 3.4\%$ se přiřadí tomuto vektoru chybový stav (jednička). Každý vektor se spolu s přiřazenou nulou nebo jedničkou zapíše jako nový řádek do textového souboru. Opakováním tohoto postupu se vytvoří sada trénovacích dat na 100 000 řádků ve tvaru popsaném v sekci 3.1, tabulka 1.

Podobným postupem, kde se ale namísto jedničky a nuly přiřadí rovnou pravděpodobnost selhání pro každý unikátní příznakový vektor, se vytvoří testovací data.

Pro generování dat byl zvolen programovací jazyk Python; celý kód lze najít v příloze A.

## 5.2. Ověření modelu

Na základě pouze generovaných dat nelze jednoznačně určit, který z algoritmu pro predikci je efektivnější. Nicméně pokud modely naučené na těchto datech dokáží predikovat správné hodnoty, lze říci, že použité algoritmy jsou funkční.

V následujícím experimentu jsou modely implementovaných prediktorů naučené na generovaných trénovacích datech. Poté je normalizován sloupec s pravděpodobnostmi selhání v testovacích datech, stejně jako v případě prediktorů (kapitola 4.3). Dále se pro každý příznakový vektor ze sady testovacích dat porovná hodnota predikovaná naučeným modelem s reálnou.

Pro porovnání a určení míry kvality regresních modelů existuje několik metrik:

- **Koeficient determinace $\mathbf{R}^2$** ("R kvadrát") se definuje jako jedna mínus podíl rozptylu chyb (tj. rozdílů mezi predikcemi modelu a skutečnými hodnotami nezávislé proměnné) a rozptylu nezávislé proměnné[17]:

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}} = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \overline{y})^2} \tag{8}$$

  kde $SS_{res}$ je suma čtverců chyb (residuí), $SS_{tot}$ suma kvadratických odchylek závislé proměnné $y$ od její střední hodnoty $\overline{y}$ a $\hat{y}_i$ je predikce i-tého pozorování. Rozsah tohoto koeficientu obvykle je od 0 do 1, a čím větší je hodnota, tím přesnější je model.

- **L2 Loss funkce** se používá k minimalizaci chyby a je definována jako suma kvadratických odchylek predikovaných hodnot od reálných[18]:

$$L2 = \sum_i (y_i - \hat{y}_i)^2 \tag{9}$$

  a tedy, čím je menší hodnota této funkce , tím menší je chybovost modelu.

- **Chyba RMS** nebo-li *Root Mean Square* je vlastně odmocnina z L2 Loss funkce.

$$RMS = \sqrt{\sum_i (y_i - \hat{y}_i)^2} \tag{10}$$

  Výsledky ověřovaní modelů na testovacích datech jsou v tabulce 4.

| Algoritmus | $R^2$ | L2 Loss |
|---|---|---|
| Fast Tree | 0,9424 | 0,0035 |
| LightGbm | 0,9227 | 0,0047 |
| Gam | 0,9250 | 0,0046 |

**Tabulka 4.** Výsledky validace modelů

Jak je vidět, hodnoty $R^2$ u všech algoritmů jsou větší než 0.9, a hodnoty *L2 Loss* funkce se blíží k nule, tedy lze považovat použité algoritmy za účinné.

## 5.3. Křížová validace

Křížová validace (angl. *Cross-Validation*) je metoda ověření modelů a jejich schopnosti predikovat neznámé vzorky dat. Tato metoda je výhodná pro omezený počet vzorků trénovacích dat a určuje obecnou účinnost modelů. [19] Postup křížové validace je následující:

1. Náhodně zamíchat vzorky trénovacích dat.
2. Rozdělit data na K skupin (parametr K se zadává na vstupu).
3. Pro každou jedinečnou skupinu
   a) Použít danou skupinu jako sadu testovacích dat
   b) Použít zbývající skupinu jako sadu trénovacích dat
   c) Naučit model na trénovacích datech a ověřit jeho funkčnost na testovacích.
   d) Uložit vyhodnocení modelů a metriky.

4. Shrnout výsledky a vybrat nejlepší naučený model na základě metrik.

Tato metoda se používá pro naučení modelu jako volitelná v navržených prediktorech.

Naučení modelů pomocí křížové validace vede k tomu, že ne všechna daná data jsou použitá pro naučení, což může vést i ke snížení přesnosti predikcí. Ale jelikož v reálném světě není možné vědět reálnou pravděpodobnost selhání funkcionalit, testovací data nejsou k dispozici, a proto křížová validace bude jedinou možností jak ověřit účinnost algoritmu. V tabulce 5 jsou zobrazeny výsledky křížové validace na trénovacích datech.

| Algoritmus | Nejlepší $R^2$ | Průměr $R^2$ | Nejlepší RMS | Průměr RMS |
|---|---|---|---|---|
| Fast Tree | 0,9065 | 0,6287 | 0,0719 | 0,1294 |
| LightGbm | 0,8652 | 0,5864 | 0,0864 | 0,1414 |
| Gam | 0,8408 | 0,6207 | 0,0939 | 0,1343 |

**Tabulka 5.** Výsledky křížové validace na trénovacích datech

Z výsledků je patrné, že model predikce naučený algoritmem *FastTree* má největší přesnost mezi použitými algoritmy. Výsledky se můžou lišit pro reálná data.

# 6. Závěr

Cílem této bakalářské práce bylo rozšíření nástroje Taster o možnost výpočtu relevance jednotlivých funkcionalit testovaných systémů na základě predikce pravděpodobnosti selhání. Po úvodním seznámení s modelovacím jazykem a analýze daných dat bylo úspěšně navrhnuto několik metod.

K výpočtu hodnoty relevance byla navrhnuta třída *Relevance*. Jelikož výpočet hodnoty relevance je implementován pomocí *property*, třída *Relevance* umožňuje snadnou výměnu konfigurací či použitých metod predikce.

Po analýze dat bylo rozhodnuto o použití algoritmů na základě rozhodovacích stromů pro predikci selhání. Celkem byly implementovány tři prediktory využívající algoritmy *FastTree*, *LightGMB* a *GAM*. Každý prediktor pomocí metody *Train()* umožňuje naučit model buď na celé datové sadě nebo pomocí metody křížové validace. Účinnost prediktorů lze porovnávat mezi sebou na základě hodnoty metriky $R^2$, jež je přístupná pomocí *property ValidationScore* nebo pomocí metody *Validate()*.

Navržené rozšíření uživatelského rozhraní umožňuje snadnou výměnu algoritmu či modelu za běhu programu Taster. V rozšířeném dialogovém okně *LearningForm* lze naučit nové modely a následně je uložit, nebo lze nahrát předem naučený model ze souboru.

Pro ověření navržených prediktorů byla generována sada trénovacích a testovacích dat. Jednotlivé modely byly naučeny na trénovacích datech a následně ověřeny na datech testovacích. Na základě výsledných metrik lze považovat implementované metody predikce za funkční.

V budoucnu budou navržené prediktory odzkoušeny s použitím reálných dat. Navržená architektura umožňuje snadné naučení nových modelů a jejich validaci. Výsledky této práce budou také vyzkoušeny při testování reálného hardwaru. Možnost výpočtu relevance s využitím predikce selhání může být použita v algoritmech průchodu testovacím modelem, čímž se může zvýšit efektivita integračních testů.

# Příloha A.

# Generování dat

**Výpis kódu A.1** Generování dat v Pythonu

```
items = [
# CU_N,     CU_ID,  S_N,     S_ID,        F_ID
[1,  -108226802,  3,  -363182128,  -1825340512],
[1,  -108226802,  3,  -363182128,  1169694764],
[1,  301541723,  2,  -1769788639,  1931265635],
[1,  301541723,  2,  -1769788639,  -1396597222],
[2,  1355003615,  2,  12833576,  2054067858],
[2,  1355003615,  2,  12833576,  352624763],
...
]
if __name__ == '__main__':

    # 1) TRAINING DATA:
    num_samples = 100000
    f = open("generated_data.txt", "w")
    f.write("ControlUnit_Num,  ControlUnit_ID,
             Source_Num,  Source_ID,
             Function_ID,
             Failure\n")

    for i in range(num_samples):

        item = random.choice(items).copy()    # choose 1 function
        f_prob = (items.index(item) + 1)/1000 # define failure prob

        item.append(np.random.choice(
            np.arange(0, 2), 1, p=[1 - f_prob, f_prob]
        ).tolist()[0])   # append 1 or 0

        f.write(','.join(map(str, item)))
        f.write("\n")

    f.close()

    # 2) TEST DATA:
    f = open("test_data.txt", "w")
    f.write("ControlUnit_Num, ControlUnit_ID,
             Source_Num, Source_ID,
             Function_ID,
             Failure_Prob\n")
    for item in items:
        f_prob = (items.index(item) + 1)/1000  # define failure prob
        item.append(f_prob)   # append 1 or 0
        f.write(','.join(map(str, item)))
        f.write("\n")
```

# Literatura


[1] Ina Schieferdecker Justyna Zander a Pieter J. Mosterman. *Model-Based Testing for Embedded Systems*. Zář. 2011.

[2] M. Bacic. "On hardware-in-the-loop simulation". In: *Proceedings of the 44th IEEE Conference on Decision and Control* (2005), s. 3194–3198.

[3] Alexandre David. "Uppaal SMC Tutorial". In: *International Journal on Software Tools for Technology Transfer* (2015).

[4] Ondřej Kobza. "Rozšíření testovacího nástroje Taster". bakalářská práce. České vysoké učení technické v Praze, 2018.

[5] Johan Bengtsson a Wang Y. "Timed Automata: Semantics, Algorithms and Tools". In: *Advanced Course on Petri Nets.* (2003), s. 87–124.

[6] Bc. Michal Veselka. "Integrační testování metodou Model-Based Testing - případová studie". magisterská práce. České vysoké učení technické v Praze, 2019.

[7] Ing. Jan Sobotka. "IMethods for Verification and Validation of Automotive Distributed Systems". Ph.D. Thesis. České vysoké učení technické v Praze, 2017.

[8] Microsoft. *How to choose an ML.NET algorithm.* URL: `https://docs.microsoft.com/en-us/dotnet/machine-learning/how-to-choose-an-ml-net-algorithm` (cit. 17. 05. 2020).

[9] Christopher M. Bishop. *Pattern recognition and machine learning*. springer, 2006.

[10] Yong Soo Kim. "Comparison of the decision tree, artificial neural network, and linear regression methods based on the number and types of independent variables and sample size". In: *Elsevier* (2008).

[11] Rebecca C. Steorts. *Tree Based Methods: Regression Trees*. URL: `http://www2.stat.duke.edu/~rcs46/lectures_2017/08-trees/08-tree-regression.pdf` (cit. 22. 04. 2020).

[12] Drago Spoljaric Brisevac Zlatko a Vlatko Gulam. "ESTIMATION OF UNIAXIAL COMPRESSIVE STRENGTH BASED ON REGRESSION TREE MODELS". In: *Rudarsko-Geolosko-Naftni Zbornik 29.1* (2014).

[13] Emil Kotrč Jan Klaschka. "KLASIFIKAČNÍ A REGRESNÍ LESY". In: (2004).

[14] *How Random Forests improve simple Regression Trees?* URL: `https://www.r-bloggers.com/how-random-forests-improve-simple-regression-trees/` (cit. 22. 04. 2020).

[15] Guolin Ke. "LightGBM: A Highly Efficient Gradient Boosting Decision Tree". In: *Advances in neural information processing systems* (2017).

[16] Microsoft. *.NET Framework 4.7.1*. URL: `https://referencesource.microsoft.com/#mscorlib/system/string.cs,833` (cit. 16. 05. 2020).

[17] G. Ciaburro. *Regression Analysis with R: Design and develop statistical nodes to identify unique relationships within data at scale*. Birmingham: Packt Publishing, 2018.

[18] *Differences between L1 and L2 as Loss Function and Regularization*. URL: `http://www.chioka.in/differences-between-l1-and-l2-as-loss-function-and-regularization/`.

[19] Jason Brownlee. *A Gentle Introduction to k-fold Cross-Validation*. URL: `https://machinelearningmastery.com/k-fold-cross-validation/` (cit. 06. 05. 2020).