Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informačních technologií

Využití .NET Core v průmyslové automatizaci

Diplomová práce

Autor: Bc. Jaroslav Langer
Studijní obor: Aplikovaná informatika

Vedoucí práce: doc. Mgr. Tomáš Kozel, Ph.D.

Hradec Králové Duben 2020

Prohlášení:

Prohlašuji, že jsem diplomovou práci zpracoval samostatně a s použitím uvedené literatury.

V Hradci Králové dne 8.4.2020 Bc. Jaroslav Langer

Poděkování:

Děkuji vedoucímu diplomové práce doc. Mgr. Tomáši Kozlovi, Ph.D. za metodické vedení práce, odborné rady, věcné připomínky, vstřícný přístup a vše, co mi pomohlo při zpracování diplomové práce.

Anotace

Název: Využití .NET Core v průmyslové automatizaci

Práce se zabývá představením technologií .NET Core a ML.NET. Dále je navrhnut scénář použití, na kterém jsou tyto technologie otestovány. Scénář je inspirován průmyslovým prostředím zaměřený na autonomní řízení výrobní linky.

V další části práce je tento scénář realizován v podobě návrhu architektury a následné implementace. Je navrhnuta simulace výrobní linky, systém řízení a sběru dat. Tyto systémy mezi sebou komunikují přes standartní průmyslový OPC UA protokol. Komunikaci zajišťuje OPC Server, jehož úlohou je zapouzdřit komunikaci s koncovými zařízeními výrobních linek. Systém řízení a sběru dat načítaná data ukládá do databáze a zároveň udržuje aktuální stav parametrů linky v paměti programu. Na základě těchto vstupů dochází k vyhodnocení a predikci, pomocí samoučícího algoritmu, nového nastavení výrobní linky. V závěru práce je shrnut výsledek a naměřená přesnost predikce učícího se algoritmu.

Klíčová slova: .net core, machine learning, ml.net, industry 4.0, opc.ua

Annotation

Title: Using .NET Core in industrial automation

The thesis deals with the introduction of .NET Core and ML.NET technologies. Furthermore, a use case scenario is proposed in which these technologies are tested. The scenario is inspired by an industrial environment focused on autonomous production line management.

In the next part of this work this scenario is realized in the form of architecture design and subsequent implementation. Simulation of production line, control and data collection system is proposed. These systems communicate with each other via industry standard OPC UA protocol. Communication is provided by OPC Server, whose task is to encapsulate communication with terminal equipment of production lines. The control and data acquisition system stores the read data into the database while maintaining the current status of the line parameters in the program memory. Based on these inputs, a new production line setting is evaluated and predicted using a self-learning algorithm. The conclusion and the measured accuracy of the learning algorithm prediction are summarized at the end of the thesis.

Keywords: .net core, machine learning, ml.net, industry 4.0, opc.ua

Obsah

1 Úvod ... 1
2 Cíl práce ... 2
3 Teoretická východiska ... 3
3.1 Objektový přístup ... 3
3.2 C# programovací jazyk ... 4
3.2.1 C# vs Java ... 4
3.3 .NET Core ... 4
3.3.1 Entity framework Core ... 5
3.4 Strojové učení ... 5
3.4.1 ML.NET ... 6
3.5 OPC Standard ... 25
3.5.1 OPC UA ... 26
3.5.2 OPC Server ... 27
3.6 Databáze ... 30
4 Návrh a implementace systému ... 31
4.1 Funkční analýza ... 31
4.2 Non funkční požadavky ... 31
4.3 Návrh architektury ... 31
4.3.1 Vrstva – průmyslové počítače ... 32
4.3.2 Vrstva OPC Server ... 33
4.3.3 Vrstva vlastní SW ... 33
4.4 Konfigurace OPC serveru ... 33
4.4.1 Nastavení pro simulaci ... 35
4.5 Pomocná OPC knihovna ... 35
4.6 Simulace výrobní linky ... 39

4.7 Hlavní aplikace 41
4.7.1 Pomocné služby pro uchování dat v paměti 41
4.7.2 Modul sběru dat 43
4.7.3 Modul AI řízení linky 47
4.7.4 Uživatelské rozhraní 51
5 Shrnutí výsledků 55
5.1 Zhodnocení splnění požadavků z analýzy 55
5.2 Uživatelský popis aplikace 55
5.2.1 Simulace výrobní linky 55
5.2.2 Systém sběru dat a autonomního řízení 56
6 Závěry a doporučení 59
7 Seznam použité literatury 61

Seznam obrázků

Obrázek 1 - ML.NET Struktura aplikačního kódu, převzato [9] 11
Obrázek 2 – Model ceny domu, převzato [9] 12
Obrázek 3 – ML Builder, výběr případu použití, vlastní zpracování 22
Obrázek 4 – ML Builder, výběr zdrojového souboru, vlastní zpracování 23
Obrázek 5 – ML Builder, výpočet modelu a výběr nejlepšího algoritmu, vlastní zpracování 23
Obrázek 6 – ML Builder, prezentace výsledků, vlastní zpracování 24
Obrázek 7 – ML Builder, vytvoření souborů pro použití vytrénovaného modelu, vlastní zpracování 24
Obrázek 8 – Výhody použití protokolu OPC, převzato [15] 26
Obrázek 9 – Role OPC Serveru, převzato [17] 27
Obrázek 10 – Návrh architektury, vlastní zpracování 32
Obrázek 11 – Instalace KepServeruEX, vlastní zpracování 34
Obrázek 12 – Výběr modulů a ovladačů, vlastní zpracování 34
Obrázek 13 – Příprava proměnných na OPC Severu, vlastní zpracování 35
Obrázek 14 – Struktura a závislosti knihovny, vlastní zpracování 36
Obrázek 15 – Algoritmus simulace, vlastní zpracování 40
Obrázek 16 – Struktura tabulky sběru dat, vlastní zpracování 43
Obrázek 17 – Výstup aplikace simulace, vlastní zpracování 56

Seznam tabulek

Tabulka 1 – Postup transformace textových dat, převzato [11] 21
Tabulka 2 – Výsledky měření úspěšnosti AI, vlastní zpracování 57

Seznam ukázek kódu

Ukázka kódu 1 – Ukázka jazyka C#, vlastní zpracování 4
Ukázka kódu 2 – Ukázka LINQ, převzato a upraveno [5] 5
Ukázka kódu 3 – Ukázka vstupních dat, převzato a upraveno [11] 14
Ukázka kódu 4 – Použití filtru nad sadou dat, převzato a upraveno [11] 14

Ukázka kódu 5 – Vstupní data pro ukázku nahrazení chybějící hodnoty, převzato a upraveno [11] 15
Ukázka kódu 6 – Nahrazení chybějící hodnoty, převzato a upraveno [11] 15
Ukázka kódu 7 – Vstupní data pro ukázku normalizace, převzato a upraveno [11] 16
Ukázka kódu 8 – Ukázka použití normalizace MinMax, převzato a upraveno [11]. 16
Ukázka kódu 9 – Vstupní data pro ukázku Binningu, převzato a upraveno [11] 17
Ukázka kódu 10 – Ukázka použití Binningu, převzato [11] 17
Ukázka kódu 11 – Vstupní data pro ukázku s kategorickými daty, převzato a upraveno [11] 18
Ukázka kódu 12 – Převedení nečíselných hodnot na číselné, převzato a upraveno [11] 18
Ukázka kódu 13 – Výsledek převedení nečíselných dat na číselné enum hodnoty, převzato [11] 18
Ukázka kódu 14 – Vstupní data pro ukázku práce s textovými daty, převzato a upraveno [11] 19
Ukázka kódu 15 – Ukázka použití transformace textových dat, převzato a upraveno [11] 19
Ukázka kódu 16 – Ukázka výsledku transformace textových dat, převzato [11] 19
Ukázka kódu 17 - Řetězení estimátorů, převzato [11] 20
Ukázka kódu 18 – Vyhodnocení modelu, převzato [9] 21
Ukázka kódu 19 – Použití vygenerovaného modelu z ML Builderu, vlastní zpracování 25
Ukázka kódu 20 – Výsledek testu programu z ML Builderu, vlastní zpracování 25
Ukázka kódu 21 – Interface IUaClient, vlastní zpracování 36
Ukázka kódu 22 – Metoda pro přidání položky ke sledování, vlastní zpracování 37
Ukázka kódu 23 – Metoda pro zrušení sledování položky, vlastní zpracování 38
Ukázka kódu 24 – Metoda pro zápis nové hodnoty, vlastní zpracování 38
Ukázka kódu 25 – Výpočet aplikovaného množství lepidla, vlastní zpracování 39
Ukázka kódu 26 – Program simulace, vlastní zpracování 41
Ukázka kódu 27 – Implementace pomocné služby pro uchování načtených dat v paměti, vlastní zpracování 42

Ukázka kódu 28 – Registrace singleton objektu, vlastní zpracování..... 42
Ukázka kódu 29 – Implementace pomocné služby pro logování, vlastní zpracování 43
Ukázka kódu 30 – Registrace objektu pro přístup k databázi, vlastní zpracování .. 44
Ukázka kódu 31 – Registrace singleton objektu UaClient, vlastní zpracování 44
Ukázka kódu 32 – Registrace služby běžící na pozadí, vlastní zpracování..... 44
Ukázka kódu 33 – Konstruktor služby sběru dat a inject závislostí, vlastní zpracování 45
Ukázka kódu 34 – Definice proměnných k vyčítání z OPC, vlastní zpracování 45
Ukázka kódu 35 – Obsluha události, změna sledované hodnoty, vlastní zpracování 45
Ukázka kódu 36 – Ukládání dat z paměti do databáze, vlastní zpracování..... 46
Ukázka kódu 37 – Implementace ukončovací metody, vlastní zpracování..... 46
Ukázka kódu 38 – Konstruktor třídy vlastního kontextu ML, vlastní zpracování ... 47
Ukázka kódu 39 – Trénování modelu, vlastní zpracování..... 47
Ukázka kódu 40 – Označení nerelevantních atributů pro ML, vlastní zpracování .. 48
Ukázka kódu 41 – Metoda pro predikci nové hodnoty, vlastní zpracování 48
Ukázka kódu 42 – Registrace služby řízení, vlastní zpracování 49
Ukázka kódu 43 – Služba AI řízení linky, vlastní zpracování..... 49
Ukázka kódu 44 – Implementace StartAsync() v AI řízení, vlastní zpracování..... 49
Ukázka kódu 45 – Spuštění vlákna pro obsluhu AI řízení, vlastní zpracování 50
Ukázka kódu 46 – Algoritmus pro přípravu učících dat, vlastní zpracování..... 51
Ukázka kódu 47 – Využití C# v razor stránkách, vlastní zpracování..... 52
Ukázka kódu 48 – Využitý C# proměnných v HTML, vlastní zpracování..... 53
Ukázka kódu 49 – Výpis logů, vlastní zpracování..... 54
Ukázka kódu 50 – Log vnitřních stavů systému, vlastní zpracování..... 56
Ukázka kódu 51 – Výpis parametrů výrobní linky a načtených dat, vlastní zpracování 57

Seznam zkratek

AI ..... Artificial intelligence

API ..... Application Programming Interface

DB ..... Database

DI ..... Dependency Injection

IDE ..... Integrated Development Environment

IoT ..... Internet of Things

MF ..... Matrix factorization

ML ..... Machine Learning

MS ..... Microsoft

OPC ..... Ole for Process Control

OPC DA ..... Data Access

OPC UA ..... Unified Architecture

ORM ..... Object-relational mapping

PCA ..... Principal Component Analysis

PLC ..... Programmable Logic Controller

SOA ..... Service Oriented Architecture

SQL ..... Structured Query Language

1 Úvod

V současné době se moderní průmysl chystá na další etapu, nazývanou Průmysl 4.0. S tímto rozvojem je spojen trend sběru mnohých dat a údajů o běžících procesech. Tato velmi objemná data však ve většině případů končí nevyužita v plnících se databázích. Přitom data mohou sloužit jako podklad pro samoučící algoritmy. V dnešní době jsou už většina moderních výrobních zařízení ovládána elektronicky a lze je tedy řídit počítačem. Průmysl 4.0 označuje dobu, kdy o běžících zařízeních víme vše a v případě výrobní společnosti seřizovač má naprostý přehled o své výrobní lince.

V rámci této práce bude myšlenka průmyslu 4.0 posunuta o další krok, kdy sbíraná data budou využita pro přímé zlepšení výkonu výroby a nížení nákladů za pomocí samoučících algoritmů. V rámci práce bude navrhnuta simulace výrobní linky a systém autonomního řízení, který bude v čase zlepšovat své možnosti. V ideálním případě by tak nemusel být seřizovač vyhrazen pro jednu výrobní linku, ale mohl by jich mít na starost více s tím, že jeho činnost nebude spočívat v reagování na změnu procesních parametrů formou úpravy řízení, ale bude mít dohled nad autonomními procesy, které budou reagovat na změny místo něj. Reakční doba systému autonomního řízení může být daleko kratší, než doba člověka, který se navíc nemusí v danou dobu pohybovat na svém pracovišti.

2 Cíl práce

V první části této práce autor popíše technologie, které budou využívány v praktické části. Cílem práce je popsat technologie .NET Core a ML.NET. Tyto technologie aplikovat na příkladu využitelném pro průmyslovou automatizaci. Součástí je návrh simulace výrobní linky, systém sběru dat a autonomního řízení za využití technologie Machine Learning pro .NET.

Cílem této diplomové práce je tak nejen rámcové představení technologie a teoretických postupů, ale také jejich aplikace do projektu, který autor popisuje.

3 Teoretická východiska

V této části diplomové práce se autor zabývá teoretickou částí zvoleného tématu. Dojde zde k vysvětlení pojmů a popisu technologií pojící se s tématem.

3.1 Objektový přístup

Objektově orientované programování je programové paradigma založené na konceptu vidění světa nebo systému jako sady objektů, které spolu vzájemně komunikují a reagují. Tyto objekty obsahují data a funkčnost. Pamatují si svůj stav a poskytují rozhraní jiným objektům. Objekt vzniká, když je vytvořena nová instance třídy, která předepisuje stavy, vlastnosti a chování objektu. Paradigma popisuje základní pravidla a vzorce, které programátor dodržuje. Mezi nejdůležitější patří například abstrakce, dědičnost, zapouzdření a polymorfismus. [1]

Abstrakcí se míní osvobození od některých detailů metod a vlastností jednotlivých objektů. Každý objekt lze vnímat jako „černou skříňku“, takže není možné jasně určit, jak funguje uvnitř, ale je možné odvodit, co lze očekávat. Například komunikační objekt odešle emailovou zprávu při vyvolání metody `SendEmail()`. Z toho můžeme odvodit výsledek, který je odeslanou zprávou, ale není možné jednoznačně určit, kterým serverem nebo poskytovatelem je zpráva odeslána.

Zapouzdření znamená zabránit přístupu k atributům z jiných objektů, aby se zabránilo nekonzistenci a nestabilitě objektu. Přístup je možný pouze prostřednictvím veřejných rozhraní.

Koncept dědičnosti znamená, že třídy, respektive objekty mohou převzít vlastnosti a metody svých předků.

Objekt má chování definované ve třídě. Tyto třídy však mohou realizovat stejné rozhraní, což znamená, že se s nimi pracuje stejně. Implementace metody je ale rozdílná. Jako příklad lze uvést připojení k databázi, kdy je možné dosadit objekt, který je optimalizovaný pro MS SQL, ale lze ho nahradit objektem optimalizovaným pro Oracle DB. Tento koncept se nazývá polymorfismus.

3.2 C# programovací jazyk

C# je jedním z nových vysokoúrovňových objektově orientovaných programovacích jazyků. Byl vyvinuta společností Microsoft, společně s platformou .NET, pro kterou je primárně určena. Jeho principy jsou založeny přímo na jazycích C/C ++ a Java, s nimiž má velmi podobnou syntaxi, a je tedy nepřímým potomkem jazyka C. Syntaxe jazyka C# je podobná jazyku C. C# lze použít k vytváření desktopových, webových a mobilních aplikací s připojením k databázovým systémům.

Jazyk je schválen standardizačními komisemi ECMA (ECMA-334) [2] a ISO (ISO/IEC 23270).

```
using System;
namespace AplikaceAhojSvete
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Ahoj vsichni!");
            Console.WriteLine("Zdraví Jaroslav Langer");
        }
    }
}
```

Ukázka kódu 1 – Ukázka jazyka C#, vlastní zpracování

3.2.1 C# vs Java

C# v některých prvcích pochází z Javy, ale nejen z Javy. Jak C#, tak Java jsou založeny na základech C. Pro některé programátory Java je C# jazyk, který se jim líbí, protože ho znají a přinášejí další výhody, jiní mají negativní přístup. Protože oba jsou založeny na C, můžeme říct, že C# umožňuje snadný přechod pro Java programátory. Syntaxe je velmi podobná, sémantika pak známá a pohodlná. [3]

3.3 .NET Core

.NET Core je Open Source vývojová platforma pro obecné účely udržovaná Microsoftem a komunitou .NET na GitHubu pod licencí MIT. Je pro různé platformy (podporující Windows, macOS a Linux) a dá se použít k sestavování aplikací pro zařízení, Cloud a IoT. [4]

Pro vývoj je možné využít jazyky C#, Visual Basic a F#. Tyto jazyky jsou, nebo je možné je integrovat do textových editorů a prostředí IDE, včetně sady Visual Studio, Visual Studio Code, Sublime Text a Vim.

U .NET Core si aplikace může nést svou vlastní verzi runtime. Na jednom počítači může být runtime nainstalován ve více verzích, čímž se riziko chyb způsobených upgradem značně snižuje – pokud aplikace s novým runtimem nefunguje, lze jí snadno říct, aby se spouštěla se starší verzí, a zároveň ji můžete vždy dodávat s runtimem, proti kterému byla otestována.
Aktuální verzí je .NET Core 3.1 [4].

3.3.1 Entity framework Core

Entity Framework (EF) Core je nová knihovna vycházející z původní EF6 pro klasický .NET Framework Využívá objektově relační mapování (O/RM), které vývojářům umožní pracovat s databází pomocí .NET objektů. Objektově relační mapování znamená, že tabulky databáze jsou mapovány na třídy programu.
Dotazování na data a práce s databází se realizuje pomocí Language Integrated Query (LINQ). [5]

```
using (var db = new ApplicationDb())
{
    var blogs = db.Blogs
        .Where(b => b.Rating > 3)
        .OrderBy(b => b.Created)
        .ToList();
}
```

Ukázka kódu 2 – Ukázka LINQ, převzato a upraveno [5]

3.4 Strojové učení

Strojové učení je aplikace umělé inteligence (AI), která umožňuje systémům automaticky se učit a zlepšovat ze zkušeností, aniž by byla explicitně naprogramována. Strojové učení se zaměřuje na vývoj počítačových programů, které mohou přistupovat k datům a používat je pro vlastní učení. [6]
Proces učení začíná pozorováním nebo daty, jako jsou příklady, přímá zkušenost nebo instrukce, aby bylo možné hledat vzorce v datech a v budoucnu se lépe rozhodovat na základě příkladů, které poskytujeme. Primárním cílem je umožnit počítačům, aby se učily automaticky bez zásahu člověka nebo asistence a podle toho upravovaly akce. [6]

3.4.1 ML.NET

ML.NET je platforma pro strojové učení s otevřeným zdrojovým kódem napříč platformami, díky níž je učení strojů přístupné vývojářům .NET.

ML.NET umožňuje vývojářům .NET vyvíjet, trénovat své vlastní modely a vkládat vlastní strojové učení do svých aplikací pomocí .NET, a to i bez předchozí zkušenosti s vývojem nebo vyladěním modelů strojového učení a zároveň mít výkonnou koncovou ML platformu načítání dat ze souborů datových souborů a databází, transformace dat a mnoho algoritmů ML.

ML.NET byl původně vyvinut v Microsoft Research a během posledního desetiletí se vyvinul do interního frameworku společnosti Microsoft, který se používá v mnoha skupinách produktů v Microsoft, jako jsou Windows, Bing, PowerPoint, Excel a další.

ML.NET umožňuje úlohy strojového učení, jako je klasifikace (například: klasifikace podpůrného textu, analýza sentimentu), regrese (například predikce ceny) a mnoho dalších úloh ML, jako je detekce anomálií, predikce časových řad, shlukování, hodnocení apod.

ML.NET také přináší .NET API pro tréninkové modely, využívající modely pro předpovědi, stejně jako základní komponenty tohoto frameworku, jako jsou učící se algoritmy, transformace a datové struktury ML. [7]

3.4.1.1 Typy modelů

Při sestavování modelu je potřeba určit, co od modelu očekáváme. Podle toho je nezbytné zvolit správný typ modelu. V ML.NET jsou implementovány následující modely.

3.4.1.1.1 Binary classification

Úloha strojového učení pod dohledem, která se používá k predikci dvou tříd (kategorií), k určení, do kterých datové instance patří. Jako vstup klasifikačního algoritmu slouží sada popisných příkladů, kde každé označení je celé číslo 0 nebo 1. Výstupem algoritmu binární klasifikace je klasifikátor, který lze použít k predikci třídy nových neznačených instancí. Mezi příklady binárních klasifikačních scénářů patří [8]:

- Princip mínění komentářů na Twitteru jako "kladné" nebo "záporné".
- Diagnostikuje, zda pacient trpí určitou chorobou, nebo ne.
- Rozhodnutí, zda označit e-mail jako "spam" nebo ne.
- Určení, zda fotka obsahuje určitou položku, například kočka nebo pes.

Pro dosažení nejlepších výsledků binární klasifikace musí být vstupní data vyvážená (tj. stejná počet kladných a záporných).
Algoritmy binární klasifikace [8]:

- AveragedPerceptronTrainer
- SdcaLogisticRegressionBinaryTrainer
- SdcaNonCalibratedBinaryTrainer
- SymbolicSgdLogisticRegressionBinaryTrainer
- LbfgsLogisticRegressionBinaryTrainer
- LightGbmBinaryTrainer
- FastTreeBinaryTrainer
- FastForestBinaryTrainer
- GamBinaryTrainer
- FieldAwareFactorizationMachineTrainer
- PriorTrainer
- LinearSvmTrainer

3.4.1.1.2 Multiclass classification

Úloha strojového učení, která se používá k předpovídání třídy dat instance. Vstup klasifikačního algoritmu je sada příkladů s popisky. Každý popisek obvykle začíná jako text. Výstup klasifikačního algoritmu je klasifikátor, který lze použít k predikci třídy nových neznačených instancí. Příklady klasifikačních scénářů pro více tříd zahrnují:

- Určení plemene psa jako "Husky", "Zlatý retriever", "Pudl" atd.
- Pochopíte recenze filmů jako "pozitivní", "neutrální" nebo "negativní".
- Kategorizace hodnocení hotelu jako "umístění", "cena", "čistota" atd.

Algoritmy určení s více třídami [8]:

- LightGbmMulticlassTrainer
- SdcaMaximumEntropyMulticlassTrainer
- SdcaNonCalibratedMulticlassTrainer
- LbfgsMaximumEntropyMulticlassTrainer
- NaiveBayesMulticlassTrainer
- OneVersusAllTrainer
- PairwiseCouplingTrainer

3.4.1.1.3 Regression

Úloha strojového učení, která se používá k předpovídání hodnoty popisku ze sady souvisejících funkcí. Popisek může mít jakoukoli reálnou hodnotu a nepochází z konečné sady hodnot jako v klasifikačních úkolech. Algoritmy regrese modelují závislost popisku na jeho souvisejících funkcích, aby určily, jak se popisek mění, protože se mění hodnoty funkcí. Vstup regresního algoritmu je sada příkladů s popisy známých hodnot. Výstupem regresního algoritmu je funkce, kterou lze použít k predikci hodnoty popisku pro všechny nové sady vstupních funkcí. Příklady regresních scénářů zahrnují [8]:

- Odhad ceny domu na základě atributů, jako je počet ložnic, umístění nebo velikost.
- Odhad budoucích cen za ceny na základě historických dat a současných vývojů na trhu.
- Odhad prodeje produktů na základě marketingových rozpočtů.

Regresní algoritmy [8]:

- LbfgsPoissonRegressionTrainer
- LightGbmRegressionTrainer
- SdcaRegressionTrainer
- OlsTrainer
- OnlineGradientDescentTrainer

- FastTreeRegressionTrainer
- FastTreeTweedieTrainer
- FastForestRegressionTrainer
- GamRegressionTrainer

3.4.1.1.4 Clustering

Úloha strojového učení, která není pod dohledem[1], se používá k seskupení instancí dat do klastrů, které obsahují podobné vlastnosti. Clustering je možné využít pro identifikaci relací v datové sadě, kterou nelze logicky odvodit pomocí procházení nebo jednoduchého sledování. Vstupy a výstupy algoritmu shlukování závisí na vybrané metodice. Příklady scénářů shlukování zahrnují [8]:

- Porozumění segmentům hostů v hotelu na základě zvyklostí a vlastností hotelu.
- Identifikujte segmenty zákazníků a demografické údaje, které pomohou vytvářet cílené marketingové kampaně.
- Kategorizace položek na základě výrobních metrik.

Algoritmus clusteringu [8]:

- KMeansTrainer

3.4.1.1.5 Anomaly detection

Vytváří model detekce anomálií pomocí hlavní komponenty analýzy (PCA). Detekce anomálií na bázi PCA pomůže vytvořit model ve scénářích, kde je snadné získat školící data z jedné třídy, ale je obtížné získat dostatečné vzorky anomálií. Příkladem jsou bankovní transakce.

Ve strojovém učení se PCA často používá v analýze dat, protože odhaluje vnitřní strukturu dat a vysvětluje odchylku dat. PCA pracuje s analýzou dat, obsahující více proměnných. Vyhledá korelaci mezi proměnnými a určuje kombinaci hodnot, které nejlépe zachycují rozdíly ve výsledcích. Tyto kombinované hodnoty funkcí se

[1] Podtřídou strojového učení, ve kterém požadovaný model najde skrytou strukturu v datech

používají k vytvoření kompaktnějšího funkčního prostoru známého jako hlavní komponenty.
Detekce anomálií zahrnuje mnoho důležitých úkolů ve strojovém učení [8]:

- Identifikujte potenciálně podvodné transakce.
- Výukové modely indikující možné narušení sítě.
- Hledání neobvyklých clusterů pacientů.
- Kontrola hodnot zadaných do systému.

Anomálie jsou podle definice vzácné, a proto shromáždit reprezentativní vzorek dat, který má být použit pro modelování, může být obtížné. Algoritmy obsažené v této kategorii byly navrženy speciálně pro řešení základních problémů modelování a školení modelů za pomoci nevyvážených datových sad.
Algoritmus detekce anomálií [8]:

- RandomizedPcaTrainer

3.4.1.1.6 Ranking

Úloha sestaví seřazení ze sady popisných příkladů. Tento soubor vzorků se skládá ze skupin případů, jejichž skóre je možné vyhodnotit pomocí daných kritérií. Hodnotící popisy jsou {0, 1, 2, 3, 4} pro každou instanci. Klasifikátor je vytrénován k třídění nových skupin instancí, kde je neznámé skóre pro každou instanci.
Algoritmy pro ranking [8]:

- LightGbmRankingTrainer
- FastTreeRankingTrainer

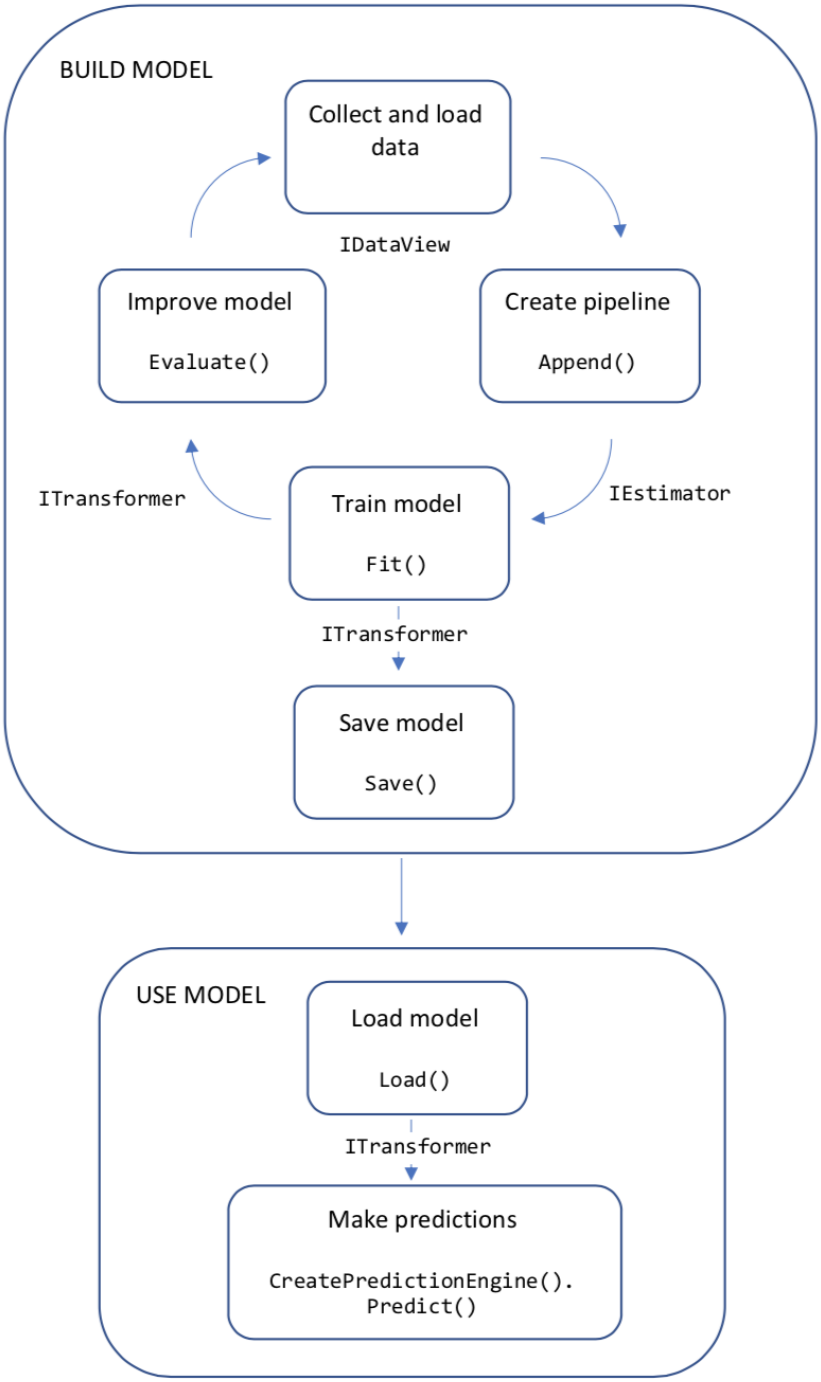
3.4.1.1.7 Recommendation

Úloha umožňuje vytvořit seznam doporučených produktů a služeb. V ML.NET se pro doporučení používá algoritmus filtrování a vytváření matic (MF), pokud existují data hodnocení daného produktu. Na základě těchto dat lze uživateli například doporučit další filmy, o které by mohl mít zájem.
Algoritmus doporučení [8]:

- MatrixFactorizationTrainer

3.4.1.2 Code workflow

Následující diagram, viz Obrázek 1, představuje strukturu aplikačního kódu a iterační proces vývoje modelu:



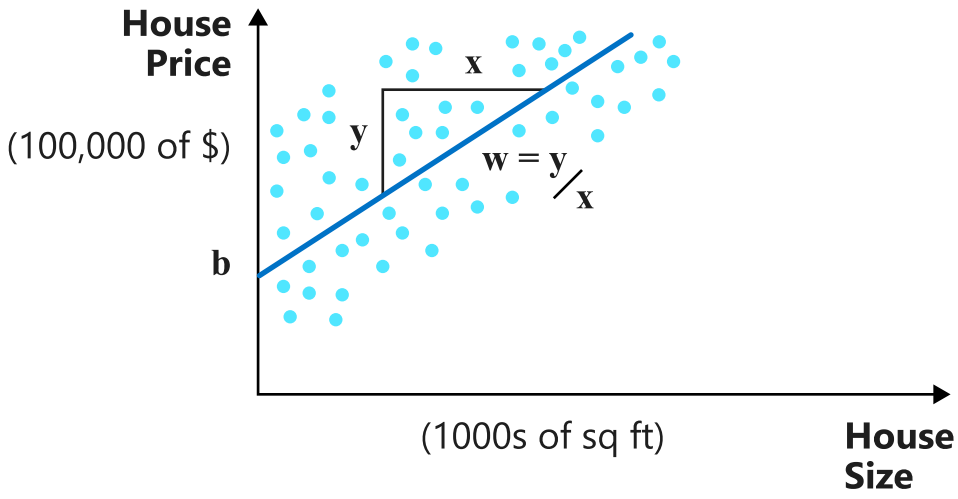
Obrázek 1 - ML.NET Struktura aplikačního kódu, převzato [9]

Proces popsán slovně:

- Shromažďování a načítání školicích dat do objektu `IDataView`
- Zadejte kanál operací pro extrakci funkcí a použití algoritmu strojového učení
- Výuka model voláním `Fit()` na kanálu [9]
- Vyhodnocení modelu a iterace pro zlepšení
- Uložit model do binárního formátu pro použití v aplikaci
- Načtení modelu zpátky do objektu `ITransformer`
- Vytvoření předpovědi voláním `CreatePredictionEngine.Predict()`

3.4.1.3 Machine learning model

Základní model je dvourozměrná lineární regrese, kde jedno průběžné množství je úměrné jinému. Příkladem je cena domu v závislosti na jeho velikosti.



Obrázek 2 – Model ceny domu, převzato [9]

`Price = b + velikost * w`. Parametry `b` a `w` jsou odhadované vytvořením řádku na základě množiny párů (Size, Price). Data, která se používají k vyhledání parametrů modelu, se nazývají školicí data. Vstupy modelu strojového učení se nazývají funkce[2]. V tomto příkladu je jedinou funkcí Size. Skutečné hodnoty[3]

[2] Anglicky features
[3] Anglicky ground-truth values

používané k výuce modelu Machine Learning se nazývají popisky[4]. V případě souboru pro trénování modelu, jsou popisky ceny. [9]

3.4.1.4 Příprava dat

Ve většině případů data, která jsou k dispozici, nejsou vhodná pro použití přímo k trénování modelu Machine Learning. Nezpracovaná data musí být připravena nebo předem zpracována před tím, než je možné ji použít k vyhledání parametrů modelu. Příslušná data možná budete muset převést z řetězcových hodnot na číselné znázornění. Vstupní data mohou obsahovat redundantní informace. V některých případech je bude potřeba zmenšit rozšířit, či normalizovat, nebo škálovat. [9]
Data často nejsou čistá a zhuštěná. Algoritmy strojového učení ML.NET očekávají zadání nebo funkce v jednom numerickém vektoru. Podobně hodnota pro předpověď. Proto je jedním z cílů přípravy dat získat data do formátu očekávaného ML.NET algoritmy.

3.4.1.4.1 Filtrování dat

V některých případech nejsou všechna data v datové sadě relevantní pro účely analýzy. Přístup k odebrání nerelevantních dat je filtrování. `DataOperationsCatalog` [10] obsahuje sadu operací k filtrování, které se využijí nad všemi daty z `IDataView` a ten následně vrátí pouze zajímavá data. Pomocí následujících vstupních dat, která jsou načtena do `IDataView`:

[4] Anglicky labels

```
HomeData[] myList = new HomeData[]
{
    new HomeData
    {
        Bedrooms=1f,
        Price=100000f
    },
    new HomeData
    {
        Bedrooms=2f,
        Price=300000f
    },
    new HomeData
    {
        Bedrooms=6f,
        Price=600000f
    }
};
```

Ukázka kódu 3 – Ukázka vstupních dat, převzato a upraveno [11]

Pro aplikaci filtru na základě hodnoty sloupce se použije metoda `FilterRowsByColumn`.

```
// Apply filter
IDataView filteredData =
mlContext.Data.FilterRowsByColumn(data, "Price", 200000, 1000000);
```

Ukázka kódu 4 – Použití filtru nad sadou dat, převzato a upraveno [11]

Výše uvedená ukázka v datové sadě vybere řádky s cenou mezi 200 000 a 1 000 000. Výsledek použití tohoto filtru vrátí jenom poslední dva řádky v datech a vyloučí první řádek, protože jeho cena je 100000 a ne mezi zadaným rozsahem. [11]

3.4.1.4.2 Nahrazení chybějících hodnot

Chybějící hodnoty jsou běžné jevy v datových sadách. Jediným způsobem, jak řešit chybějící hodnoty, je nahradit je výchozí hodnotou pro daný typ. Pokud hodnotu nelze vhodně vybrat, nahrazuje se střední hodnotou souboru.

Pomocí následujících vstupních dat, která jsou načtena do `IDataView`.

```
HomeData[] myList = new HomeData[]
{
    new HomeData
    {
        Bedrooms =1f,
        Price=100000f
    },
    new HomeData
    {
        Bedrooms =2f,
        Price=300000f
    },
    new HomeData
    {
        Bedrooms =6f,
        Price=float.NaN
    }
};
```

Ukázka kódu 5 – Vstupní data pro ukázku nahrazení chybějící hodnoty, převzato a upraveno [11]

Poslední prvek seznamu neobsahuje platnou hodnotu ceny. Pro nahrazení chybějící hodnoty ve ceny, se použije metoda `ReplaceMissingValues`.

```
var estimator =
       mlContext.Transforms
       .ReplaceMissingValues("Price",
        replacementMode: MissingValueReplacingEstimator.ReplacementMode.Mean);

ITransformer transformer = estimator.Fit(data);

IDataView transformedData = transformer.Transform(data);
```

Ukázka kódu 6 – Nahrazení chybějící hodnoty, převzato a upraveno [11]

Tuto metodu lze použít pouze pro číselné hodnoty.

Metoda `ReplaceMissingValues()` umožňuje práci ve více režimech. Výše uvedená ukázka používá *Mean* režim nahrazení, ve kterém se vyplní chybějící hodnota průměrnou hodnotou tohoto sloupce. Výsledek nahrazení vyplní *Price* vlastnost pro poslední prvek v datech s 200 000, protože se jedná o průměr 100 000 a 300 000. [11]

3.4.1.4.3 Normalizace

Normalizace je metoda předběžného zpracování dat, která slouží ke standardizaci funkcí, které nejsou ve stejném měřítku, což pomáhá zvýšit sblížení algoritmů. Například rozsahy pro hodnoty, jako je věk a příjem, se výrazně liší v rozmezí 0-100 a příjem je obecně v rozsahu od 0 do tisíců.

Pomocí následujících vstupních dat, která jsou načtena do `IDataView`:

```
HomeData[] myList = new HomeData[]
{
    new HomeData
    {
        Bedrooms = 2f,
        Price = 200000f
    },
    new HomeData
    {
        Bedrooms = 1f,
        Price = 100000f
    }
};
```

Ukázka kódu 7 – Vstupní data pro ukázku normalizace, převzato a upraveno [11]

Normalizace se dá použít na sloupce s jednou číselnou hodnotou i vektory. Normalizuje data ve sloupci *Price* pomocí normalizace min-max `NormalizeMinMax()` metodou.

```
var estimator = mlContext.Transforms.NormalizeMinMax("Price");

ITransformer transformer = estimator.Fit(data);

IDataView transformedData = transformer.Transform(data);
```

Ukázka kódu 8 – Ukázka použití normalizace MinMax, převzato a upraveno [11]

Původní cenové hodnoty [200000,100000] se převedou na [1, 0.5] použití *MinMax* vzorce normalizace, který generuje výstupní hodnoty v rozsahu 0-1. [11]

3.4.1.4.4 Binning

Binning převádí souvislé hodnoty do diskrétní reprezentace vstupu. Pokud je například jedna z funkcí věková. Místo použití skutečné věkové hodnoty binningu vytvoří rozsahy pro tuto hodnotu. 0-18 může být jedna přihrádka, další by mohla být 19-35, a tak dále.

Pomocí následujících vstupních dat, která jsou načtena do `IDataView`:

```
HomeData[] myList = new HomeData[]
{
    new HomeData
    {
        Bedrooms=1f,
        Price=100000f
    },
    new HomeData
    {
        Bedrooms=2f,
        Price=300000f
    },
    new HomeData
    {
        Bedrooms =6f,
        Price=600000f
    }
};
```

Ukázka kódu 9 – Vstupní data pro ukázku Binningu, převzato a upraveno [11]

Normalizační funkce Binning normalizuje data do skupin pomocí `NormalizeBinning()` metody. Parametr *maximumBinCount* umožňuje zadat počet přihrádek potřebných ke klasifikaci vašich dat. V tomto příkladu budou data vložena do dvou skupin.

```
var estimator = mlContext.Transforms
                    .NormalizeBinning("Price", maximumBinCount: 2);

var transformer = estimator.Fit(data);

IDataView transformedData = transformer.Transform(data);
```

Ukázka kódu 10 – Ukázka použití Binningu, převzato [11]

Výsledek binningu vytvoří meze skupiny pro [0, 200000, Infinity]. Proto výsledné přihrádky jsou [0,1,1], jelikož první pozorování je mezi 0 - 200 000 a ostatní jsou větší než 200 000, ale menší než nekonečno. [11]

3.4.1.4.5 Práce s kategorickými daty

Před použitím ML modelu se musí nečíselná data (například typy karoserií vozidel) převést na číselná.

Pomocí následujících vstupních dat, která jsou načtena do `IDataView`:

```
CarData[] myListOfCars = new CarData[]
{
    new CarData
    {
        Color = "Green",
        Type = "SUV"
    },
    new CarData
    {
        Color = "Blue",
        Type="Sedan"
    },
    new CarData
    {
        Color = "Red",
        Type="SUV"
    }
};
```

Ukázka kódu 11 – Vstupní data pro ukázku s kategorickými daty, převzato a upraveno [11]

Vlastnost kategorií `Type` lze převést na číslo pomocí metody `OneHotEncoding`.

```
var estimator =
      mlContext.Transforms.Categorical.OneHotEncoding("VehicleType");

ITransformer transformer = estimator.Fit(data);

IDataView transformedData = transformer.Transform(data);
```

Ukázka kódu 12 – Převedení nečíselných hodnot na číselné, převzato a upraveno [11]

Výsledná transformace převede textovou hodnotu `Type` na číslo. Při použití transformace se `Type` položky ve sloupci stanou následujícím:

```
[
    1, // SUV
    2, // Sedan
    1 // SUV
]
```

Ukázka kódu 13 – Výsledek převedení nečíselných dat na číselné enum hodnoty, převzato [11]

3.4.1.4.6 Práce s textovými daty

Před použitím ML modelu se musí nečíselná textová data převést na číselná.

Pomocí dat, jako jsou následující data, která byla načtena `IDataView`:

```
ReviewData[] listOfReviews = new ReviewData[]
{
    new ReviewData
    {
        Description="Tento produkt je dobrý",
        Rating=4.8f
    },
    new ReviewData
    {
        Description="Tento produkt je špatný",
        Rating=2.2f
    }
};
```

Ukázka kódu 14 – Vstupní data pro ukázku práce s textovými daty, převzato a upraveno [11]

Minimální krok pro převod textu na číselnou vektorovou reprezentaci je použití `FeaturizeText` metody. Použije-li se `FeaturizeText` transformaci, použije se pro vstupní textový sloupec řada transformací, což má za následek numerické vektory, které představují lineární normalizované slovo a znak n-gram[5].

```
var estimator = mlContext.Transforms.Text
                         .FeaturizeText("Description");

ITransformer transformer = estimator.Fit(data);

IDataView transformedData = transformer.Transform(data);
```

Ukázka kódu 15 – Ukázka použití transformace textových dat, převzato a upraveno [11]

Výsledná transformace by převedla textové hodnoty ve `Description` sloupci na číselný vektor, který vypadá podobně jako výstup níže:

```
[0.2041241, 0.2041241, 0.2041241, 0.4082483, 0.2041241, 0.2041241, 0.2041241,
0.2041241, 0.2041241, 0.2041241, 0.2041241, 0.2041241, 0.2041241, 0.2041241,
0.2041241, 0.2041241, 0.2041241, 0.2041241, 0.2041241, 0.2041241, 0.2041241,
0, 0, 0, 0, 0.4472136, 0.4472136, 0.4472136, 0.4472136, 0.4472136, 0]
```

Ukázka kódu 16 – Ukázka výsledku transformace textových dat, převzato [11]

Kombinací složitých kroků zpracování textu do *EstimatorChain* lze odebrat šum a případně snížit množství potřebných prostředků pro zpracování podle potřeby.

[5] Jako n-gram se označuje prosté sřetězení, posloupnost n jednotek stejného druhu (písmen, častěji však slov) v textu, převzato [20]

```
var estimator = mlContext.Transforms.Text.NormalizeText("Description")
    .Append(mlContext.Transforms.Text.TokenizeIntoWords("Description"))
    .Append(mlContext.Transforms.Text.RemoveDefaultStopWords("Description"))
    .Append(mlContext.Transforms.Conversion.MapValueToKey("Description"))
    .Append(mlContext.Transforms.Text.ProduceNgrams("Description"))
    .Append(mlContext.Transforms.NormalizeLpNorm("Description"));
```

Ukázka kódu 17 - Řetězení estimátorů, převzato [11]

Ukazatel `estimator` obsahuje podmnožinu operací provedených `FeaturizeText()` metodou. Výhodou složitějšího kanálu je řízení a viditelnost při transformaci aplikovaných na data.

Jako příklad je použita první položka, což je podrobný popis výsledků, které byly vytvořeny pomocí kroků transformace definovaných v `textEstimator` v následujících krocích:

Původní text: „*This is a good product*“ [11]

Transformace	**Popis**	**Výsledek**
1. NormalizeText	Ve výchozím nastavení převede všechna písmena na malá.	this is a good product
2. TokenizeWords	Rozdělí řetězec na jednotlivá slova.	["this","is","a","good","product"]
3. RemoveDefaultStopWords	Odebere stopslova jako je „is“ a „a“.	["good","product"]
4. MapValueToKey	Mapuje hodnoty na klíče (kategorie) na základě vstupních dat.	[1,2]
5. ProduceNGrams	Transformuje text na posloupnost po	[1,1,1,0,0]

	sobě jdoucích slov.	
6. NormalizeLpNorm	Škálování vstupů podle jejich LP-normy	[0.577350529, 0.577350529, 0.577350529, 0, 0]

Tabulka 1 – Postup transformace textových dat, převzato [11]

3.4.1.5 Vyhodnocení modelu

Jakmile je model vytrénovaný, je potřeba zjistit jeho chybovost. Provede se test nad novými testovacími daty, která nebyla v původním učícím souboru.

Každý typ úlohy strojového učení obsahuje metriky, které vyhodnocuje přesnost a přesnost modelu proti sadě testovacích dat.

Pro příklad ceny za dům byla využita regresní úloha. Vyhodnocení viz následující ukázka kódu.

```
HouseData[] testHouseData =
{
    new HouseData() { Size = 1.1F, Price = 0.98F },
    new HouseData () { Size = 1.9F, Price = 2.1F },
    new HouseData () { Size = 2.8F, Price = 2.9F },
    new HouseData () { Size = 3.4F, Price = 3.6F }
};

var testHouseDataView = mlContext.Data.LoadFromEnumerable(testHouseData);
var testPriceDataView = model.Transform(testHouseDataView);

var metrics = mlContext.Regression
                .Evaluate(testPriceDataView,labelColumnName: "Price");

Console.WriteLine($"R^2: {metrics.RSquared:0.##}");
Console.WriteLine($"RMS error: {metrics.RootMeanSquaredError:0.##}");

// R^2: 0.96
// RMS error: 0.19
```

Ukázka kódu 18 – Vyhodnocení modelu, převzato [9]

Metriky vyhodnocení sdělí, že chyba je nízká, a že korelace mezi předpokládaným výstupem a výstupem testu je vysoká. V praxi je vhodné porovnat více algoritmů a vybrat ten s nelepším výsledkem. [9]

3.4.1.6 Model builder

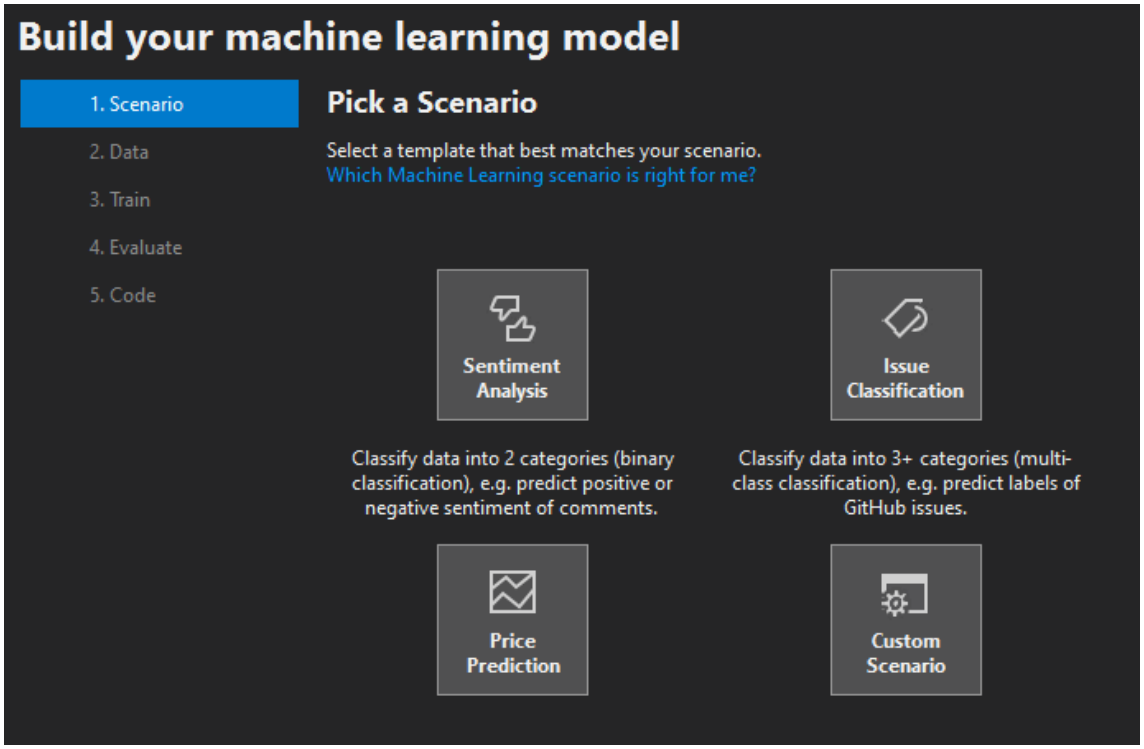
Model builder je rozšíření pro aplikaci Visual Studio, které umožní jednoduše vygenerovat potřebný kód, vybrat vhodný algoritmus a připravit model k použití

pomocí jednoduchého grafického průvodce. Uživatel vybere případ použití, zdrojový soubor, označí příslušné atributy – predikce, nebo zdroj funkce. Následně je proveden test algoritmů a výběr nejúspěšnějšího z nich a prezentace výsledku. Pokud je výsledek přijatelný, jsou do řešení přidány projekty se zdrojovým kódem na trénování a použití modelu. [12]

Aktuálně je rozšíření ve verzi Preview a při testování bylo funkční pouze pro některé scénáře.

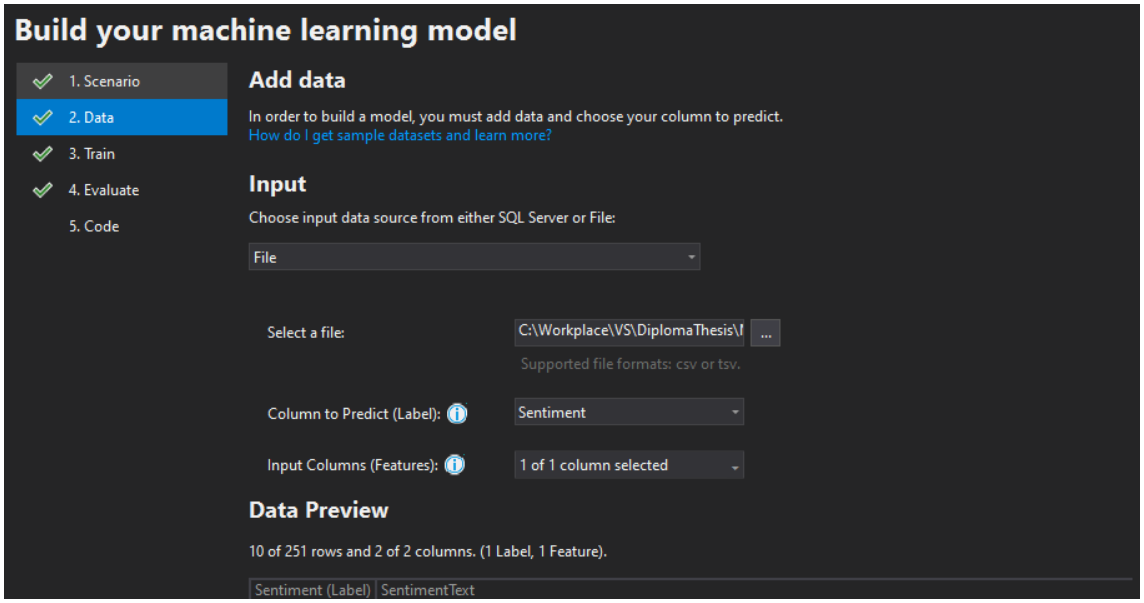
V následujících obrázcích se nachází ukázka vytvoření a vytrénování modelu za pomocí Model Builderu. Model má za úkol označit větu jako pozitivní, či negativní.

Na následujícím obrázku lze vidět výběr scénáře. Autor vybere binární klasifikaci – Sentiment Analysis. Binární klasifikací se rozumí zařazení vstupu do A nebo B skupiny. V tomto případě, zda je zdrojová věta pozitivní, nebo negativní.



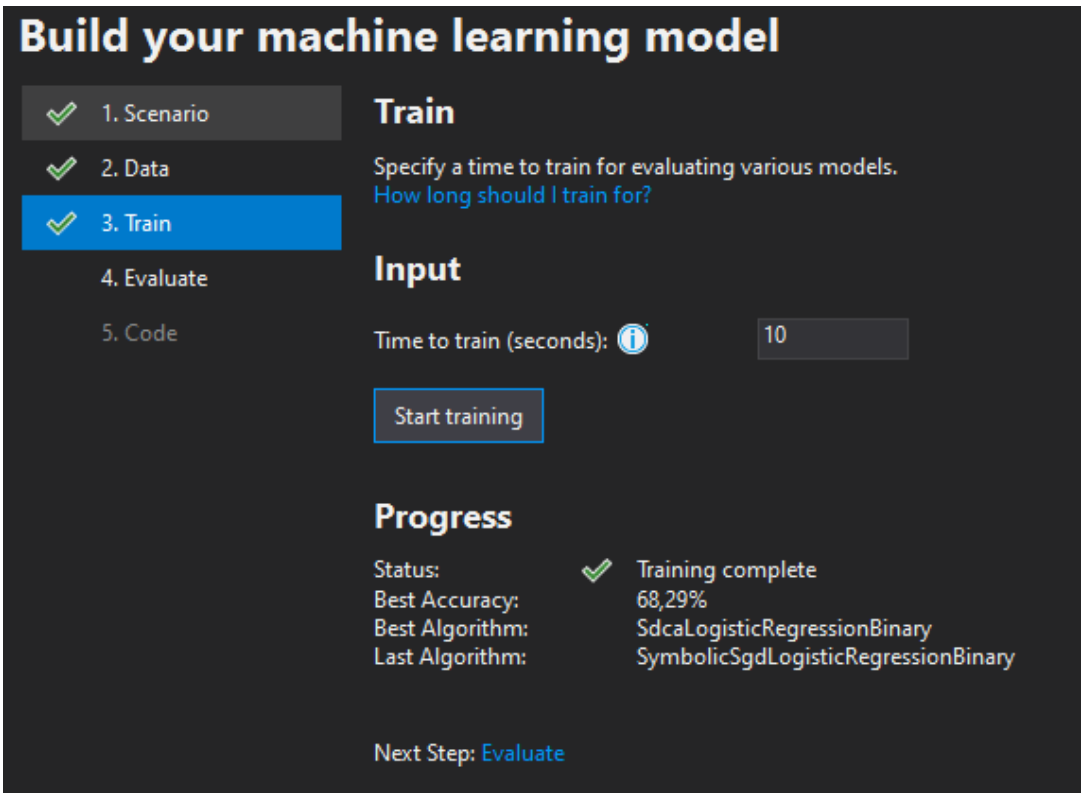
Obrázek 3 – ML Builder, výběr případu použití, vlastní zpracování

Na následujícím obrázku autor vybere zdroj dat a správně označí sloupce v souboru.



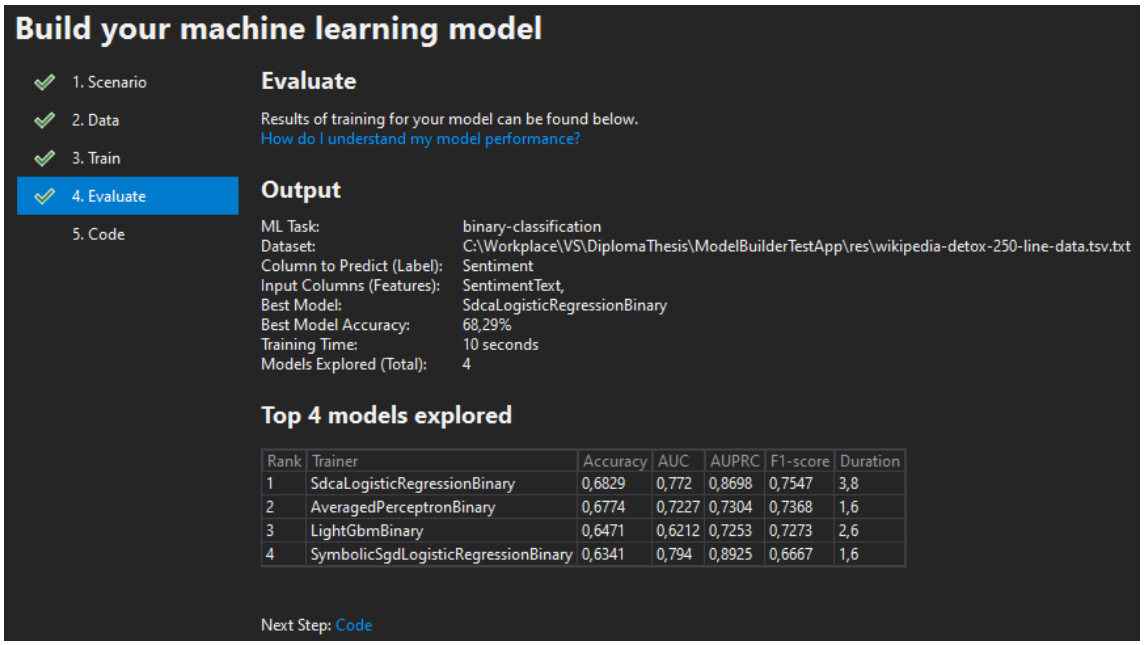
Obrázek 4 – ML Builder, výběr zdrojového souboru, vlastní zpracování

Na následujícím obrázku je spuštěn výpočet – trénování modelu, během kterého *builder* spustí dostupné algoritmy pro vybraný scénář a porovná výsledky.



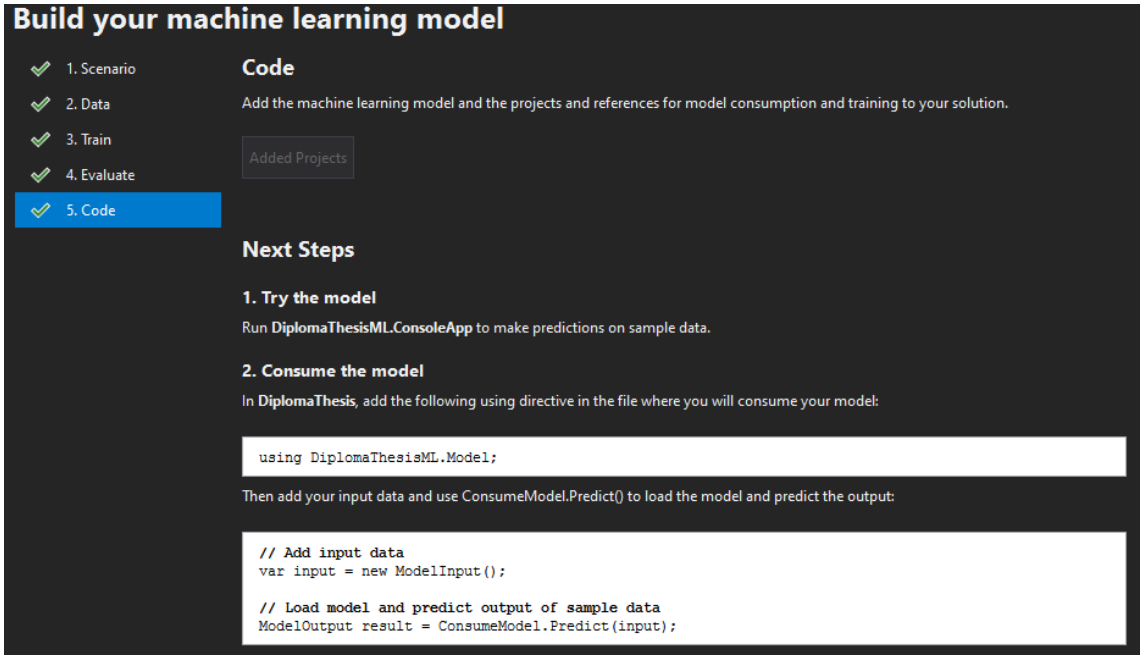
Obrázek 5 – ML Builder, výpočet modelu a výběr nejlepšího algoritmu, vlastní zpracování

Následuje krok vyhodnocení, kde je uživateli prezentována přesnost jednotlivých algoritmů. Forma prezentace výsledků závisí na zvoleném scénáři a typech spuštěných algoritmů. [12]



Obrázek 6 – ML Builder, prezentace výsledků, vlastní zpracování

V poslední části je vytrénovaný model zabalen a připraven k použití v projektu.



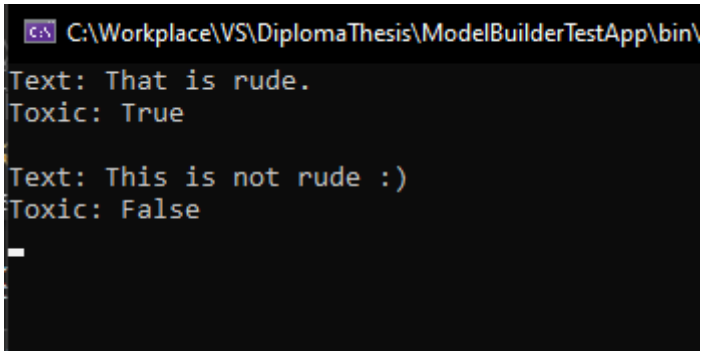
Obrázek 7 – ML Builder, vytvoření souborů pro použití vytrénovaného modelu, vlastní zpracování

V následující ukázce kódu je prezentováno použití vygenerovaného modelu. Vstupem je text – věta „*This is rude*“ a očekávaný výsledek je logická hodnota `True`. Druhým vstupem je opek věty a výsledek je očekávaná logická hodnota `False`.

```
class Program
{
    static void Main(string[] args)
    {
        var input = new ModelInput
        {
            SentimentText = "That is rude."
        };
        var result = ConsumeModel.Predict(input);
        Console.WriteLine($"Text: {input.SentimentText}");
        Console.WriteLine($"Toxic: {result.Prediction}");
        Console.WriteLine();

        input = new ModelInput
        {
            SentimentText = "This is not rude :)"
        };
        result = ConsumeModel.Predict(input);
        Console.WriteLine($"Text: {input.SentimentText}");
        Console.WriteLine($"Toxic: {result.Prediction}");

        Console.ReadKey();
    }
}
```

Ukázka kódu 19 – Použití vygenerovaného modelu z ML Builderu, vlastní zpracování

Následující obrázek je výsledkem spuštění programu a prezentace výsledku provedeného testu.



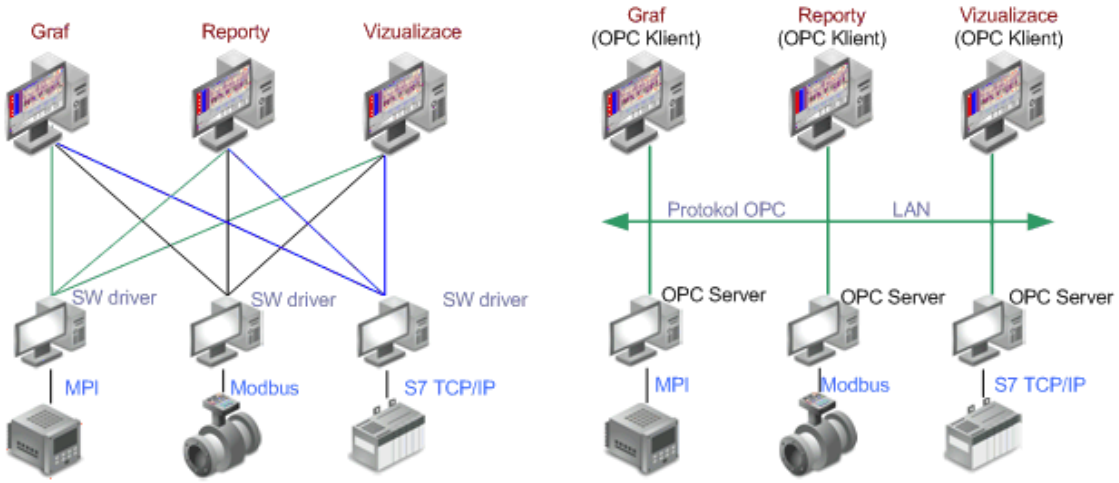
Ukázka kódu 20 – Výsledek testu programu z ML Builderu, vlastní zpracování

3.5 OPC Standard

OPC je řada komunikačních standardů a specifikací spravovaných Nadací OPC, která byla založena v roce 1996. [13]

OPC je komunikační protokol navržený k vytvoření jednotného komunikačního rozhraní mezi hardwarovými a softwarovými produkty průmyslové automatizace. [14]

S OPC mohou uživatelé (např. Systémoví integrátoři) integrovat hardware a software různých dodavatelů do svých projektů bez ohledu na komunikační rozhraní těchto komponent. Jedinou podmínkou je existence rozhraní OPC pro obě strany nebo existence příslušného serveru OPC pro použitý hardware a rozhraní klienta OPC pro použitý software. [15]



Obrázek 8 – Výhody použití protokolu OPC, převzato [15]

Používané OPC specifikace:

- OPC DA (Data Access)
- OPC UA (Unified Architecture)
- OPC AE (Alarms & Events)
- OPC HDA (Historical Data Access)
- OPC XML-DA

Protokol DA byl postupně nahrazen protokolem UA, který je stabilnější a nabízí lepší možnosti zabezpečení. [16]

3.5.1 OPC UA

OPC Unified Architecture (UA) odpovídá dnešním i budoucím požadavkům průmyslových komunikačních potřeb. Sjednocuje a rozšiřuje jednotlivé standardy

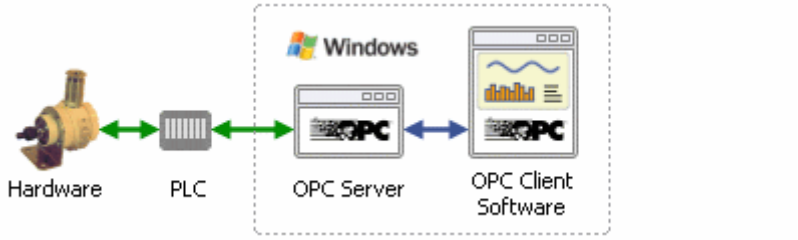
OPC první generace (nazývané také OPC COM nebo OPC Classic, DA) pomocí paradigmatu architektury orientované na služby (SOA). Tento výsledek je platformě nezávislá, škálovatelná a vysoce výkonná komunikační infrastruktura. Použití procesní a měřicí technologie v malých zařízeních s jejich specializovanými operačními systémy je stejně dobře možné jako použití v podnikových aplikacích na strojích Unix / Linux nebo Mainframes.

Bohatý informační model může představovat komplexní vztahy dat a jejich sémantiku. Speciálně navržené protokoly nabízejí vysokou rychlost komunikace a interoperabilitu.

OPC UA dále poskytuje bezpečnostní mechanismy, jako je autentizace, autorizace, šifrování a integrita dat na základě nejnovějších kryptografických standardů, jako jsou PKI, AES a SHA. [16]

3.5.2 OPC Server

OPC Server je program, který konvertuje komunikaci použitou PLC[6] do OPC protokolu. Umožňuje OPC klientům přístup k datům PLC, nebo posílání příkazů PLC. [17]



Obrázek 9 – Role OPC Serveru, převzato [17]

V rámci diplomové práce autor využil OPC KepServerEx společnosti KepWare, který umožňuje uživatelsky definovat simulace, pro jednodušší testování během vývoje.

3.5.2.1 Integrované funkce ovladače simulací

Ovladač simulací implementuje vybrané funkce, které lze využít pro testovací projekt, který tak může pracovat s reálně vypadajícími daty.

[6] Malý průmyslový počítač, který ovládá jedno či více zařízení

K dispozici jsou následující funkce:

- Ramp
- Random
- Sine
- User

3.5.2.2 Funkce Ramp

Funkci RAMP lze použít k vytvoření hodnoty, která se zvyšuje nebo snižuje v číselném rozsahu. Dolní a horní mez by měly být použity k nastavení požadovaného rozsahu. Nízké nebo vysoké limity mohou být upraveny tak, aby aplikovaly offset na generovaná data. Hodnota přírůstku může být kladná nebo záporná hodnota. Pokud je hodnota přírůstku kladná, generuje hodnota rampy od dolního limitu k vysokému limitu požadovanou rychlostí. Je-li hodnota přírůstku záporná, generuje hodnota rampy od nejvyššího limitu k nejnižšímu limitu požadovanou rychlostí. Hodnoty dolního limitu, vysokého limitu a přírůstku lze zadat buď jako celá čísla, nebo ve formátu s pohyblivou řádovou čárkou. [18]

Příklady:

```
RAMP (120, 35, 100, 4)
```

Tím se vytvoří hodnota, která se zvýší z 35 na 100 zvýšených o 4 každých 120 milisekund.

```
RAMP (300, 150,75, 200,50, -0,25)
```

Tím se vytvoří hodnota, která se sníží z 200,50 na 150,75 sníženou o 0,25 každých 300 milisekund.

3.5.2.3 Funkce Random

Funkci lze použít k vytvoření položky, která se náhodně změní v určitém číselném rozsahu. Dolní mez a horní mez by měly být použity k nastavení požadovaného rozsahu. Dolní nebo horní mez lze upravit tak, aby se na generovaná data aplikoval offset. [18]

Příklad:

```
RANDOM (30, -20, 75)
```

Tím se vytvoří hodnota, která se náhodně mění v rozmezí -20 až 75 rychlostí 30 milisekund.

3.5.2.4 Funkce Sine

Funkci SINE lze použít k vytvoření položky, která sleduje sinusovou změnu hodnoty. Dolní mez a horní mez by měly být použity k nastavení požadovaného rozsahu. Dolní nebo horní mez lze upravit tak, aby se na generovaná data aplikoval offset. Vlastnost Frequency lze použít k určení požadovaného tvaru vlny v Hertzech. Maximální efektivní frekvence je asi 5 Hz. Platný rozsah pro vlastnost Frekvence je 0,001 až 5 Hz. Vlastnost Phase může být použita k vyrovnání sinusové vlny generované specifickým úhlem. Fáze by měla být zadána v rozmezí 0,0 až 360,0. Vlastnost Rate v tomto případě hraje klíčovou roli při fungování této simulační funkce. Aby se z této funkce získal dobrý sinusový výstup, musí být rychlost alespoň dvakrát rychlejší než požadovaná frekvence. Například pokud si uživatelé přejí sinusovou vlnu 5 Hertzů, která se mění rychlostí přibližně 200 milisekund, měla by být vlastnost Rate nastavena na 100 milisekund maximálně. Pro dosažení nejlepších výsledků sinusové vlny doporučujeme nastavení frekvence na 10 nebo 20 milisekund. Platný rozsah frekvence pro funkci SINE je 10-1000 milisekund. [18]

Příklad:

```
SINE (10, -40, 40, 2, 0)
```

Tím se vytvoří sinusová hodnota s frekvencí 2 Hz, která se pohybuje od -40 do 40 bez fázového posunu.

3.5.2.5 Funkce User

Funkce USER poskytuje maximální flexibilitu při určování, jaký typ dat simulační funkce vrací. Na rozdíl od ostatních funkcí, které fungují v určitém rozsahu, lze funkci USER použít k určení sady číselných nebo řetězcových hodnot, které mají být cyklicky procházeny specifikovanou rychlostí. Zadané hodnoty se používají k určení datového typu této položky. Například pokud je jako jedna z uživatelských hodnot zadána hodnota 100,45, bude výstup simulačního objektu považován za data s pohyblivou řádovou čárkou. Pokud by jedna ze zadaných hodnot byla „Hello World“,

výstup simulačního objektu by se považoval za řetězcová data. Tyto výchozí výběry lze přepsat zadáním požadovaného typu dat, když je položka definována. [18]

Příklady:

```
USER(250, Hello, World, toto je, a, test)
```

Tím se vytvoří hodnota řetězce datového typu, který se mění z jednoho textového slova v sekvenci na další rychlostí 250 milisekund.

```
USER(20,1,25,100,56,200,11,75,1)
```

Tím se vytvoří hodnota typu float datového typu, která se mění z jedné hodnoty s pohyblivou desetinnou čárkou v sekvenci na další rychlostí 20 milisekund.

```
USER(50,1,1,0,1,0,1,0,0,1,1,1,0,0,0)
```

To generuje hodnotu typu BooleanClosedRepresents nebo odkazuje na hodnotu True nebo False (zapnuto nebo vypnuto). které se mění z jednoho booleovského stavu v pořadí na další rychlostí 50 milisekund. To lze použít k vytvoření velmi složitých bitových vzorců.

3.6 Databáze

Databáze je uspořádaná sada dat. Součástí databázového systému je SŘDB, což jsou softwarové nástroje pro přístup a manipulaci s daty. Mezi hlavní výhody patří například výrazně rychlejší přístup k datům než při čtení a přístupu k souboru. Umožňuje přímý přístup k datům, takže není nutné je číst jako celek. K jedné databázi lze přistupovat paralelně s jinými procesy, obsahuje také systém uživatelských práv. Tyto funkce usnadňují vyhledávání datových sad, které splňují požadovaná kritéria. [19]

4 Návrh a implementace systému

Tato část diplomové práce popisuje jednotlivé části návrhu a realizace řešení jednotlivých vrstev výsledného systému.

4.1 Funkční analýza

Informační systém musí pokrýt následující okruhy:

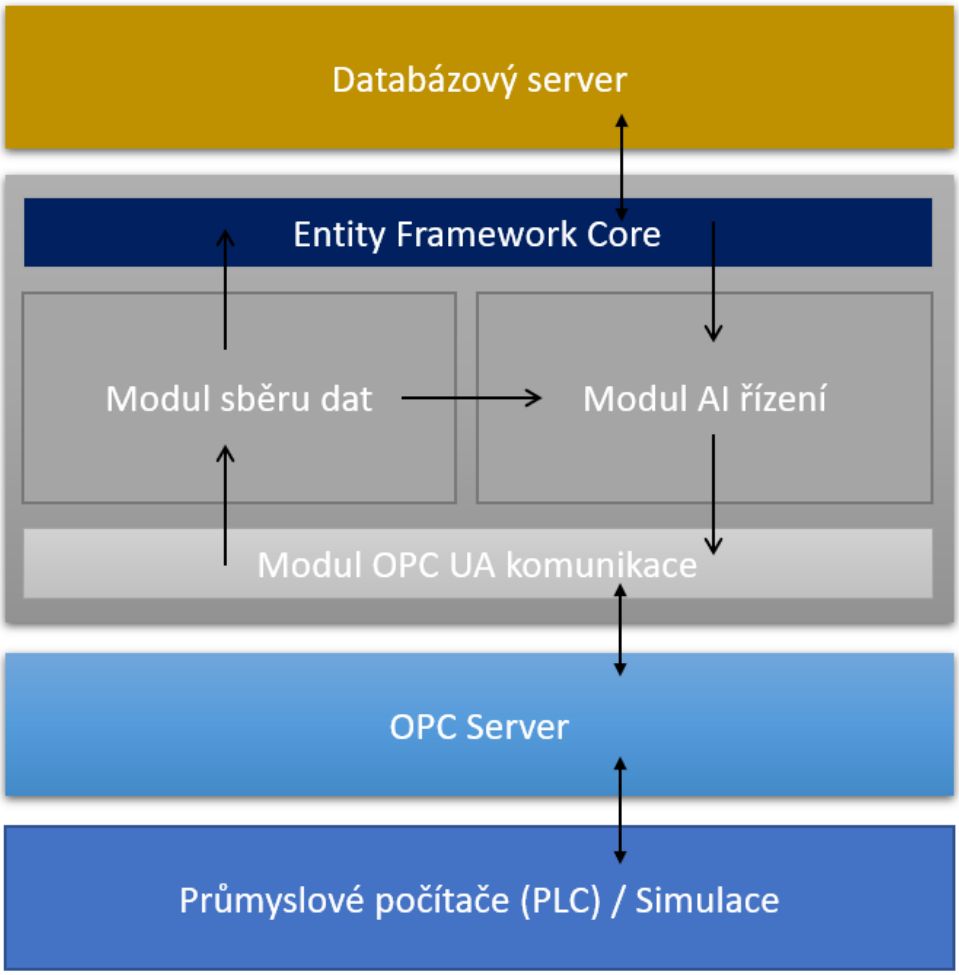
1. Systém musí být schopný komunikovat s různými typy řízení linek
2. Systém musí být schopný vyčítat a ukládat načtená data do databáze
3. Systém musí řídit linku na základě naučených dat pomocí samoučícího algoritmu

4.2 Non funkční požadavky

Vývoj systému probíhá v nástroji Visual Studio 2019 ve verzi Profesional na počítači s operačním systémem Windows 10 a lokálním databázovým serverem MS SQL verze Developer.
Pro běh aplikace je vyžadováno běhové prostředí .NET Core 3.1.
Komunikace mezi simulací a řízením probíhá pomocí průmyslového protokolu OPC.UA. OPC server pro zprostředkování komunikace se zařízeními zajišťuje OPC Server KEPServerEX společnosti KepWare ve verzi 6.8.

4.3 Návrh architektury

Systém je vhodné rozdělit na více vrstev, které umožní značnou míru abstrakce. Jelikož existuje velké množství řídících systému výrobních linek, je vhodné mít pro komunikaci s řízeními prostředníka v podobě OPC serveru, který se postará o rozdíly v komunikaci mezi jednotlivými výrobci a typy PLC řídících systémů. Další vrstva navrhovaného systému tak bude komunikovat standartním OPC UA protokolem pouze s OPC serverem. Pro tento projekt byl zvolen OPC server KepServerEx společnosti Kepware.



Obrázek 10 – Návrh architektury, vlastní zpracování

4.3.1 Vrstva – průmyslové počítače

Vrstva v návrhu architektury reprezentuje všechny řídící systémy, ze kterých bude vyčítáno, nebo případně do nich bude zapisováno. Tato zařízení mají vlastní program a logiku, díky které mají přehled například o počtu vyrobených kusů, prodlevám mezi operacemi, problémech a chybách, ve kterých se stanice nachází. Všechna tato data lze kategorizovat do tří úrovní konektivity.

1 Přehled o počtu dobrých i špatných vyrobených kusů
2 Chybové stavy, ve kterých se stanice nachází
3 Procesní data – veškeré další hodnoty naměřené při procesu výroby

Tato zařízení jsou připojena do sítě ethernet.
V rámci této diplomové práce autor vrstvu průmyslových počítačů nahradí simulací.

4.3.2 Vrstva OPC Server

Tato vrstva reprezentuje software třetí strany, který zajistí komunikaci s průmyslovými počítači, připojených do sítě ethernet. Výběr serveru záleží na rozmanitosti výrobců používaných průmyslových počítačů. Autor využívá OPC server společnosti KepWare, která implementuje většinu používaných průmyslových počítačů.

4.3.3 Vrstva vlastní SW

Software popisovaný v rámci této práce Autor rozdělil na tři moduly.

4.3.3.1 Modul OPC UA komunikace

Modul komunikace implementuje použití protokolu OPC UA, který je průmyslovým standardem. Modul nabízí objekty, které díky zapouzdření umožní jednoduše využít komunikaci s OPC serverem dalším modulům navrhovaného softwaru.

4.3.3.2 Modul sběru dat

Tento modul zajistí vyčítání dat z OPC serveru a data uloží do databáze. Zároveň uchovává poslední hodnoty v paměti programu a poskytuje je tak modulu AI řízení, který na jejich základě predikuje nové nastavení linky.

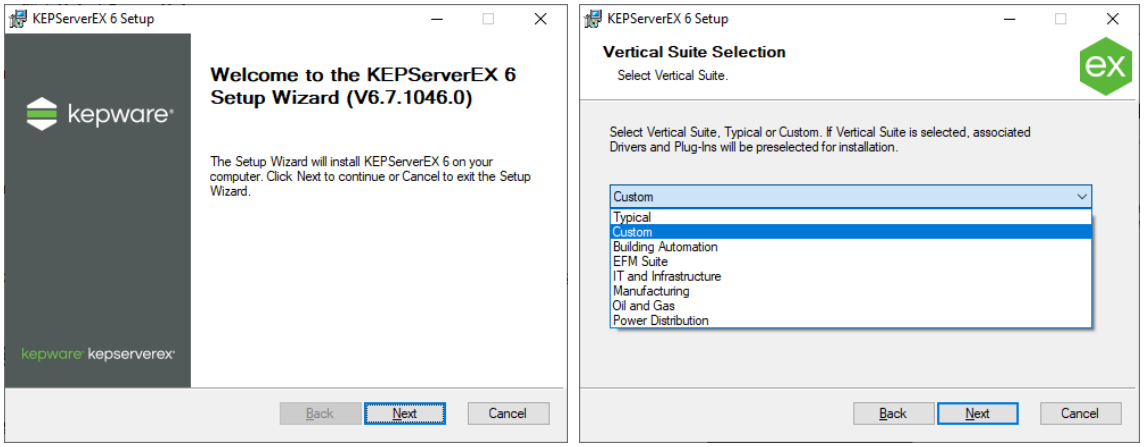
4.3.3.3 Modul AI řízení

Modul si na základě již uložených dat vytrénuje model predikce nastavení akčních členů linky, bude sledovat výrobní parametry a v reálném čase bude nastavovat akční členy, dle vytrénovaného modelu.

4.4 Konfigurace OPC serveru

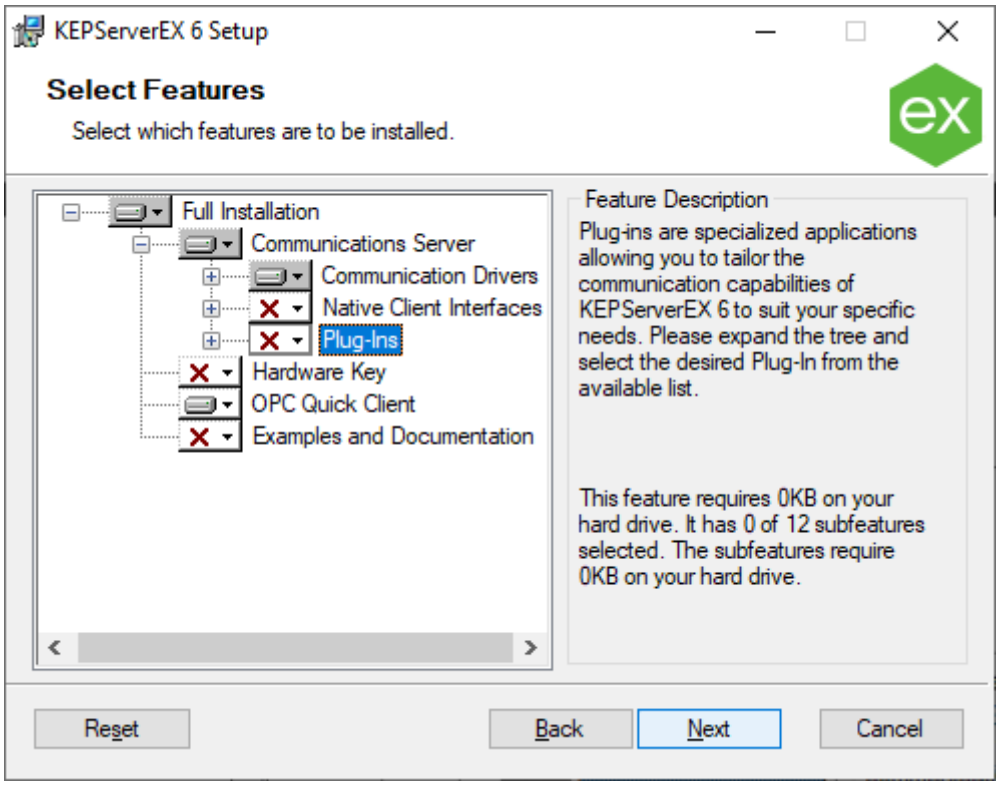
KepServerEX je dostupný online z webu společnosti KepWare jako exe[7] instalační balíček. Po spuštění instalace jsem vyzváni k výběru obsahu a pluginů. Autor vybere vlastní instalaci. Viz následující obrázky.

[7] Spustitelný soubor v prostředí Windows



Obrázek 11 – Instalace KepServeruEX, vlastní zpracování

Následně je uživatel vyzván k vybrání modulů a ovladačů, které si žádá nainstalovat. Autor v tomto případě nezvolí žádný modul a v ovladačích pouze Siemens Simatic a ovladače pro simulace. Viz následující obrázek.



Obrázek 12 – Výběr modulů a ovladačů, vlastní zpracování

Po zadání dalších nezbytných údajů, jako jsou cesty aplikace a umístění pro data je KepServerEx nainstalován. Zda je server spuštěn lze zjistit ve službách, kde běh indikuje služba KEPServerEX 6.8 Runtime.

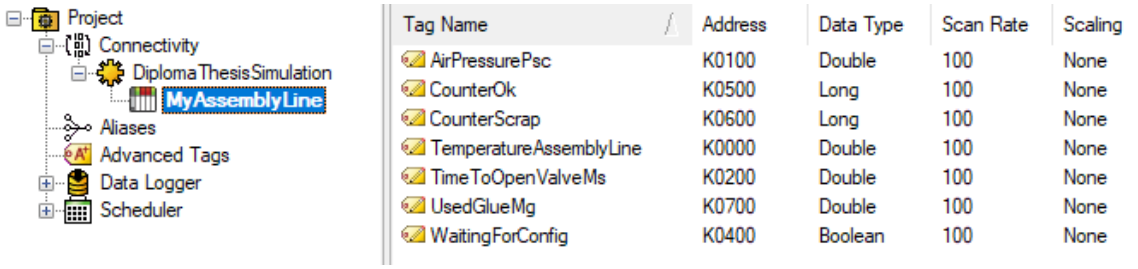
4.4.1 Nastavení pro simulaci

V rámci OPC serveru je nutné připravit prostředí pro testovací účely. KepServer umožňuje krom přímého napojení řídících průmyslových počítačů PLC také virtuální zařízení pro účely simulace.

V projektu v části `Connectivity` autor přidá nový kanál typu, `Simulator`, s názvem `DiplomaThesisSimulation`.

Do kanálu autor přidá nové zařízení s názvem `MyAssemblyLine`. Tyto názvy kanálu a zařízení jsou později použity v rámci URI pro přístup k registrům zařízení, v tomto případě virtuálnímu zařízení.

U zařízení jsou následně přidány proměnné, které budou využity v rámci komunikace mezi simulací linky a řízením.



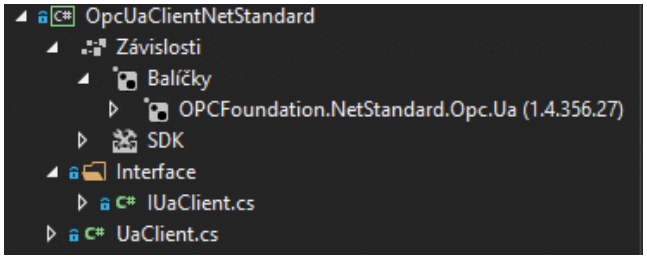
Obrázek 13 – Příprava proměnných na OPC Severu, vlastní zpracování

Proměnné musí být definovány s přístupem `read/write`. Aby do nich bylo pomocí OPC klientů možné i zapisovat.

4.5 Pomocná OPC knihovna

Autor v rámci projektu vytvoří knihovnu `OpcUaClientNetStandard` jejímž výstupem je dll[8]. Knihovna vzniká za účelem zjednodušení implementace protokolu OPC UA a rozšiřuje tak oficiálně dostupnou knihovnu `OPCFoundation.NetStandard.Opc.Ua` [20]. Knihovna cílí na architekturu .Net Standard a je tak kompatibilní s projekty .NET Core i .Net Framework.

[8] Dynamic-link library (Dynamicky linkovaná knihovna)



Obrázek 14 – Struktura a závislosti knihovny, vlastní zpracování

Knihovna má za cíl otevřít spojení s OPC serverem, přidat hodnoty ke sledování a při jejich změně vyvolat událost, zapisovat hodnoty na OPC server, vyčistit paměť a ukončit spojení.

```
namespace OpcUaClientNetStandard.Interface
{
    public interface IUaClient : IDisposable
    {
        void Start();
        MonitoredItem AddItem(string name, string path);
        ResponseHeader Write<T>(
             MonitoredItem item,
             T val,
             out StatusCodeCollection results,
             out DiagnosticInfoCollection diagnosticInfos);
        void RemoveItem(string name);
    }
}
```

Ukázka kódu 21 – Interface IUaClient, vlastní zpracování

Interface `IUaClient` rozšiřuje interface `IDisposable`, který zajistí veřejnou metodu `void Dispose()`. Tato metoda implementuje řádné ukončení spojení s OPC serverem.

```
public MonitoredItem AddItem(string name, string path)
{
    if (!_subscriptions.Any(x => x.MonitoredItemCount < ItemsPerSubscription))
    {
        _subscriptions.Add(
             new Subscription(_session.DefaultSubscription)
             { PublishingInterval = 100 });
        if (IsRunning)
            _subscriptions
                    .First(x => x.MonitoredItemCount < ItemsPerSubscription)
                    .Create();
    }
    var subscription =
       _subscriptions.First(x => x.MonitoredItemCount < ItemsPerSubscription);

    var item = new MonitoredItem(subscription.DefaultItem)
    {
        DisplayName = name,
        StartNodeId = path
    };
    if (MonitoredItems.Any(x => x.StartNodeId == item.StartNodeId))
    {
        return MonitoredItems.First(x => x.StartNodeId == item.StartNodeId);
    }
    item.Notification += ItemOnNotification;
    subscription.AddItem(item);
    if (IsRunning)
        subscription.ApplyChanges();

    return item;
}
```

Ukázka kódu 22 – Metoda pro přidání položky ke sledování, vlastní zpracování

Metoda `AddItem()` slouží pro přidání položky do objektu `subscription`, který slouží jako balík pro x položek a data tak odesílá v rámci jednoho socketu. Díky vysoké míře zapouzdření nemusí vyšší vrstvy tyto objekty řešit. Metoda zařídí založení a použití objektu `subscription`. Metoda rovněž řeší duplicity, a pokud zjistí požadavek na přidání již sledující hodnoty, vrátí již existující referenci. Objekt `UaClient` si udržuje informaci o vnitřním stavu a na základě něj v případě, že je spojení již aktivní metoda, zavolá nezbytné rutiny pro aktualizaci objektu `subscription`.

```
public void RemoveItem(string name)
{
    foreach (var subscription in _subscriptions)
    {
        var item =
             subscription.MonitoredItems
             .FirstOrDefault(x => x.DisplayName == name);
        if (item == null)
            continue;
        subscription.RemoveItem(item);
        if (IsRunning)
            subscription.ApplyChanges();
    }
}
```

Ukázka kódu 23 – Metoda pro zrušení sledování položky, vlastní zpracování

Metoda `RemoveItem()` zajistí odebrání sledované položky a v případě již běžícího spojení zavolá nezbytné rutiny pro aktualizaci objektu `subscription`.

```
public ResponseHeader Write<T>(MonitoredItem item,
    T val,
    out StatusCodeCollection results,
    out DiagnosticInfoCollection diagnosticInfos)
{
    var writeValue = new WriteValue
    {
        NodeId = (NodeId)item.ResolvedNodeId,
        AttributeId = Attributes.Value,
        IndexRange = null,
        Value = new DataValue()
        {
            Value = val
        }
    };
    return _session.Write(
        null,
        new WriteValueCollection()
            { writeValue },
        out results,
        out diagnosticInfos);
}
```

Ukázka kódu 24 – Metoda pro zápis nové hodnoty, vlastní zpracování

Metoda `Write<T>` je generická a umožňuje tak zápis hodnoty jakéhokoliv datového typu. Metoda zařídí zápis nové hodnoty a vrací výsledek požadavku.

Třída `UaClient` dále obsahuje privátní metody zajišťující udržování spojení a jeho obnovení při výpadku. Metodu pro správné připravení objektu a konfiguraci spojení. Třída dále udržuje informace o sledovaných položkách v rámci jednotlivých `subscriptions`.

4.6 Simulace výrobní linky

Simulace je implementována jako konzolová aplikace, která slouží k simulaci průmyslového počítače výrobní linky. Komunikace s aplikací řízení a sběru probíhá přes OPC server. Simulace odráží reálný scénář procesu lepení komponent, kde hlavním sledovaným parametrem je množství aplikovaného lepidla. Tento parametr je rozhodující pro výsledek a vyhodnocení, zda byl vyrobený kus v pořádku.
V rámci simulace jsou v hlavní roli následující parametry:

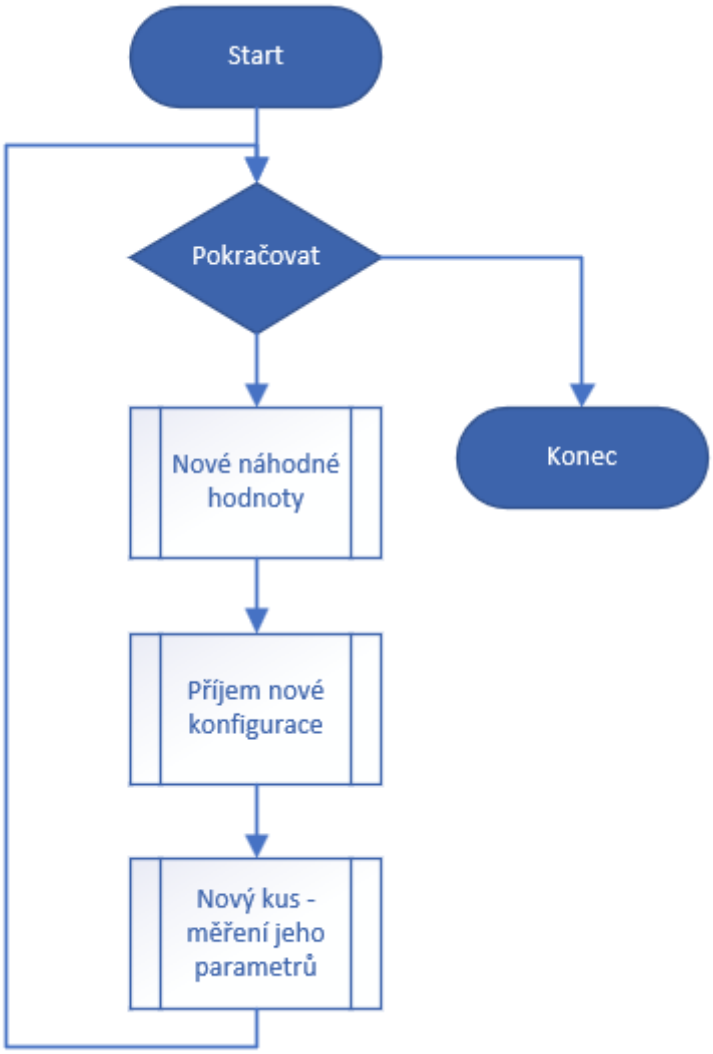
- Teplota prostředí
- Tlak vzduchu tlačícího na píst pro aplikaci lepidla
- Doba otevření ventilu pro aplikaci lepidla

Teplota a tlak jsou proměnné prostředí, které je v rámci simulace s každým cyklem náhodně změněno pro demonstraci reakce učícího se algoritmu. Doba otevření je předmětem predikce a slouží tedy jako uživatelský vstup pro korekci aplikovaného množství lepidla.

```
GlueFlowMgPerMs = Math.Pow(10 * Temperature, -1) + Math.Pow(AirPressurePsc, 2);
UsedGlueMg = GlueFlowMgPerMs * TimeToOpenMs;
```

Ukázka kódu 25 – Výpočet aplikovaného množství lepidla, vlastní zpracování

Výpočet naneseného množství používá přepočet množství za milisekundu, do kterého vstupují náhodné parametry prostřední. Použité množství pak odráží uživatelský vstup na dobu otevření ventilu.
Algoritmus celé aplikace simulace je zobrazen v následujícím obrázku.



Obrázek 15 – Algoritmus simulace, vlastní zpracování

Autor následně popíše implementaci použitých metod.

Metoda `CurrentValues()` provede náhodnou úpravu parametrů teploty a tlaku. Po vygenerování nových hodnot, jsou hodnoty odeslány na OPC server ke zpracování řídícím systémem.

Metoda `Calibrate()` zařídí vyčkání na načtení nové konfigurace z OPC serveru.

Metoda `NextPiece()` provede úkon aplikování lepidla a vyhodnotí výsledek. Podle výsledku zvýší odpovídající počítadlo kusů – dobrých, nebo špatných. Počítadlo a vypočítané hodnoty jsou odeslané na OPC server.

```
static void Main(string[] args)
{
    var keepRunning = true;
    Console.CancelKeyPress +=
      delegate (object sender, ConsoleCancelEventArgs e) {
        e.Cancel = true;
        keepRunning = false;
    };
    using var line = new AssemblyLine();
    while (keepRunning)
    {
        line.CurrentValues();
        line.Calibrate();
        line.NextPiece();
        Thread.Sleep(500);
    }
}
```

Ukázka kódu 26 – Program simulace, vlastní zpracování

Výstup vypisuje náhodně vygenerované hodnoty pro simulaci prostředí, načtenou novou hodnotu času otevření ventilu a výsledek výpočtu aplikace lepidla, a tedy i vyhodnocení dobrého, nebo špatného kusu.

4.7 Hlavní aplikace

Hlavní aplikace je realizovaná formou webové aplikace postavené na technologii .NET Core. Cílem tohoto projektu je obsluha modulů sběru dat a AI řízení linky. Webová služba uživateli poskytuje zpětnou vazbu o dění uvnitř modulů, které jsou popsané v další části práce.

Autor pro tvorbu webového rozhraní využil experimentální knihovnu Blazor, která rozšiřuje standartní webový framework a využívá moderní standard WebAssembly.

4.7.1 Pomocné služby pro uchování dat v paměti

V rámci programu je potřeba uchovat některá data v paměti programu. Autor definuje dvě pomocné služby:

- AssemblyLineService
- LogService

Obě služby budou později využity v rámci modulů i uživatelského rozhraní.

První zmíněná služba slouží k uložení aktuálního stavu výrobní linky, respektive hodnot načtených z OPC Serveru. Druhou její funkcí je uložení posledních 20

načtených hodnot naměřených k jednomu vyrobenému kusu. Implementace služby viz následující ukázka kódu.

```
public class AssemblyLineService
{
    public AssemblyLine AssemblyLine { get; set; }
    private IList<AssemblyLineData> _data =
              new List<AssemblyLineData>(20);

    public AssemblyLineService()
    {
        AssemblyLine = new AssemblyLine();
    }
    public void AddData(AssemblyLineData data)
    {
        if (_data.Count == 20)
            _data.RemoveAt(0);
        _data.Add(data);
    }
    public IList<AssemblyLineData> GetLastData()
    {
        return _data;
    }
}
```

Ukázka kódu 27 – Implementace pomocné služby pro uchování načtených dat v paměti, vlastní zpracování

Služba je registrovaná jako singleton[9] objekt. Pro řízení závislostí autor využívá vestavěného Dependenci Injection kontejneru. Instance objektu pro přístup k databázi je vytvořena již při konfiguraci služeb, definované ve třídě `Startup`, v metodě `ConfigureServices()` viz následující ukázka kódu.

```
services.AddSingleton<AssemblyLineService>();
```

Ukázka kódu 28 – Registrace singleton objektu, vlastní zpracování

Druhá služba slouží k logování událostí běhu programu. Je registrovaná stejným způsobem jako služba stavu linky a načtených dat, pomocí singleton vzoru. Implementace služby viz následující ukázka kódu.

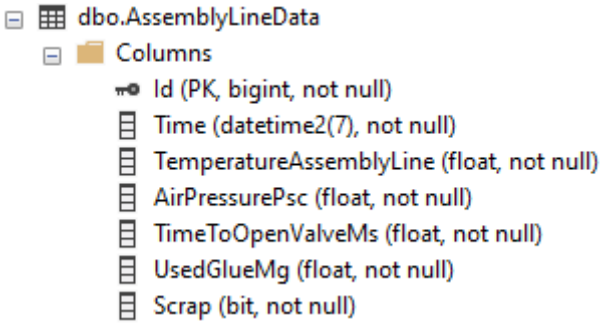
[9] Návrhový vzor zajišťující vznik pouze jedné instance

```
public class LogService
{
    private readonly IList<Log> _events = new List<Log>(10);

    public void AddEvent(Log log)
    {
        if(_events.Count>10)
            _events.RemoveAt(0);
        _events.Add(log);
    }

    public void AddEvent(string source, string text)
    {
        AddEvent(new Log(source,text));
    }

    public async Task<IList<Log>> GetLogsAsync()
    {
        return await Task.Run(() => _events);
    }
}
```

Ukázka kódu 29 – Implementace pomocné služby pro logování, vlastní zpracování

4.7.2 Modul sběru dat

Autor pro sběr dat využívá databázový server MS SQL. Veškerá načtená data z výrobní linky jsou průběžně ukládána do tabulky `AssemblyLineData`. Strukturu popisuje následující obrázek.



Obrázek 16 – Struktura tabulky sběru dat, vlastní zpracování

Autor využívá ORM framework `EntityFrameworkCore` a tedy v projektu existuje stejnojmenná třída obsahující výše popsané atributy. Autor zvolil způsob mapování „*code first*“ a databáze je tedy generovaná ORM frameworkem pomocí nástrojů EFCore.Tools.

Přístup k databázi je realizován pomocí Singleton objektu `ApplicationDbContext`.

```
services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(
        Configuration.GetConnectionString("DefaultConnection")));
```

Ukázka kódu 30 – Registrace objektu pro přístup k databázi, vlastní zpracování

Objektu je z konfiguračního souboru předán připojovací řetězec k databázi. Tento řetězec je umístěn v souboru `appsettings.json`.

Pro sběr dat je nezbytné se připojit na server KepWare pomocí průmyslového protokolu OPC.UA. Autor využije pomocnou knihovnu popsanou v rámci předchozí kapitoly – 4.5 Pomocná OPC Knihovna. Přístup k serveru, je realizován pomocí objektu `UaClient`, který autor zaregistruje do DI kontejneru jako singleton, aby zajistil inicializaci pouze jedné instance a realizaci jednoho spojení. Registrace probíhá v metodě `ConfigureServices()` způsobem popsaným v následující ukázce.

```
services.AddSingleton<UaClient>(new UaClient(
    Configuration.GetValue<string>("OpcUaClient:EndpointAddress"),
    Configuration.GetValue<string>("OpcUaClient:InstanceName")));
```

Ukázka kódu 31 – Registrace singleton objektu UaClient, vlastní zpracování

V rámci ukázky kódu je založena instance třídy `UaClient`, které je předána konfigurace spojení, která je uložena v souboru `appsettings.json`.

Modul sběru dat autor implementuje formou hostované služby, definované v třídě `DataReadingService`. Tato třída implementuje rozhraní `IHostedService`. To zařídí implementaci metody `StartAsync()` a `StopAsync()` nezbytné pro spuštění, respektive ukončení úlohy běžící na pozadí.

Služba je v rámci aplikace registrovaná v metodě `ConfigureServices` stejně jako objekt pro přístup k databázi.

```
services.AddHostedService<DataReadingService>();
```

Ukázka kódu 32 – Registrace služby běžící na pozadí, vlastní zpracování

V rámci služby `DataReadingService` je nezbytné od DI kontejneru vyžádat reference spravovaných objektů viz následující ukázka kódu.

```
private readonly IServiceProvider _serviceProvider;
private readonly LogService _logService;
private readonly AssemblyLineService _assemblyLineService;
private readonly UaClient _uaClient;
public DataReadingService(
    IServiceProvider serviceProvider,
    LogService logService,
    AssemblyLineService assemblyLineService,
    UaClient uaClient)
{
    _serviceProvider = serviceProvider;
    _logService = logService;
    _assemblyLineService = assemblyLineService;
    _uaClient = uaClient;
}
```

Ukázka kódu 33 – Konstruktor služby sběru dat a inject závislostí, vlastní zpracování

Následuje definování proměnných, které budou vyčítány z OPC serveru, uložení jejich reference a navázání spojení pomocí injektovaného objektu `UaClient`. Tyto akce proběhnou v asynchronní metodě `StartAsync()`.

```
private const string TemperaturePath =
"ns=2;s=DiplomaThesisSimulation.MyAssemblyLine.TemperatureAssemblyLine";

_temperature = _uaClient.AddItem("temperatureAssemblyLine", TemperaturePath);
_airPressurePsc = _uaClient.AddItem("airPressurePsc", AirPressurePscPath);
_timeToOpenValveMs =
       _uaClient.AddItem("timeToOpenValveMs", TimeToOpenValveMsPath);
_usedGlueMg = _uaClient.AddItem("usedGlueMg", UsedGlueMgPath);
_counterOk = _uaClient.AddItem("counterOk", CounterOkPath);
_counterScrap = _uaClient.AddItem("counterScrap", CounterScrapPath);
_waitingForConfig =
       _uaClient.AddItem("waitingForConfig", WaitingForConfigPath);
```

Ukázka kódu 34 – Definice proměnných k vyčítání z OPC, vlastní zpracování

Součástí metody `StartAsync()` je i definice obsluhy události při změně načtené hodnoty. Tato událost je zpracována metodou `UaClientOnItemChanged(object sender, EventArgs e)`. Implementaci metody autor popisuje v následující ukázce kódu.

```
private void UaClientOnItemChanged(object sender, EventArgs e)
{
    var item = (MonitoredItem)sender;
    if (item.DisplayName == _temperature.DisplayName)
    {
        foreach (var value in item.DequeueValues())
        {
            _assemblyLine.Temperature = Convert.ToSingle(value.ToString());
        }
    }
    else if(item.DisplayName == _airPressurePsc.DisplayName){…}
}
```

Ukázka kódu 35 – Obsluha události, změna sledované hodnoty, vlastní zpracování

Metoda příjme jako atribut objekt typu `MonitoredItem`, který identifikuje sledovanou proměnnou, u které došlo ke změně hodnoty. Autor v ukázce kódu uloží nejnovější získanou hodnotu do paměti programu.

Další volanou částí z metody `StartAsync()` je zavolání metod `StoreData()` a `UploadConfig()`, které spustí dvě obslužná vlákna, kde první je pro ukládání načtených dat do databáze a druhé pro odeslání konfigurace z paměti programu, kterou připravuje služba AI řízení linky, popsaná v další části této diplomové práce.

```
using var scope = _serviceProvider.CreateScope();
var applicationDbContext =
       scope.ServiceProvider.GetRequiredService<ApplicationDbContext>();
while (!cancellationToken.IsCancellationRequested)
{
    while (_dataToStore.TryDequeue(out var item))
    {
        _assemblyLineService.AddData(item);
        await applicationDbContext.AssemblyLineData
             .AddAsync(item, cancellationToken);
    }
    await applicationDbContext.SaveChangesAsync(cancellationToken);
    Thread.Sleep(1000);
}
```

Ukázka kódu 36 – Ukládání dat z paměti do databáze, vlastní zpracování

Data určená k uložení do databáze, jsou v průběhu běhu programu vkládána do fronty typu `ConcurrentQueue<AssemblyLineData>`. Tento datový typ je „*threadsafe*“[10]. Po uložení dat do databáze je vlákno na 1000ms uspáno.

Druhé pomocné vlákno pro nahrávání nové konfigurace využívá připravenou metodu `Write(MonitoredItem,T value)` třídy `UaClient`.

V případě zastavení programu je volaná metoda `StopAsync()`, která je implementovaná na základě rozhraní `IHostedService`.

```
Task StopAsync(CancellationToken cancellationToken)
{
       _uaClient.Dispose();
        return Task.CompletedTask;
}
```

Ukázka kódu 37 – Implementace ukončovací metody, vlastní zpracování

Účelem popsané metody je korektní uvolnění zdrojů, v tomto případě uzavření spojení s OPC serverem.

[10] Umožňuje bezpečný přístup a manipulaci s pamětí programu z více vláken

4.7.3 Modul AI řízení linky

Pro práci s modelem umělé inteligence autor definuje třídu `ApplicationML`. Třída obsahuje referenci na objekt `MLContext`, která je vytvořena v rámci konstruktoru.

```
private readonly MLContext _mlContext;
private ITransformer _model;

public ApplicationML()
{
    _mlContext = new MLContext(seed: 0);
}
```

Ukázka kódu 38 – Konstruktor třídy vlastního kontextu ML, vlastní zpracování

Třída obsahuje referenci na vytrénovaný model, která bude vytvořena v rámci metody `Task LoadAndTrainModel()` třídy `ApplicationML`. Implementace metody je znázorněna v následující ukázce kódu.

```
public async Task LoadAndTrainModel(ApplicationDbContext dbContext)
{
    var list = await dbContext.AssemblyLineData
        .Where(x => !x.Scrap)
        .ToListAsync();

    IDataView dataView =
        _mlContext.Data.LoadFromEnumerable<AssemblyLineData>(list);

    var pipeline = _mlContext.Transforms.CopyColumns(
            outputColumnName: "Label",
            inputColumnName: "FTimeToOpenValveMs")
        .Append(_mlContext.Transforms.Concatenate(
            "Features",
            "FTemperatureAssemblyLine",
            "FAirPressurePsc"))
        .Append(_mlContext.Regression.Trainers.FastTreeTweedie());

    _model = pipeline.Fit(dataView);
}
```

Ukázka kódu 39 – Trénování modelu, vlastní zpracování

V první části metody `LoadAndTrainModel()` je načtení vzorových dat z databáze, bez nepovedených záznamů. Následuje vytvoření datasetu relevantních dat. Třída `AssemblyLineData` obsahuje všechny nezbytné atributy pro vytvoření modelu. Atributy, které jsou nerelevantní pro ML model jsou označeny anotací `NoColumn`. Označení nerelevantních atributů viz následující ukázka kódu.

```
public class AssemblyLineData
{
    [Key, NoColumn]
    public long Id { get; set; }
    [NoColumn]
    public DateTime Time { get; set; } = DateTime.Now;
    …
}
```

Ukázka kódu 40 – Označení nerelevantních atributů pro ML, vlastní zpracování

Po vytvoření datasetu následuje vytvoření řetězce estimatorů[11], které definují postup pro práci s daty a návrh modelu. Do tohoto řetězce jsou následně vložena data z připraveného datasetu pomocí metody `Fit()`. Tato operace se nazývá trénování modelu a v závislosti na velikosti datasetu se zvyšuje výpočetní, respektive časová náročnost.

V rámci třídy `ApplicationML` autor implementuje druhou metodu `PredictValue()`, pro použití vytrénovaného modelu a predikci nového nastavení linky s ohledem na vstupní parametry.

```
public float PredictValue(double temperature, double airPressure)
{
    var predictionFunction =
       _mlContext.Model
       .CreatePredictionEngine<AssemblyLineData, ModelOutput>(_model);
    var predictVal = new AssemblyLineData()
    {
        TemperatureAssemblyLine = temperature,
        AirPressurePsc = airPressure
    };
    var prediction = predictionFunction.Predict(predictVal);
    return prediction.FTimeToOpenValveMs;
}
```

Ukázka kódu 41 – Metoda pro predikci nové hodnoty, vlastní zpracování

Metoda přijímá dva vstupní parametry a vrací predikovanou hodnotu, typu `float`. Autor modul do systému implementuje jako úlohu, respektive službu běžící na pozadí, stejně jako modul sběru dat popsaný v předchozí kapitole. Úloha je reprezentována třídou `MachineLearningService`. Tato třída implementuje rozhraní `IHostedService`. Služba je v rámci aplikace registrovaná v metodě `ConfigureServices()` stejně jako služba sběru dat popsaná v předchozí kapitole.

[11] Objekt, který se učí pomocí dat

```
services.AddHostedService<MachineLearningService>();
```

Ukázka kódu 42 – Registrace služby řízení, vlastní zpracování

Třída obsahuje konstruktor, pomocí kterého autor přijme reference z DI kontejneru. Reference si uloží v rámci třídy do privátních atributů.

```
public class MachineLearningService : IHostedService
{
    private readonly IConfiguration _configuration;
    private readonly LogService _logService;
    private readonly IServiceProvider _serviceProvider;
    private readonly AssemblyLineService _assemblyLineService;

    private readonly ApplicationML _mlContext;
    public MachineLearningService(
        IConfiguration configuration,
        IServiceProvider serviceProvider,
        AssemblyLineService assemblyLineService,
        LogService logService)
    {
        _configuration = configuration;
        _serviceProvider = serviceProvider;
        _assemblyLineService = assemblyLineService;
        _logService = logService;

        _mlContext = new ApplicationML();
    }
    public async Task StartAsync(CancellationToken cancellationToken){
    …
    }
    private void DoAction(CancellationToken cancellationToken){
    …
    }
}
```

Ukázka kódu 43 – Služba AI řízení linky, vlastní zpracování

Následuje implementace metody `StartAsync()` viz následující ukázka kódu.

```
public async Task StartAsync(CancellationToken cancellationToken)
{
    _logService.AddEvent(nameof(MachineLearningService), "Starting");
    if (_configuration.GetValue<bool>("UseMlToPredict"))
    {
        using var serviceScope =
             _serviceProvider
             .GetRequiredService<IServiceScopeFactory>()
             .CreateScope();
        var context =
             serviceScope.ServiceProvider
             .GetService<ApplicationDbContext>();
        await _mlContext.LoadAndTrainModel(context);
    }
    DoAction(cancellationToken);
    _logService.AddEvent(nameof(MachineLearningService), "Started");
}
```

Ukázka kódu 44 – Implementace StartAsync() v AI řízení, vlastní zpracování

Metoda uloží informace inicializaci startu do paměti programu. Následně podle konfigurace v `appsettings.json` připraví přístup k databázi a předá jej objektu učení zavoláním dříve definované metody `LoadAndTrainModel()`, která zajistí, že si kontext stáhne data pro samoučící algoritmus a připraví model, který bude využit pro predikci nových hodnot. V další části zavolá proceduru `DoAction()`, která spustí vlákno pro řízení linky a predikování nového nastavení.

```
private void DoAction(CancellationToken cancellationToken)
{
    Task.Run(() =>
    {
        while (!cancellationToken.IsCancellationRequested)
        {
            if (_assemblyLineService.AssemblyLine.WaitingForConfig)
            {
                if (_configuration.GetValue<bool>("UseMlToPredict"))
                {
                    var prediction =
                           _mlContext
                            .PredictValue(
                                  _assemblyLineService
                                   .AssemblyLine.Temperature,
                                  _assemblyLineService
                                   .AssemblyLine.AirPressurePsc);
                    _assemblyLineService
                           .AssemblyLine.TimeToOpenMs = prediction;
                    _assemblyLineService.AssemblyLine.WriteNewConfig = true;
                }
                else
                {
                    CalibrateByAlgorithm();
                }
                Thread.Sleep(100);
            }
            else
                Thread.Sleep(100);
        }
    }, cancellationToken);
}
```

Ukázka kódu 45 – Spuštění vlákna pro obsluhu AI řízení, vlastní zpracování

Metoda spustí vlákno, které běží v konečném cyklu až do doby, kdy je pomocí tokenu předána informace k zastavení činnosti. Vlákno po 100ms kontroluje, zda je od něj vyžadována akce. Dále obsahuje rozhodovací logiku, zda se má použít AI řízení, nebo algoritmický výpočet. Ten je možný, protože autor zná vnitřní logiku simulace. Tento způsob lze použít pouze v tomto konkrétním případě a autor jej využil pro vytvoření správných dat, ze kterých se trénuje model. Metoda je implementovaná v následující ukázce kódu.

```
private void CalibrateByAlgorithm()
{
    var tmpGlueFlowMgPerMs =
       Math.Pow(10 * _assemblyLineService.AssemblyLine.Temperature, -1) +
       Math.Pow(_assemblyLineService.AssemblyLine.AirPressurePsc, 2);
    var tmpUsedGlueMg =
       tmpGlueFlowMgPerMs * _assemblyLineService.AssemblyLine.TimeToOpenMs;
    while (!(tmpUsedGlueMg > AssemblyLine.LowLevel &&
             tmpUsedGlueMg < AssemblyLine.TopLevel))
    {
        _assemblyLineService.AssemblyLine.TimeToOpenMs =
             (AssemblyLine.LowLevel +
             ((AssemblyLine.TopLevel - AssemblyLine.LowLevel) / 2)) /
             tmpGlueFlowMgPerMs;
        tmpUsedGlueMg =
             tmpGlueFlowMgPerMs *
             _assemblyLineService.AssemblyLine.TimeToOpenMs;
    }
    _assemblyLineService.AssemblyLine.WriteNewConfig = true;
}
```

Ukázka kódu 46 – Algoritmus pro přípravu učících dat, vlastní zpracování

4.7.4 Uživatelské rozhraní

Autor využívá webový výstup pro komunikaci s uživatelem. Uživatel nemá žádné vstupy, ale jsou mu pouze poskytovány informace o běhu programu. Uživatelské rozhraní je rozděleno na dvě části. První z nich je přehled o stavu linky, respektive simulace a její parametry. Druhá část je čtení logu o vnitřních stavech programu.

4.7.4.1 Stav linky a načtené hodnoty

Autor využívá experimentální knihovnu Blazor, která vývojáři umožní využít C# na straně klienta a využít tak třídy a služby definované v backend části programu. Stránka, respektive akce je definovaná razor souborem, který kombinuje html a C#.

```
@page "/"
@using System.Threading
@using BlazorApp.Data
@using BlazorApp.Models
@inject AssemblyLineService AssemblyLineService
@code
{
    private int _counterOk = 0;
    private int _counterScrap = 0;
    private double _temperature;
    private double _glueFlowMgPerMs;
    private double _usedGlueMg;
    private double _timeToOpenMs;
    private double _airPressurePsc;

    private IList<AssemblyLineData> _data = new List<AssemblyLineData>();

    private Timer _timer;
    protected override async Task OnInitializedAsync()
    {
        _timer = new Timer(async _ =>
        {
            await InvokeAsync(() =>
            {
                _counterOk = AssemblyLineService.AssemblyLine.CounterOk;
                _counterScrap = AssemblyLineService.AssemblyLine.CounterScrap;
                _temperature = AssemblyLineService.AssemblyLine.Temperature;
                _glueFlowMgPerMs =
                    AssemblyLineService.AssemblyLine.GlueFlowMgPerMs;
                _usedGlueMg = AssemblyLineService.AssemblyLine.UsedGlueMg;
                _timeToOpenMs = AssemblyLineService.AssemblyLine.TimeToOpenMs;
                _airPressurePsc =
                    AssemblyLineService.AssemblyLine.AirPressurePsc;

                _data = AssemblyLineService.GetLastData();
                this.StateHasChanged();
            });

        }, null, 1000, 1500);
    }
}
```

Ukázka kódu 47 – Využití C# v razor stránkách, vlastní zpracování

Na začátku souboru je definována cesta, respektive routování stránky. Následuje deklarace `using` a `inject` pro načtení závislostí z DI kontejneru. V sekci kód autor definuje proměnné, které přímo vypisuje v HTML, viz následující ukázka kódu. A v rámci metody `OnInitializedAsync()` definuje `timer`, který definované proměnné a list posledních 20 načtených hodnot v pravidelných intervalech 1500ms aktualizuje.

```
<label>Counter OK: @_counterOk</label><br />
<label>Counter Scrap: @_counterScrap</label><br />
<label>Temperature: @_temperature</label><br />
<label>Time to open valve: @_timeToOpenMs</label><br />
<label>Air pressure: @_airPressurePsc</label><br />
<label>Used glue (mg): @_usedGlueMg</label><br />
<table class="table">
    <thead>
        <tr>
            <th>Time</th>
            <th>Temperature</th>
            <th>Time to open valve</th>
            <th>Air pressure</th>
            <th>Used glue (mg)</th>
        </tr>
    </thead>
    <tbody>
        @foreach (var data in _data.OrderByDescending(x=>x.Time))
        {
            <tr class="@(data.Scrap?"text-danger":"text-success")">
                <td>@data.Time</td>
                <td>@data.TemperatureAssemblyLine</td>
                <td>@data.TimeToOpenValveMs</td>
                <td>@data.AirPressurePsc</td>
                <td>@data.UsedGlueMg</td>
            </tr>
        }
    </tbody>
</table>
```

Ukázka kódu 48 – Využitý C# proměnných v HTML, vlastní zpracování

Proměnné vypsané v tagu `<label>` jsou pravidelně aktualizovány při volání metody `this.StateHasChanged()`. Stejně tak je aktualizovaná tabulka, kterou autor vypisuje pomocí cyklu `foreach`.

4.7.4.2 Logování vnitřních stavů programu

Logy autor vypisuje podobným způsobem jako stav a data z linky. Je zde využita stránka `Logs.razor`. V rámci stránky jsou uživateli vypsány logy z paměti programu. Implementace viz následující ukázka kódu.

```
@page "/logs"
@using BlazorApp.Data
@using BlazorApp.Models
@inject LogService LogService
@code {
    private IList<Log> logs;
    protected override async Task OnInitializedAsync()
    {
        logs = await LogService.GetLogsAsync();
    }
}

<h1>Logs</h1>
@if (logs == null)
{
    <p><em>Loading...</em></p>
}
else
{
    <table class="table">
        <thead>
            <tr>
                <th>Time</th>
                <th>Source</th>
                <th>Text</th>
            </tr>
        </thead>
        <tbody>
            @foreach (var log in logs.OrderByDescending(x=>x.Time))
            {
            <tr>
                <td>@log.Time</td>
                <td>@log.Source</td>
                <td>@log.Text</td>

            </tr>
            }
        </tbody>
    </table>
}
```

Ukázka kódu 49 – Výpis logů, vlastní zpracování

V horní části autor opět definuje routování, using a inject závislostí. Následuje logika načtení dat ze služby a výpis v podobě HTML.

5 Shrnutí výsledků

5.1 Zhodnocení splnění požadavků z analýzy

Vytvořený systém splňuje případy použití a požadavky analýzy. Systém je schopný sbírat data a ukládat je do databáze. Tato data jsou navíc sbírána přes OPC server, pomocí standartního průmyslového protokolu OPC.UA. Tím je systém odstíněn od konkrétních typů zařízení a zodpovědnost implementace komunikace pro každý typ byla přesunuta na daný server, v tomto případě OPC KepServer společnosti Kepware.

Nasbíraná data jsou při každém spuštění systému využita pro vytrénování machine learning modelu pro predikci nového nastavení linky.

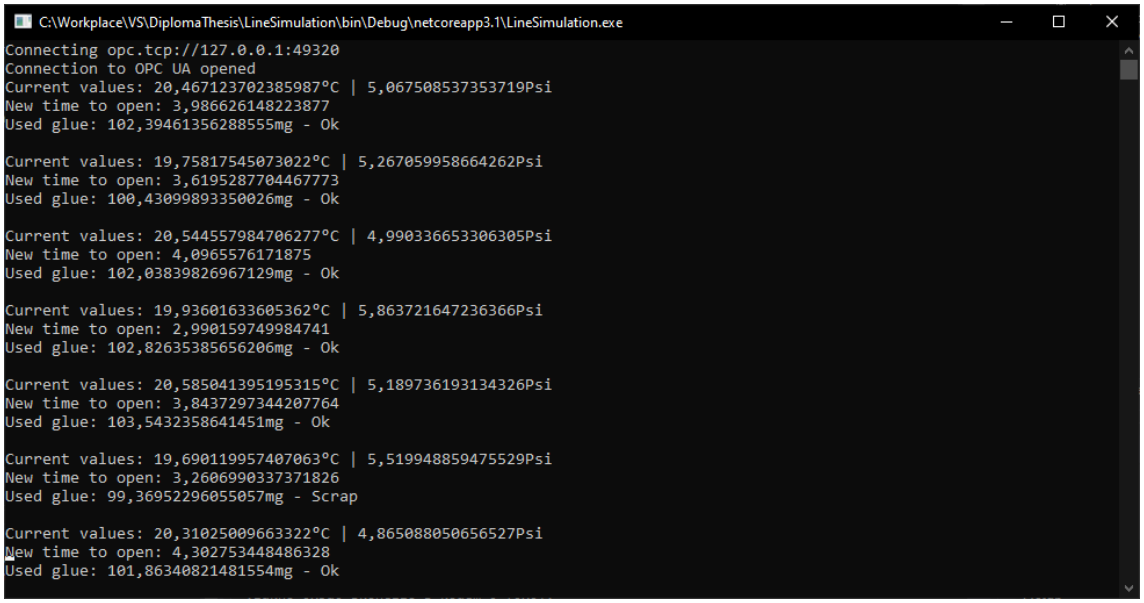
V rámci práce byla navrhnuta simulace výrobní linky využívající proces lepení, kde hlavními parametry jsou teplota, množství aplikovaného lepidla a tlak v pístu tlačícího na lepidlo. Výsledkem je porovnávání aplikovaného množství lepidla s horní a dolní mezí.

Simulace a systém řízení, respektive sběru spolu komunikují přes OPC server. V rámci testování je AI prokazatelně využitelná pro korekci provozních parametrů linky v reálném čase.

5.2 Uživatelský popis aplikace

5.2.1 Simulace výrobní linky

Simulace běží v rámci konzolové aplikace a uživateli je poskytnut textový výstup do konzole o aktuálním dění. Aplikace vypisuje vygenerované hodnoty pro simulaci prostředí, načtenou novou hodnotu času otevření ventilu a výsledek výpočtu aplikace lepidla, a tedy i vyhodnocení dobrého, nebo špatného kusu. Výstup je znázorněn na následujícím obrázku.



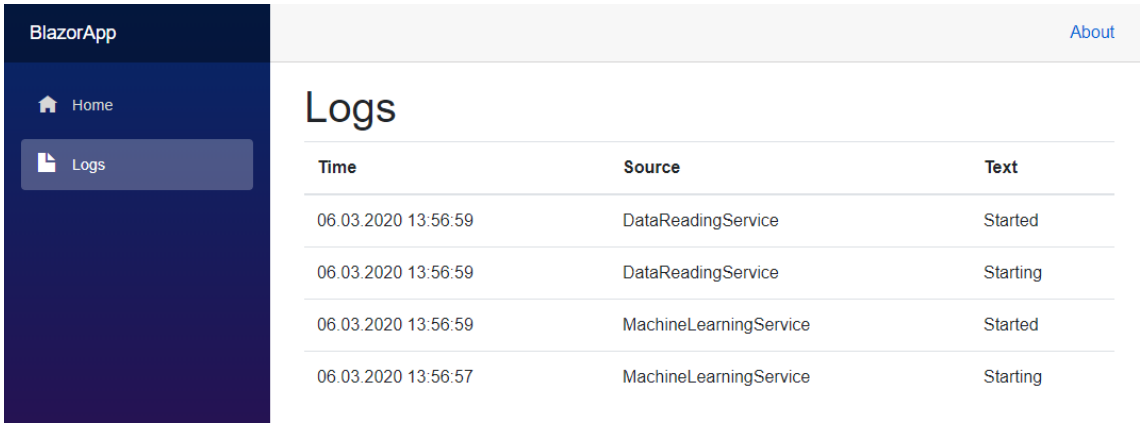
Obrázek 17 – Výstup aplikace simulace, vlastní zpracování

5.2.2 Systém sběru dat a autonomního řízení

Tato hlavní část systému je vytvořena jako webová aplikace v .NET Core, která na sebe váže služby běžící na pozadí.

Informace o běhu služeb jsou uživateli zobrazeny pomocí blazor stránek, tak aby měl přehled o vnitřním stavu systému a o načítaných datech, respektive úspěšnosti řídícího algoritmu.

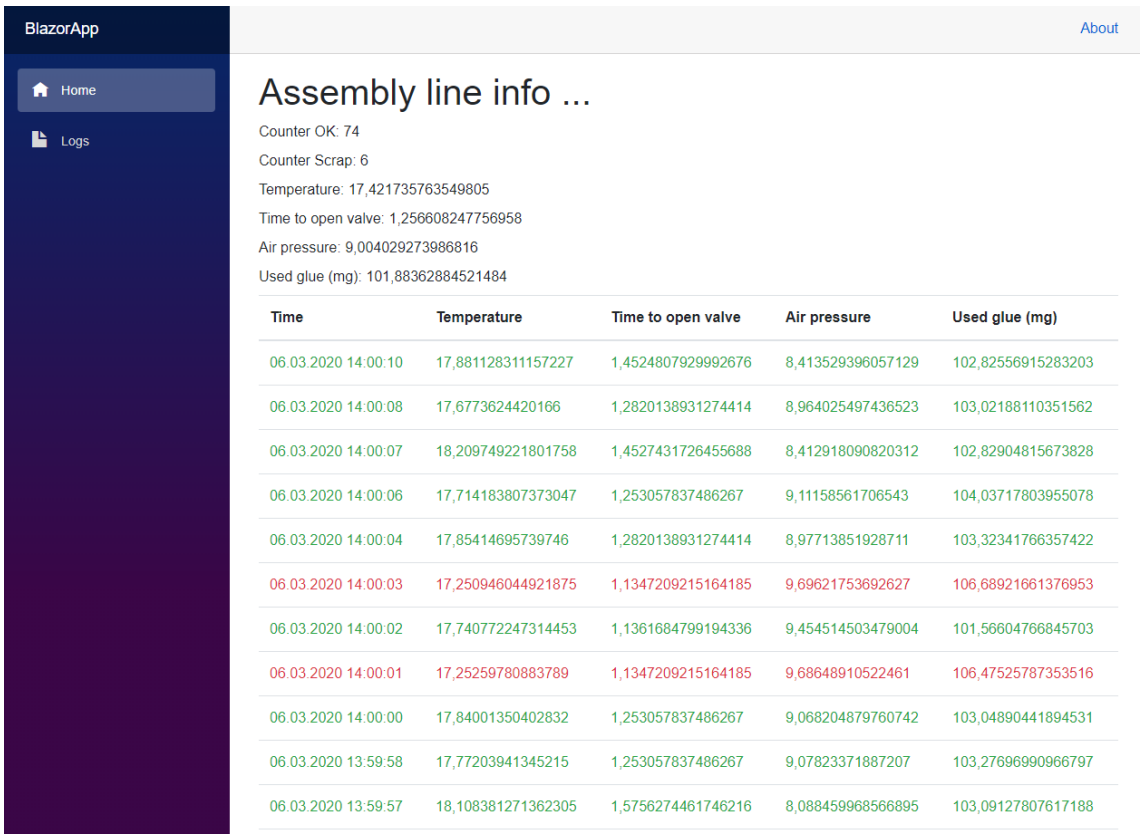
Výpis stavů systému viz následující obrázek.



Ukázka kódu 50 – Log vnitřních stavů systému, vlastní zpracování

Ve výpise má uživatel k dispozici informaci o jednotlivých modulech a jejich vnitřním stavu. U modulu `MachineLearningService` pak lze sledovat dobu startu a z něj odvodit náročnost trénování modelu.

Stránka s přehledem výrobní linky poskytuje uživateli náhled na aktuální stav parametrů, respektive simulace a výpis posledních naměřených hodnot spolu s výsledkem testování množství lepidla, respektive zda je množství v určených mezích. Náhled zobrazení viz následující obrázek.



Ukázka kódu 51 – Výpis parametrů výrobní linky a načtených dat, vlastní zpracování

Výsledky zjištěny v rámci experimentu viz následující tabulka.

Počet učících dat	Úspěšnost na vzorku 100ks	Průměrná úspěšnost
100	0.55, 0.60, 0.58, 0.70, 0.71	0.628
250	0.95, 0.49, 0.56, 0.83, 0.79	0.724
500	0.94, 0.96, 0.72, 0.77, 0.80	0.838
1000	1.00, 0.93, 0.93, 0.87, 0.95	0.936

Tabulka 2 – Výsledky měření úspěšnosti AI, vlastní zpracování

Testování probíhalo pro každý počet vstupních učících dat vždy v pěti opakováních, se stejnou vstupní sadou dat, ale novým spuštěním simulace, jejíž `seed` generátoru

náhodných čísel tak byl pozměněn, aby test pokryl co nejširší možný rozsah vstupů. Zjištěné výsledky byly zprůměrovány. Z výsledků je patrný rozdíl mezi úspěšnostmi v závislosti na počtu vstupních školících dat. Z tabulky je patrný nárůst úspěšnosti v závislosti na množství vstupních školících dat.

6 Závěry a doporučení

Cílem této práce bylo rámcové představení technologií .Net Core a Machine learning pro .NET, zároveň jejich aplikace a ověření využitelnosti v průmyslové automatizaci. Aby bylo možné technologie implementovat, bylo nutné nastudování teoretických předpokladů obou technologií, zejména technologie ML.NET. Technologie Machine learning nabízí spoustu možností a takřka neomezenou množinu scénářů použití. Autoři pro začínající vývojáře připravili grafický nástroj ModelBuilder popsaný v teoretické části této práce. Avšak tento nástroj, zatím ve verzi preview, se mi nepodařilo využít nad sadou mých dat kvůli vnitřní chybě nástroje. Chyba spočívala v tom, že jeden z regresních algoritmů, který se nástroj snažil využít nenaplnil očekávaný výsledek a havaroval. Nástroj ModelBuilder si s touto výjimkou neporadil a nenabídl ani možnost výběru ze zbylých algoritmů, které proběhly v pořádku. V rámci studování této technologie jsem tak byl nucen rovnou začít psát celý program svépomocí.

V rámci práce byl navrhnut systém řízení výrobní linky. Pro účely práce byla navrhnuta simulace, kde data byla distribuována přes OPC Server, tak aby byl scénář co nejblíže skutečnosti. V rámci simulace jsem se snažil přiblížit skutečnosti z reálného prostředí, výrobní linka aplikující určité množství lepidla na výrobek.

Simulace a hlavní systém byly úspěšně propojeny komunikací přes OPC Server a data ukládána do databáze. Tato data byla později využita pro trénování modelu pro autonomní řízení. Systém byl otestován při vzorku až 1000 vstupních dat pro trénování modelu a zjištěná úspěšnost predikce dosáhla 0.936. S vyšším množstvím vstupních dat se tato úspěšnost zvyšuje. Nutné je podotknout, že simulace byla záměrně tvořena tak, aby se vstupní parametry měnily rychle a v daleko větším rozsahu, než je tomu v reálném prostředí, právě pro účely testování samoučících algoritmů. V reálném prostředí může být přínos v podobě zvýšení produktivity, respektive snížení zmetkovitosti, protože reakční doba systému autonomního řízení je jeden výrobní cyklus, kdežto seřizovač na to potřebuje cyklů několik, a to pouze v případě, že se nachází na svém pracovišti.

Pokud bych měl zhodnotit, co mi tato práce přinesla, pak je to bezesporu úvod do technologií samoučících algoritmů a vzbudila můj zájem o tuto disciplínu. Následně

bych rád tuto technologii aplikoval na reálný příklad, kterým jsem se v rámci simulace inspiroval.

7 Seznam použité literatury

[1] B. Henderson-Sellers, A book of object-oriented knowledge: an introduction to object-oriented software engineering. 2nd ed., Upper Saddle River, NJ: Prentice Hall PTR, 1997, pp. 253, ISBN 0135688906.

[2] ECMA International, „Ecma international," Ecma, Prosinec 2017. [Online]. Available: https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-334.pdf. [Přístup získán 4 Září 2019].

[3] J. Liberty, Programming C#: Building. NET Applications with C#, Sebastopol: O'Reilly Media, Inc., 2005.

[4] Microsoft, „Průvodce platformou .NET Core," 2018. [Online]. Available: https://docs.microsoft.com/cs-cz/dotnet/core/. [Přístup získán 3 Září 2019].

[5] Microsoft, „Entity Framework Core," 2016. [Online]. Available: https://docs.microsoft.com/en-us/ef/core/. [Přístup získán 4 Září 2019].

[6] Expert System, „What is Machine Learning? A definition," 2019. [Online]. Available: https://www.expertsystem.com/machine-learning-definition/. [Přístup získán 4 Září 2019].

[7] Microsoft, „GitHub - dotnet/machinelearning," 2019. [Online]. Available: https://github.com/dotnet/machinelearning. [Přístup získán 4 Září 2019].

[8] Microsoft, „Machine learning tasks in ML.NET," 2019. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/machine-learning/resources/tasks. [Přístup získán 9 Listopad 2019].

[9] Microsoft, „What is ML.NET and how does it work?," 2019. [Online]. Available: https://docs.microsoft.com/en-US/dotnet/machine-learning/how-does-mldotnet-work. [Přístup získán 18 Září 2019].

[10] Microsoft, „DataOperationsCatalog Class," 1 Březen 2020. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/api/microsoft.ml.dataoperationscatalog?view=ml-dotnet. [Přístup získán 1 Březen 2020].

[11] Microsoft, „Prepare data for building a model,“ 2019. [Online]. Available: https://github.com/dotnet/docs/blob/live/docs/machine-learning/how-to-guides/prepare-data-ml-net.md. [Přístup získán 11 Listopad 2019].

[12] Microsoft, „ML.NET Model Builder | Machine learning in Visual Studio,“ 2019. [Online]. Available: https://dotnet.microsoft.com/apps/machinelearning-ai/ml-dotnet/model-builder. [Přístup získán 11 Listopad 2019].

[13] OPC Foundation, „History - OPC Foundation,“ OPC Foundation, 2020. [Online]. Available: https://opcfoundation.org/about/opc-foundation/history/. [Přístup získán 3 Březen 2020].

[14] OPC Foundation, „What is OPC?,“ 2019. [Online]. Available: https://opcfoundation.org/about/what-is-opc/. [Přístup získán 3 Září 2019].

[15] Foxon, „CO JE OPC? OPC SERVER? OPC KLIENT?,“ 2013. [Online]. Available: https://www.foxon.cz/blog/prakticka-teorie/159-co-je-opc-opc-server-opc-klient. [Přístup získán 3 Září 2019].

[16] OPC Foundation, „UA Overview,“ 2019. [Online]. Available: http://wiki.opcfoundation.org/index.php/UA_Overview. [Přístup získán 3 Září 2019].

[17] OPC DataHub, „What is OPC?,“ 2010. [Online]. Available: https://opcdatahub.com/WhatIsOPC.html. [Přístup získán 11 Listopad 2019].

[18] PTC Inc., „KepWare - Simulation Driver,“ 2019. [Online]. Available: https://www.kepware.com/getattachment/51b48fd3-28a8-43fd-96b7-5f3584329cb1/simulator-manual.pdf. [Přístup získán 11 Listopad 2019].

[19] T. Horvát, „Teoretický úvod do relačních databází,“ 11 Srpen 2007. [Online]. Available: http://programujte.com/clanek/2007110801-teoreticky-uvod-do-relacnich-databazi/. [Přístup získán 4 Září 2019].

[20] OPC Foundation, „OPCFoundation/UA.NETStandard,“ 7 Březen 2020. [Online]. Available: https://github.com/OPCFoundation/UA-.NETStandard. [Přístup získán 10 Březen 2020].

[21] Český Národní Korpus, „pojmy:ngram - Příručka ČNK,“ 2015. [Online]. Available: https://wiki.korpus.cz/doku.php/pojmy:ngram. [Přístup získán 11 Listopad 2019].

Univerzita Hradec Králové
Fakulta informatiky a managementu

Zadání diplomové práce

Autor:	**Bc. Jaroslav Langer**
Studium:	I1700322
Studijní program:	N1802 Aplikovaná informatika
Studijní obor:	Aplikovaná informatika
Název diplomové práce:	**Využití .NET Core v průmyslové automatizaci**
Název diplomové práce AJ:	Using .NET Core in industrial automation

Cíl, metody, literatura, předpoklady:

Cílem práce je popsat technologie .NET Core a ML.NET. Tyto technologii aplikovat na příkladu využitelném pro průmyslovou automatizaci. Součástí je navrhnout simulaci výrobní linky, systém sběru dat a autonomního řízení za využití technologie Machine Learning for .NET.

Cílem této diplomové práce je tak nejen rámcové představení technologie a teoretických postupů, ale převážně jejich praktické využití na projektu, který autor popisuje.

Osnova

- Úvod
- Cíl práce
- Teoretická východiska
- Praktická část
- Shrnutí výsledků
- Závěr literatura

Garantující pracoviště:	Katedra informatiky a kvantitativních metod, Fakulta informatiky a managementu
Vedoucí práce:	doc. Mgr. Tomáš Kozel, Ph.D.
Datum zadání závěrečné práce:	14.1.2018