УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ

Никола Сојчић

ПРИМЕНА ВЕБ ТРАГАЧА У ОБРАДИ И АНАЛИЗИ ЈАВНО ДОСТУПНИХ СПОРТСКИХ ПОДАТАКА

мастер рад

Београд, 2022.

Ментор:

др Александар Картељ, доцент
Универзитет у Београду, Математички факултет

Чланови комисије:

др Владимир Филиповић, редовни професор
Универзитет у Београду, Математички факултет

др Стефан Мишковић, доцент
Универзитет у Београду, Математички факултет

Датум одбране:

Наслов мастер рада: Примена веб трагача у обради и анализи јавно доступних спортских података

Резиме: Количина података која се генерише на спортским веб страницама је велика и самим тим њихова претрага и анализа представља изазов у контексту развоја одговарајућех софтвера. С обзиром да веб претраживач није погодан за аутоматско похрањивање и анализу података, а ручна веб претрага може бити дуготрајна и склона људским грешкама, као решење за прикупљање великог броја података намеће се Веб трагач. Веб трагач (енг. web crawler) игра важну улогу у прикупљању веб података. Веб трагач се може подесити тако да преузима и у ходу обрађује садржај са јавних веб локација, а информације се у сировој, делимично или потпуно обрађеној форми се могу даље похранити у циљном формату.

Циљ овог рада је развој Веб трагача за прикупљање спортских података са јавно доступних Веб сајтова и развој пропратне апликације за обраду прикупљених података са циљем предвиђања спортских резултата.

Кључне речи: веб трагач, класификација, спортски резултати, предвиђање

Садржај

Глава 1

Увод

Подаци доступни на спортским веб страницама се генеришу свакодневно и њихова количина, начин структуирања и даља анализа представљају изазове у контексту развоја одговарајућег софтвера. Веб претраживач може да пружи резултате претраге у току интерактивног претраживања од стране човека и није погодан за аутоматско похрањивање и даљу анализу података. Ручна веб претрага такође може бити дуготрајна и склона људским грешкама у уносу критеријума претраге као и чувању добијених резултата. Веб трагач (енгл. web crawler) игра важну улогу у прикупљању веб података. Веб трагач се може подесити тако да преузима и у ходу обрађује целокупни или делимични садржај са јавних веб локација, а информације се у сировој, делимично или потпуно обрађеној форми се могу даље похранити у циљном формату.

Циљ овог рада је развој веб трагача за прикупљање спортских података са јавно доступних веб сајтова и развој пропратне апликације за обраду прикупљених података са циљем предвиђања спортских резултата. Обе апликације су развијене у програмском језику Јава.

Веб трагач је програм чија је улога да претражује одређене интернет странице и прикупља податке који се касније могу обрађивати и користити за разне анализе.

Предвиђање спортских резултата — циљ апликације је да уз помоћ веб трагача прикупи спортске резултате фудбалских утакмица и да уз помоћ класификације покуша да одреди коначан исход утакмице (победа домаћина, нерешено, победа гостију).

1.1 Веб трагач

Веб трагач представља програм који омогућава аутоматско претраживање интернет страница, преузимање и анализирање одређених података. Постоји више врста трагања. На слици 1.1 можемо видети неке од врста трагања података на интернету који се могу поделити по типу података, а то су слике, новински чланци, видеи, разни текстови итд.

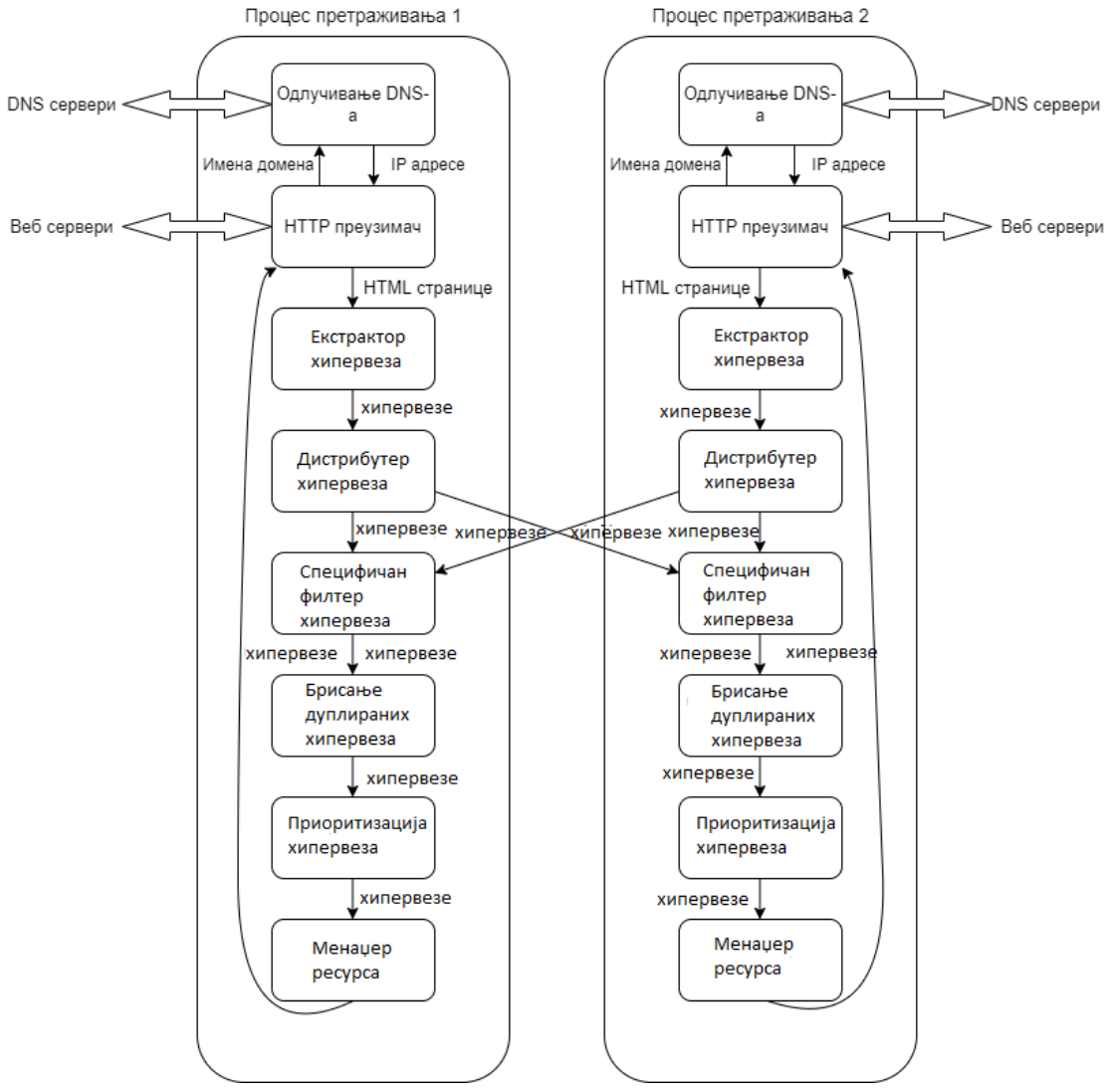
Слика 1.1: Типови трагања података на интернету

Основни алгоритам веб трагача ради на следећи начин. Веб трагач се састоји од већег броја процеса, а сваки процес се састоји од већег броја нити, где свака нит извршава поновљене циклусе. На слици 1.2 је приказана архитектура веб трагача која је описана у наставку текста.

На почетку сваког циклуса нит добија хипервезу (енгл. URL — Uniform Resource Locator) од структуре која се зове менаџер хипервеза (енгл. Frontier) и чија је улога да распоређује хипервезе према приоритету. Затим нит позива HTTP преузимач (енгл. HyperText Transfer Protocol fetcher) који позива DNS (енгл. Domain Name System) подмодул који преводи URL у одговарајућу IP

(енгл. Internet Protocol) адресу и затим се повезује на веб сервер одакле покушава да преузме веб страну.

Након успешног преузимања веб странице, она се прослеђује објекту под називом екстрактор хипервеза (енгл. Link extractor), који парсира HTML (енгл. Hyper Text Markup Language) садржај странице и који извлачи хипервезе који се налазе на тој страници. Извучени URL-ови се затим прослеђују URL дистрибутеру који сваком URL-у додели по један процес који ће обрађивати тај URL (Olson & Najork, 2010).



Слика 1.2: Архитектура веб трагача

Политика лепог понашања

Један од циљева рада веб трагача јесте да он буде ефикасан, а да и у истом тренутку не утиче на перформансе веб странице која се анализира. Да би успео у овоме сваки веб трагач има политику лепог понашања (енгл. politeness policy).

Веб трагачи могу да претражују и преузимају информације много брже него људи, али исто тако могу и да утичу лоше на перформансе веб странице. Цена употребе веб трагача подразумева:

- Мрежне ресурсе (трагачи могу да захтевају значајан проток);
- Преоптерећење сервера (ако је учесталост посећивања серверу превелика);
- Лоше имплементиране трагаче (трагачи који могу да наруше перформансе сервера или да преузму странице које не могу да обраде);
- Личне трагаче (трагачи који, ако су употребљени од стране више корисника, могу да поремете мрежу и веб сервер).

Веб трагачи би требало да се придржавају стандарду искључења робота (енгл. Robots Exclusion Standard), конвенцији која дозвољава администратору веб сајта да забрани веб трагачима приступ свим или само одређеном броју веб страница на тој веб адреси. То је имплементирано тако што се дода датотека изворног кода robots.txt од стране веб администратора. Ова датотека садржи скуп правила која одређују које веб странице се могу преузети од стране веб трагача. На овај начин се веб страница може одбранити од веб трагача и спречити да веб трагач утиче лоше на њене перформансе.

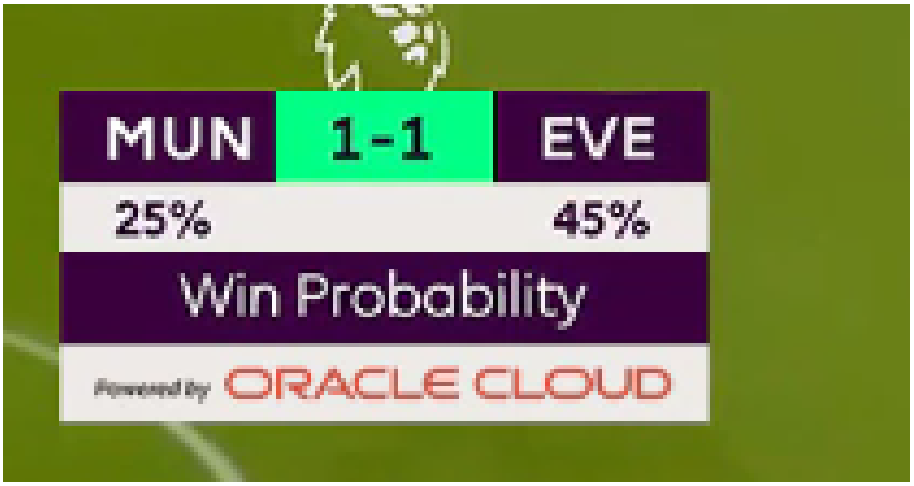
1.2 Предвиђање спортских резултата

Предвиђање будућности је веома популарно у данашње време, а кад се прича о спорту популарно је предвиђање спортских резултата.

Код фудбалских утакмица постоји доста игара које људи покушавају да предвиде, али игре које се најчешће играју су 3 могућа коначна исхода на једној утакмици, а то су победа домаће екипе, нерешен исход и победа гостујуће екипе.

У данашње време постоје разни сервиси који рачунају вероватноће за коначан исход током самог трајања меча и пример тога јесте сервис *Orange cloud* који за утакмице Премијер лиге на сваких 30 секунди рачуна вероватноће за коначан исход меча, које гледаоци могу видети током преноса уживо. На слици 1.3 може се видети приказ ових информација у току утакмице.

Циљ овог рада ће бити да се уз помоћ класификације и података са претходно одиграних утакмица две екипе, предвиди исход будућег меча.



Слика 1.3: Приказ вероватноћа коначног исхода утакмице Премијер лиге[1]

1.3 Класификација

Класификација представља једну од метода машинског учења где за циљ имамо да неком податку доделимо једну од више предефинисаних класа. Она заправо представља учење циљне функције, коју зовемо и модел класификације (Ajzenhajmer, Bukurov, & Stanković, 2017).

Свака техника класификације употребљава алгоритам за учење како би одредила модел који најбоље одговара вези између скупа атрибута и класе података. Модел који алгоритам генерише требало би да одговара улазним подацима као и да што тачније предвиђа класу података које раније није видео, односно да што боље *генерализује*.

Општи приступ решавању класификационог проблема јесте да се мора имати обезбеђен почетни тренинг скуп који се састоји од података чије су

[1]Може се пронаћи на `https://www.oracle.com/sg/premier-league`

класе познате. Након тренинга, квалитет добијеног модела се утврђује применом над тест подацима. Улога тест података је да симулирају будуће непознате податке и њихове придружене класе. Тест подаци се не смеју ни на који начин укључити у процес тренирања, чак ни имплицитно.

Примена метода класификације је веома широка и неки од примера где се може применити јесте у медицини за класификацију тумора (бенигни или малигни) на основу резултата магнетне резонанце, откривање да ли је електронска пошта спам на основу заглавља поруке и њеног садржаја, препознавање лица и гласа, за откривање превара са кредитним картицама, класификација штетног софтвера (енгл. malware) итд.

Глава 2

Архитектура система

Апликација која је развијена за потребе овог рада је веб апликација која се састоји из 2 дела (серверског и клијентског). Серверски део апликације је написан у програмском језику Јава и коришћен је развојни оквир отвореног кода *Spring Boot*, док је клијентски део апликације написан као Angular апликација. Комуникација ове две целине се одвија тако што се са клијентског дела позива REST API који се налази на серверском делу и који служе да се довезу подаци из базе података. База података је релациона, а систем за управљање базом података је MySQL.

2.1 Клијентски део апликације

Клијентски део је написан као једностранична Angular апликација (енгл. single-page application) преко које корисник може интерактивно да види списак утакмица које треба да се одиграју и предвиђени исход меча.

Данас Angular представља један од најпознатијих и најкоришћенијих развојних оквира отвореног кода (енгл. open-source) који је развијен од стране Google компаније. Неки од разлога због чега је Angular погодан за креирање динамичких веб страница су:

- Google подршка — Google омогућава Long-Term-Support (LTS) језику Angular и самим тим не постоји брига да Google неће одржавати и унапређивати Angular и у будућности (*8 Proven Reasons You Need Angular for Your Next Development Project*, n.d.).

- Typescript — Angular апликације су реализоване у језику Typescript, скрипт језику који подржава типове (примитивне и интерфејсе (објектне)).
- POJO — уз Angular није потребно декларисати методе које служе за довлачење и постављање објеката (енгл. getter и setter) додатно пошто сваки објекат који Angular користи јесте POJO (енгл. Plain Old JavaScript Object).
- Структура модула — Angular апликација може бити подељена по модулима где сваки модул представља једну целину. Оваква структура олакшава функционалност апликације и омогућава могућност поновне употребе одређеног кода.
- Лењо учитавање (енгл. Lazy loading) — поменута модуларна архитектура такође омогућава да се дохватају само модули који су потребни у том тренутку што омогућава да апликација има добре перформансе.

Управљање стањем

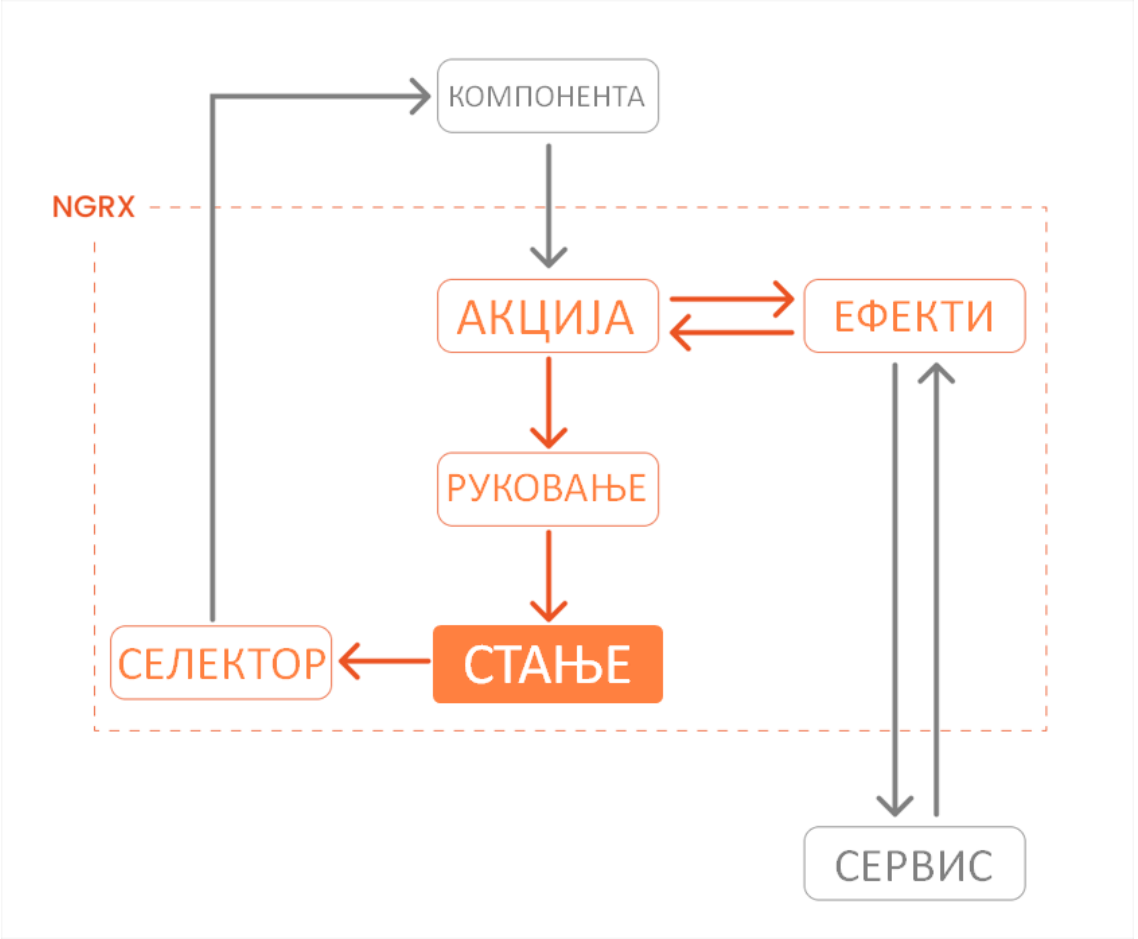
Управљање стањем (енгл. State Management) представља један од кључних аспеката када говоримо о креирању веб апликација. Код управљања стањем подаци теку од апликације ка стању и обрнуто. Алати за управљање стањем дају тренутни снимак свих података и на тај начин тачно се зна где се подаци налазе, што доста убрзава развој апликације. Алат за управљање стањем који је коришћен у развоју апликације је NgRx који представља развојни оквир отвореног кода за развој реактивних апликација у језику Angular (*State Management in Angular Using NgRx: Pt. 1*, n.d.).

NgRx је заснован на Redux[1]пројектном обрасцу који је заснован на једносмерном току података, где сви подаци пролазе кроз исти животни циклус. Пример животног циклуса података може се видети на слици 2.1. Једносмерни ток података чини стање апликације предвидљивим и много лакшим за разумевање.

NgRx се састоји од:

1. *Стање* — представља базу података клијентске стране.

[1]Може се пронаћи на `https://redux.js.org/`



Слика 2.1: Животни циклус управљања стањем

2. *Акција* — представља јединствени догађај (енгл. event) који се дешава у апликацији. Могу представљати догађаје животног циклуса, интеракције корисника или мрежне захтеве. Акције представљају начин на који апликација комуницира са NgRx-ом којем говори шта да ради.

3. *Руковање* — они су одговорни за руковање промена између 2 стања. Они реагују на акције које су позване и на тај сигнал извршавају функцију која ажурира стање. Те функције су фунцкије које су предвидљиве и које немају нежељене ефекте (енгл. side-effects) што значи да на исти скуп улаза функција ће увек враћати исти скуп излаза.

4. *Селектор* — селектори представљају чисте функције које служе да врате део података из стања. Селектори представљају начин на који наша апликација може да слуша промене стања.

5. *Ефекат* — компонента Ефекти управљају нежељеним ефектима сваке акције који могу бити или комуникација са спољним API-јем преко HTTP-а када је одређена акција позвана или ако је акција позвана да би се ажурирао неки други део стања.

2.2 Серверски део апликације

Серверски део је написан као Spring Boot апликација и циљ апликације је да се при покретању покреће и веб трагач који ће претраживати веб странице, прикупљати информације и чувати их у бази података.

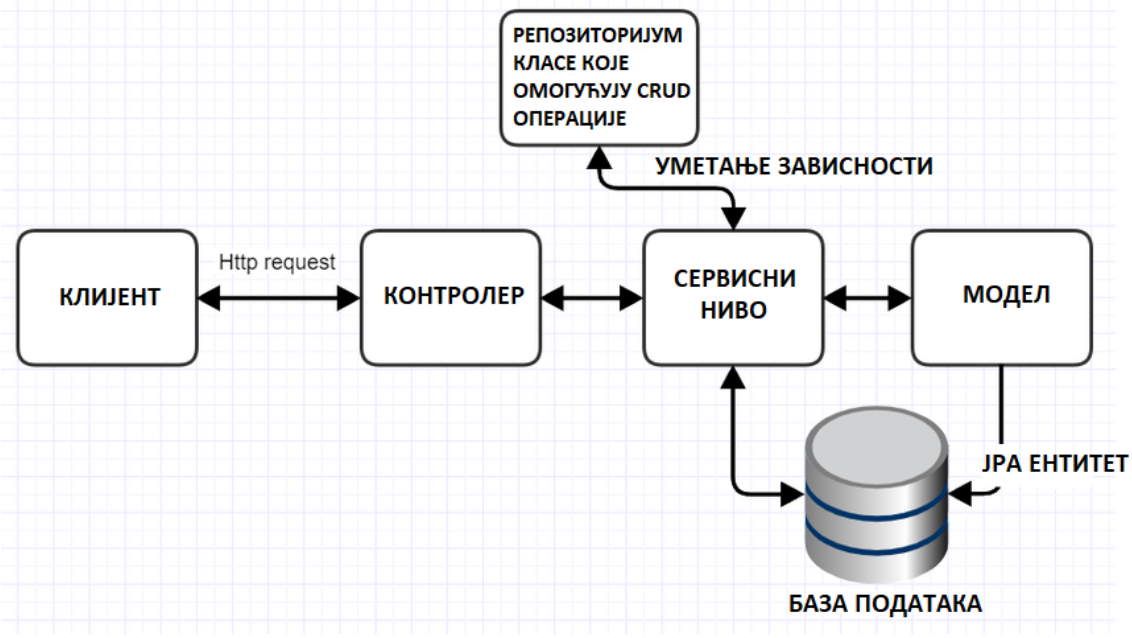
Веб трагач који је коришћен за циљеве овог рада је Crawler4j и изворни код се може пронаћи на адреси[2], а за парсирање самих података које је веб трагач покупио коришћена је JSoup и изворни код се може пронаћи на адреси[3].

Серверски део апликације се састоји из 3 компоненте: контролера, сервисног нивоа и модела. На слици 2.2 је приказана архитектура серверског дела апликације.

- Контролер — компонента која служи за комуникацију са клијентским делом апликације. Захтев се шаље са клијентског дела, контролер прима захтев и на крају враћа податке клијенту.
- Сервисни ниво — представља међу-ниво између контролера и модела чија је улога да садржи сву пословну логику.
- Модел — представља скуп класа које дефинишу и описују податке пословне логике.

Апликација је реализована по принципима REST (енгл. REpresentational state transfer) архитектуре. REST представља скуп правила који се успостављају за размену порука између клијентског и серверског дела апликације и који олакшавају комуникацију између система. Другим речима REST представља образац за креирање API-ја (енгл. Application Programming Interface).

[2] `https://github.com/yasserg/crawler4j`
[3] `https://jsoup.org/`



Слика 2.2: Приказ архитектуре серверског дела апликације

Spring Boot

Spring Boot представља тренутно најпопуларнији развојни оквир отвореног кода за развој Јава апликација. За потребе рада ће бити креирана апликација у којој ће постојати један сервис (модул) који ће садржати сву логику.

Али свакако Spring Boot има још доста својих предности због којих је за развој апликације баш коришћен овај развојни оквир отвореног кода. Неки од њих су:

- Лако постављање апликације на веб сервер. Spring Boot има свој уграђен веб сервер Tomcat, који се такође може лако заменити и са осталим веб серверима (Jetty, Undertow, Resin и остали). Овакав начин рада олакшава посао програмерима који не морају да размишљају о конфигурисању веб сервера (*Pros and Cons of Using Spring Boot*, n.d.).

- Аутоматска конфигурација Spring Boot-а се прилагођава укљученим библиотекама. Нпр. ако се користи spring-boot-starter-jdbc, Spring Boot ће аутоматски регистровати одређена зрна (енгл. beans) и читаће информације о конекцији за базу података директно из application.properties датотеке изворног кода. Сва аутоматска конфигурација може да буде прегажена у било ком тренутку.

- WAR (енгл. Web Application Resource) датотеке изворног кода нису потребне — уместо њих се могу користити JAR (енгл. Java ARchive) датотеке изворног кода које имају једноставнију структуру и лакше су за коришћење.

- Један од главних аспеката Spring развојног оквира отвореног кода јесте уметање зависности (енгл. Dependency Injection) који представља један од пројектних образаца. Он служи да би се имплементирао принцип који се зове инверзија контроле (енгл. Inversion of Control). Проблем који може да настане код апликација јесте да временом оне расту и постају све сложеније. Тада класе апликације могу да постану чврсто везане (енгл. tightly coupled) што знатно отежава њихово мењање и даљи развој апликације. Зато је циљ да везе између ових класа буду „лабаве" (енгл. loosely coupled). То ће се остварити коришћењем уметања зависности тако што ће се класи прослеђивати објекти који су јој потребни и тиме елиминисати одговорност класе да их она сама креира.

Crawler4j

Crawler4j представља библиотеку која служи за веб трагање и која је коришћена у сврхе овог рада. Први корак додавања ове библиотеке јесте убацивање артефакта у pom.xml датотеку изворног кода.

```
<dependency>
   <groupId>edu.uci.ics</groupId>
   <artifactId>crawler4j</artifactId>
   <version>4.4.0</version>
</dependency>
```

Код 2.1: Приказ артефакта Crawler4j библиотеке

Главни део креирања веб трагача коришћењем ове бибилиотеке јесте класа која садржи две методе *visit()* и *shouldVisit()*. У овим методама се дефинишу филтери који омогућују веб трагачу да претражује само оне странице које испуњавају одређен услов. Ово убрзава рад веб трагача тако што се не посећују странице које не испуњавају дате услове.

Такође се могу побољшати перформансе веб трагача подешавањем броја који представља колико трагач може да иде нивоа у дубину од задате почетне

странице.

JSoup

Jsoup представља Јава библиотеку отвореног кода која служи за издвајање (енгл. extracting) HTML садржаја. Као и код претходне библиотеке први корак је додавање артефакта у pom.xml датотеку.

```
<dependency>
   <groupId>org.jsoup</groupId>
   <artifactId>jsoup</artifactId>
   <version>1.14.3</version>
</dependency>
```

Код 2.2: Приказ артефакта JSoup библиотеке

Главна функција ове библиотеке јесте да у тренутку кад веб трагач довуче интернет страницу она извуче HTML податке потребне за даљу анализу.

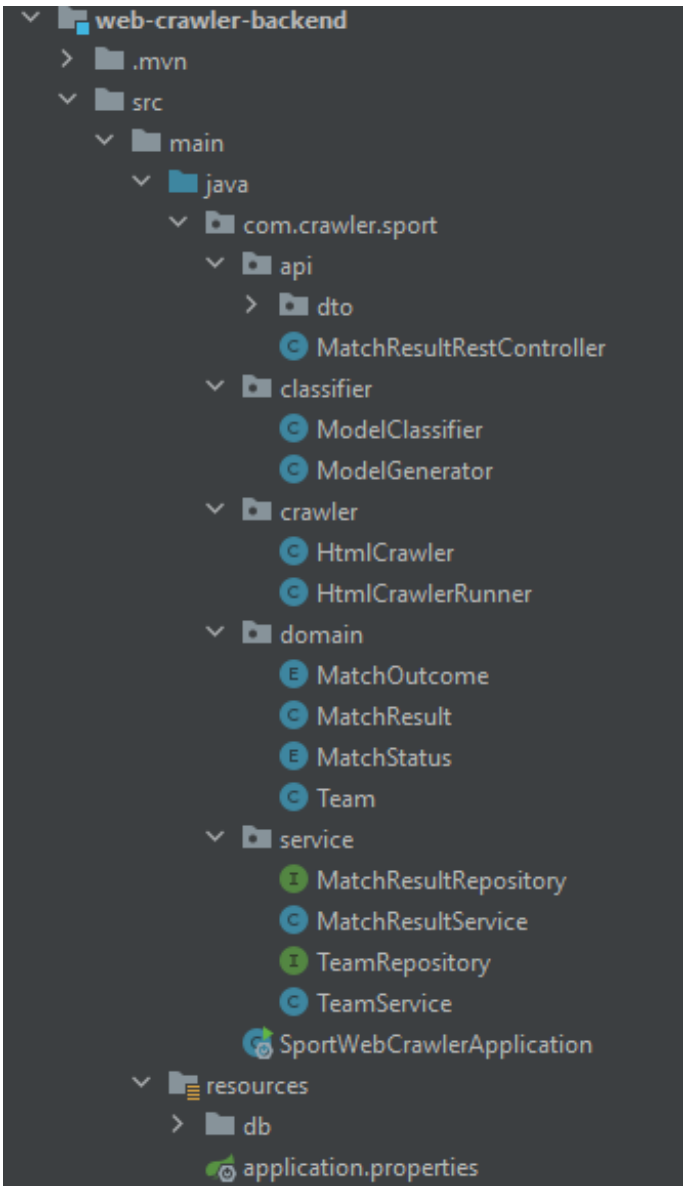
2.3 Структура изворног кода апликације

Апликација која је рађена за потребе овог рада се зове *SportWebCrawlerApplication* која се састоји из 2 модула: *web-crawler-backend* (серверски део) и *web-crawler-frontend* (клијентски део).

Серверски део се састоји од главне класе *SportWebCrawlerApplication* која покреће апликацију. На слици 2.3 је приказана структура серверског дела апликације која се састоји од 5 пакета:

- *api* — пакет у који се смештају сви REST API позиви које се зову са клијентске стране као и DTO (Data Transfer Object) датотеке изворног кода. Оне служе за енкапсулацију класа из доменског пакета.
- *crawler* — пакет у којем се налази главна класа која представља веб трагач и где се налази главна логика трагања.
- *domain* — пакет у којем се налазе ентитети који представљају табеле у бази података.

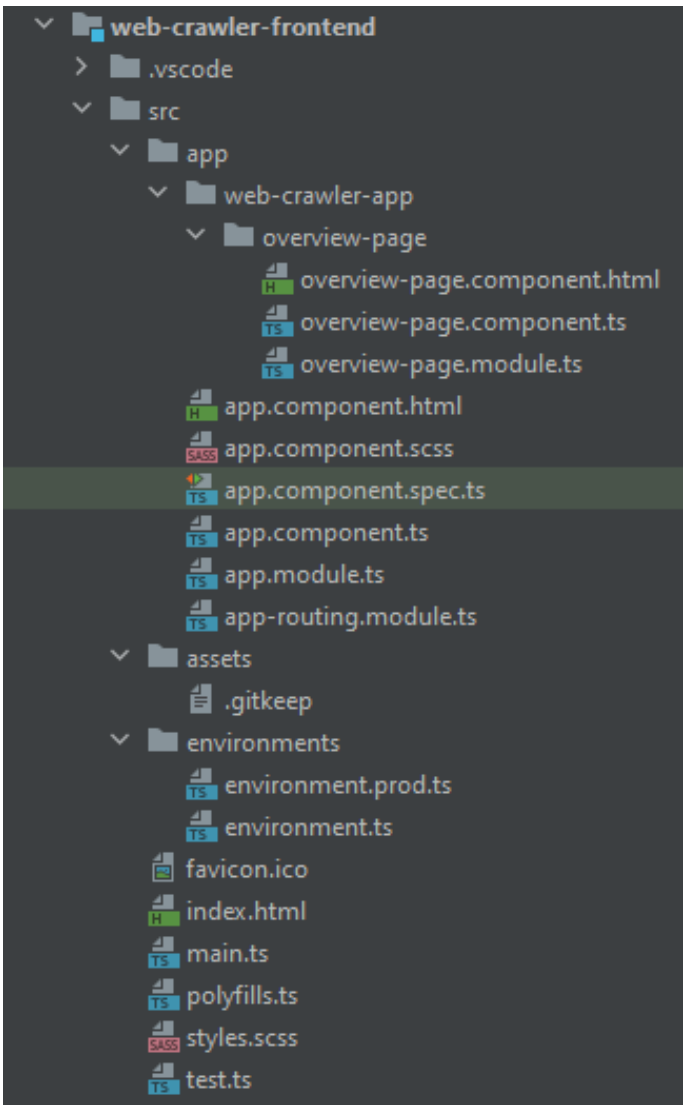
- *service* — пакет где су смештени сервиси и репозиторијуми који омогућавају комуникацију са базом података и уз помоћ којих се чувају и читају подаци.
- *classifier* — пакет у којем се налазе класе везане за класификацију исхода мечева.



Слика 2.3: Приказ структуре изворног кода серверског дела апликације

Клијентски модул апликације *web-crawler-frontend*, чија је структура приказана на слици 2.4, представља једностраничну Angular апликацију. Она

поред датотека изворног кода, који су аутоматски изгенерисани од стране језика Angular, има и пакет *overview-page*, који представља главну страницу апликације. На овој страници је приказана листа фудбалских утакмица са предвиђеном класом исхода утакмице.



Слика 2.4: Приказ структуре изворног кода клијентског дела апликације

Као што је поменуто у раду, апликација ће имати једну страницу на којој је ће корисник моћи да види листу утакмица које нису још одигране и једну од предвиђених предефинисаних класа за тај меч (победа домаће екипе, нерешено, победа гостујуће екипе). Предвиђена класа се израчунава на основу резултата обе екипе који су прикупљени уз помоћ веб трагача.

Листа утакмица на клијентски део се довлачи са серверског дела уз помоћ REST API позива, а такође уз сваку утакмицу корисник види и претходне одигране утакмице оба тима на основу којих је и добијена предвиђена класа коначног исхода актуелне утакмице.

Глава 3

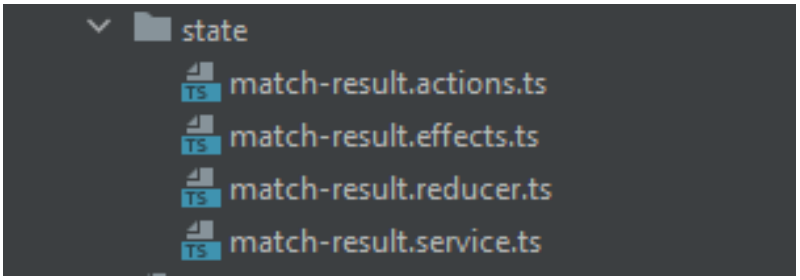
Имплементација апликације

У овом поглављу ће бити приказана имплементација апликације. Такође ће бити приказани и најбитнији делови кода који садрже главну логику апликације, а комплетан код апликације се налази на адреси[1]

3.1 Клијентски део

Као што је споменуто у секцији 2, клијентски део апликације је написан у развојном оквиру Angular уз коришћење библиотеке NgRx, чија је структура датотека изворних кодова приказана на слици 3.1. Она служи да контролише податке који се размењују са серверским делом апликације.

Имплементација Redux обрасца у апликацији се састоји из пакета *state* у ком су смештене *typescript* датотеке изворних кодова, који чине управљање подацима лакшим.



Слика 3.1: Структура Redux пројектног обрасца

[1] `https://github.com/soja4/SportsWebCrawlerApplication`

- *match-result.actions.ts* — представља скуп акција које се активирају када треба да се покрене одређени догађај;
- *match-result.effects.ts* — представља скуп сервиса који ослушкују акције и сваки пут када је одређена акција позвана, ефекат који је ослушкује ће покренути одређени посао;
- *match-result.reducer.ts* — представља објекат који обрађује транзицију из једног у друго стање;
- *match-result.service.ts* — представља скуп метода који позивају API серверског дела апликације.

Када корисник изабере одређен датум, на клијентској страни ће се активирати акција, коју ће ослушнути одговарајући ефекат. Након тога ће ефекат послати захтев за довлачење утакмица које требају да се одиграју на тај датум. Корисник ће затим видети списак утакмица и предвиђени исход утакмице.

3.2 Серверски део

Као што је већ споменуто, за развој апликације и имплементацију веб трагача коришћена је библиотека отвореног кода *Crawler4j*. Логика покретања и извршавања веб трагача је смештена у пакет *crawler* у ком се налазе 2 класе *HtmlCrawler* и *HtmlCrawlerRunner*.

HtmlCrawlerRunner — класа која имплементира интерфејс *CommandLineRunner* и у којој је дефинисана метода *run()* која је приказана у коду 3.1. Ова методапокреће уједно и веб трагач. У оквиру ове класе се може дефинисати још доста подешавајућих вредности као што су локација где ће се чувати странице, број трагача и почетну страницу одакле ће веб трагач започети претрагу. За потребе овог рада почетна страница је постављена на `https://www.bbc.com/sport/football/premier-league/scores-fixtures/2022-05`

```
public void run(String... args) throws Exception {
        File crawlStorage = new File("src/test/resources/crawler4j");
        CrawlConfig config = new CrawlConfig();
        config.setCrawlStorageFolder(crawlStorage.getAbsolutePath());

        int numCrawlers = 12;

        PageFetcher pageFetcher = new PageFetcher(config);
        RobotstxtConfig robotstxtConfig = new RobotstxtConfig();
        RobotstxtServer robotstxtServer = new
            RobotstxtServer(robotstxtConfig, pageFetcher);
        CrawlController controller = new CrawlController(config,
            pageFetcher, robotstxtServer);

        BlockingQueue<String> urlsQueue = new ArrayBlockingQueue<>(400);

        controller.addSeed(urlRoot);

        CrawlController.WebCrawlerFactory<HtmlCrawler> factory =
                () -> new HtmlCrawler(matchResultService, teamService);

        controller.start(factory, numCrawlers);
    }
```

Код 3.1: Приказ кода методе *run()*

HtmlCrawler — класа која наслеђује класу *WebCrawler* и у којој је дешава сва логика веб трагача. Две основне методе које су потребне су *shouldVisit()* и *visit()*:

- *shouldVisit()* — метода у којој се постављају филтери који помажу да се претражују само одређене странице. То се ради тако што се дефинише регуларни израз који мора бити испуњен да би се одеђена страница посетила. Ово је приказано у коду 3.2.
- *visit()* — метода у којој се, ако одређен линк испуњава критеријуме да буде посећен, преузима страница, затим уз помоћ споменуте библиотеке

JSoup парсира HTML садржај и извлаче подаци који су потребни. На крају се преузети садржај чува у бази података. Почетак методе је приказан у блоку кода 3.3 где се приказује употреба JSoup библиотеке за парсирање HTML страница.

```
@Override
    public boolean shouldVisit(Page referringPage, WebURL url) {
        String urlString = url.getURL().toLowerCase();
        return !EXCLUSIONS.matcher(urlString).matches()
                &&
                    urlString.startsWith("https://www.bbc.com/sport/football");
    }
```

Код 3.2: Приказ кода методе *shouldVisit()*

```
@Override
    public void visit(Page page) {
        String url = page.getWebURL().getURL();

        if (page.getParseData() instanceof HtmlParseData &&
            url.matches(".+\\/football\\/\\d+")) {
            HtmlParseData htmlParseData = (HtmlParseData)
                page.getParseData();

            String html = htmlParseData.getHtml();

            Document doc = Jsoup.parseBodyFragment(html);

            .
            .
            .
            .
        }
    }
```

Код 3.3: Приказ кода методе *visit()*

Подаци који су потребни се узимају уз помоћ класа HTML елемента:

- резултат;
- име домаће екипе;
- име гостујуће екипе;
- датум утакмице;
- статус.

Сви ови подаци се користе да би се креирали објекти класа *MatchResult* и *Team* које се налазе у пакету *domain*. Уз то постоје још и енумерисани типови *MatchStatus* и *MatchOutcome*.

За везу између класа *MatchResult* и *Team* се користи анотација *@OneToOne*, која је приказана у делу кода 3.4. Она се користи како би у класи *MatchResult* за поља *homeTeam* и *awayTeam* био сачуван идентификатор (id) тима из ентитета *Team*. На овај начин се избегава дуплирање, а и олакшава претраживање утакмица за одређену екипу.

```
public class MatchResult {

   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Integer id;

   @OneToOne(cascade = CascadeType.MERGE)
   @JoinColumn(name = "home_team_id", nullable = false)
   private Team homeTeam;

   @OneToOne(cascade = CascadeType.MERGE)
   @JoinColumn(name = "away_team_id", nullable = false)
   private Team awayTeam;
}
```

Код 3.4: Приказ кода ентитета MatchResult

У пакету *service* су смештени сви сервиси и репозиторијуми који служе као веза са базом података. Ту је смештена и сва пословна логика апликације.

За сваки ентитет понаособ постоје репозиторијуми који су интерфејси и сервисне класе који опслужују тај ентитет. Тако нпр. за ентитет *MatchResult* постоји креиран интерфејс *MatchResultRepository* и сервис класа *MatchResultService*. Репозиторијуми су интерфејси који имплементирају интерфејс *JpaRepository*. Они имају уграђене CRUD операције за креирање (енгл. Create), читање (енгл. Read), ажурирање (енгл. Update), брисање (енгл. Delete) (код 3.5). У репозиторијуму *MatchResultRepository* постоје методе које се ослањају на коришћење интерфејса JpaRepository, а постоји и метода *findFinishedByHomeTeamIdOrAwayTeamId()* која формира SQL упит.

```
@Repository
public interface MatchResultRepository extends JpaRepository<MatchResult,
    Integer> {

    @Query(
           nativeQuery = true,
           value =
                  "select * "
                         + "from match_result "
                         + "where (home_team_id = ?1 or away_team_id = ?1)
                             "
                         + "and status = 'FINISHED' order by match_date
                             desc "
                         + "limit 10")
    List<MatchResult> findFinishedByHomeTeamIdOrAwayTeamId(Integer teamId,
        MatchStatus matchStatus);

    List<MatchResult> findByStatus(MatchStatus matchStatus, Pageable
        pageable);

    List<MatchResult> findByHomeTeamIdAndStatus(
           Integer teamId, MatchStatus matchStatus, Pageable pageable);

    List<MatchResult> findByAwayTeamIdAndStatus(
           Integer teamId, MatchStatus matchStatus, Pageable pageable);

    List<MatchResult>
        findByHomeTeamIdAndAwayTeamIdAndHomeTeamGoalsAndAwayTeamGoalsAndMatchDate(
            Integer homeTeamId,
            Integer awayTeamId,
            Integer homeTeamGoals,
            Integer awayTeamGoals,
            LocalDate matchDate);
}
```

Код 3.5: Приказ кода репозиторијума MatchResultRepository

3.3 Класификација исхода меча

Да би се креирао класификациони модел потребно је прво да се издвоје које су то најбитније карактеристике које ће се користити у рачуници.

Карактеристике које се користе су (Buursma, 2017):

1. gol_dom_n — број постигнутих голова од стране домаће екипе у последњих N утакмица;
2. gol_gost_n — број постигнутих голова од стране гостујуће екипе у последњих N утакмица;
3. prim_gol_dom_n — број примљених голова од стране домаће екипе у последњих N утакмица;
4. prim_gol_gost_n — број примљених голова од стране гостујуће екипе у последњих N утакмица;
5. avg_points_dom_n — просечан број поена освојених од стране домаће екипе у последњних N утакмица;
6. avg_points_gost_n — просечан број поена освојених од стране гостујуће екипе у последњних N утакмица;
7. nr_wins_dom_n — број добијених утакмица на домаћем терену од стране домаће екипе у последњих N утакмица;
8. nr_wins_gost_n — број добијених утакмица на гостујућем терену од стране гостујуће екипе у последњих N утакмица;
9. gol_dom_athome_n — број постигнутих голова на домаћем терену од стране домаће екипе у последњих N утакмица;
10. gol_gost_ataway_n — број постигнутих голова на гостујућем терену од стране гостујуће екипе у последњих N утакмица.

Просечан број поена из ставки 5. и 6. се рачуна тако што се збир освојених поена подели са бројем одиграних утакмица, а поени које тим може освојити на једној утакмици су:

- победа — 3 поена;

- нерешено — 1 поен;
- пораз — 0 поена.

Проблем који постоји јесте количина података. Идеално би било да постоје све утакмице икада одигране за неку екипу, али пошто је количина података ограничена мора се наћи оптимална вредност за N, за које ће успешност модела бити највећа. Овом темом ћемо се бавити у поглављу 4.

Логика класификације је смештена у пакет *classifer* и састоји се од 2 класе:

- *ModelClassifier* — класа у којој се прави класификациони модел и која се састоји од конструктора модела који садржи све атрибуте. Метода *classify()* као аргумент прима листу инстанци (података) које треба класификовати.
- *ModelGenerator* — класа која се састоји од метода:
  - *buildClassifier()* — метода која креира модел и којој се као аргумент прослеђују тренинг подаци (видети код 3.6).
  - *evaluateModel()* — метода којој се као аргумент прослеђују тест подаци и која за циљ има да израчуна успешност класификационог модела (видети код 3.7).
  - *saveModel()* — метода која чува модел (видети код 3.8).

```
public Classifier buildClassifier(Instances traindataset) {
       MultilayerPerceptron m = new MultilayerPerceptron();

       try {
           m.buildClassifier(traindataset);

       } catch (Exception ex) {
           log.error("Error building classifier");
       }
       return m;
   }
```

Код 3.6: Приказ кода методе *buildClassifier()*

```
public String evaluateModel(Classifier model, Instances traindataset,
    Instances testdataset) {
        Evaluation eval = null;
        try {
            // Evaluate classifier with test dataset
            eval = new Evaluation(traindataset);
            eval.evaluateModel(model, testdataset);
        } catch (Exception ex) {
            log.error("Error evaluating model: {}",
                ex.getLocalizedMessage());
        }
        return eval.toSummaryString("", true);
    }
```

Код 3.7: Приказ кода методе *evaluateModel()*

```
public void saveModel(Classifier model, String modelPath) {

        try {
            SerializationHelper.write(modelPath, model);
        } catch (Exception ex) {
            log.error("Error saving model");
        }
    }
```

Код 3.8: Приказ кода методе *saveModel()*

Библиотека која је коришћена за потребе овог рада је Weka. У питању је библиотека отвореног кода која садржи колекцију алгоритама за различите области машинског учења. Прецизније, она садржи функционалности за припрему података, класификацију, регресију, кластеровање и визуелизацију. Убацивање артефакта за Weka библиотеку приказано је на 3.9.

```
<dependency>
   <groupId>nz.ac.waikato.cms.weka</groupId>
   <artifactId>weka-stable</artifactId>
   <version>3.8.6</version>
</dependency>
```

Код 3.9: Приказ артефакта Weka библиотеке

Логика креирања класификационог модела као и предвиђање класа података се налази у методи *getMatchesToBeplayedAndPredictions()* у оквиру сервиса *MatchResultService*. У делу кода 3.8 приказан је први део методе у ком се довлаче подаци из базе података, тачније утакмице које су одигране. Оне се користе за тренинг скуп, на основу којих се прави класификациони модел, као и за тест скуп који даље служи за његову евалуацију. Код 3.10 приказује креирање скупа података за тренинг и тест. Метода у којој се учитавају подаци приказана је у коду 3.11. Поставља се однос између величина тренинг и тест скупова на 80% према 20%. Потом се из базе података извлаче утакмице које нису одигране и за сваку од њих, уколико испуњавају услове да обе екипе имају одређен број одиграних утакмица у прошлости, се предвиђа коначан исход утакмице.

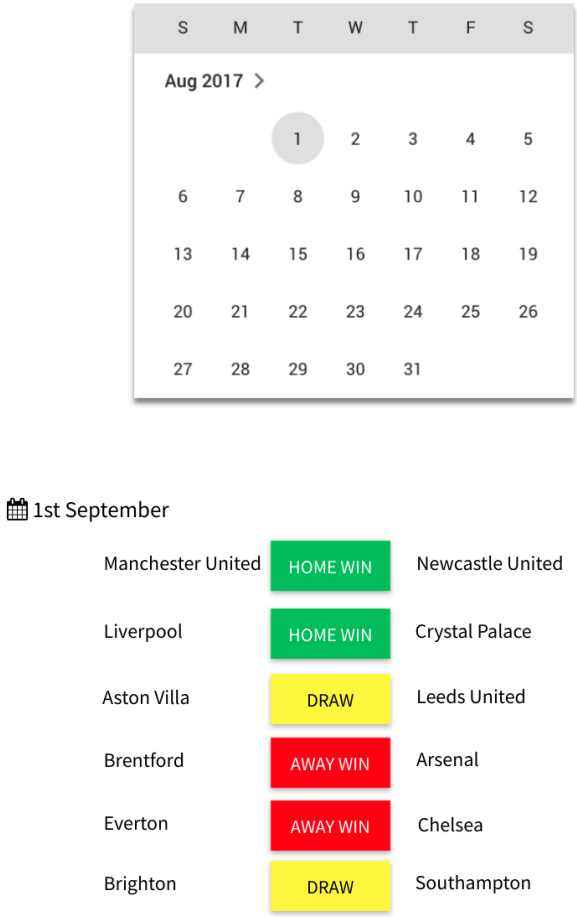
```
public List<MatchResult> getMatchesToBePlayedAndPredictions() throws
    Exception {
        List<MatchResult> matches = getMatchesToBePlayed();

        Instances dataset = loadDataset();

        // divide dataset to train dataset 80% and test dataset 20%
        int trainSize = (int) Math.round(dataset.numInstances() * 0.8);
        int testSize = dataset.numInstances() - trainSize;

        dataset.randomize(new Debug.Random(1));

        // Normalize dataset
        filter.setInputFormat(dataset);
        Instances datasetnor = Filter.useFilter(dataset, filter);

        Instances traindataset = new Instances(datasetnor, 0, trainSize);
        Instances testdataset = new Instances(datasetnor, trainSize,
            testSize);

        // build classifier with train dataset
        MultilayerPerceptron ann = (MultilayerPerceptron)
            mg.buildClassifier(traindataset);

        // Evaluate classifier with test dataset
        String evalsummary = mg.evaluateModel(ann, traindataset,
            testdataset);
        System.out.println("Evaluation: " + evalsummary);

        // Save model
        mg.saveModel(ann, MODEL_PATH);

}
```

Код 3.10: Приказ дела кода методе *getMatchesToBeplayedAndPredictions()*

```
private Instances loadDataset() {
       Instances dataset = getFinishedMatchesAndMakeInstances();
       try {

           if (dataset.classIndex() == -1) {
               dataset.setClassIndex(dataset.numAttributes() - 1);
           }
       } catch (Exception ex) {
           log.error("Error loading data");
       }

       return dataset;
   }

}
```

Код 3.11: Приказ дела кода методе *loadDataset() која прави инстанце за тренинг и тест скупове*

И на крају се листа неодиграних утакмица заједно за предвиђеном класом шаље на клијентску страну које се приказују кориснику што је приказано на слици 3.2.



Слика 3.2: Изглед клијентског дела апликације

Глава 4

Евалуација модела

Мере квалитета којим се може описати квалитет модела су: тачност (енгл. accuracy), прецизност (енгл. precision), одзив (енгл. recall), F-мера (енгл. F1) (Nikolić & Zečević, 2019). Све мере квалитета почивају на матрици конфузије која је приказана на у табели 4.1

- Стварно позитивно (TP) — позитивне инстанце које је модел препознао као позитивне;
- Стварно негативно (TN) — негативне инстанце које је модел препознао као негативне;
- Лажно позитивно (FP) — негативне инстанце које је модел препознао као позитивне;
- Лажно негативно (FN) — позитивне инстанце које је модел препознао као негативне.

1. Тачност (енгл. accuracy)

$$Acc = \frac{TP}{TP + FP + TN + FN}$$

Табела 4.1: Матрица конфузије код бинарне класификације

Стварно/Предвиђено	Позитивно	Негативно
Позитивно	Стварно позитивно (TP)	Лажно негативно (FN)
Негативно	Лажно позитивно (FP)	Стварно негативно (TN)

2. Прецизност (енгл. precision)

$$Prec = \frac{TP}{TP + FP}$$

3. Одзив (енгл. recall)

$$Rec = \frac{TP}{TP + FN}$$

4. F-мера

$$F_1 = 2 * \frac{Prec * Rec}{Prec + Rec}$$

Можда и најбитнија ставка код евалуације класификационог модела јесте тачност предвиђања класе. У овом поглављу ћемо приказати пар примера како тачност може да варира у зависности који број инстанци узимамо за креирање класификационог модела као и критеријуме које инстанце морају да испуњавају. У табели 4.2 могу се видети подаци о класификационим моделима који се разликују у броју инстанци који је коришћен.

Појашњење назива колона из табеле 4.2

- TI (Total Instances) — укупан број инстанци;
- CCI (Correctly Classified Instances) — број тачно предвиђених инстанци;
- PCCI (Percentage of Correctly Classified Instances) — проценат тачно предвиђених инстанци;
- ICI (Incorrectly Classified Instances) — број погрешно предвиђених инстанци;
- PICI (Percentage of Correctly Classified Instances) — проценат погрешно предвиђених инстанци;
- CCI + ICI — збир тачно предвиђених инстанци и погрешно предвиђених инстанци.

Однос величина тренинг и тест скупова података је 80% према 20%. Минимални број доступних података о претходним утакмицама сваког тима је 10.

Табела 4.2: Упоредни однос резултата класификације на тест скуповима за различити број инстанци

TI	CCI	PCCI	ICI	PICI	CCI + ICI
1000	105	57.69%	77	42.30%	182
5000	454	50.61%	443	49.38%	897
26607	2068	42.25%	2826	57.74%	4894

Примећује се да број инстанци који учествује у тест фази је смањен и да не представља 20% од укупног броја инстанци, тј. да збир броја тачно предвиђених инстанци и броја погрешно предвиђених инстанци није једнак укупном броју инстанци у тест скупу. Ово је последица тога што за неке утакмице, једна од екипа не испуњава услов, тј. за ту екипу немамо информације о броју одиграних утакмица у прошлости (N) и самим тим ту утакмицу не узимамо у обзир.

Број N је одређен на основу тога што са том вредношћу се добија најбоља (највећа) тачност на тренинг скупу. Када се ради са 800 тренинг података, добијају се следеће тачности за различите вредности N:

- $N = 10 \rightarrow 60.27\%$
- $N = 20 \rightarrow 59.34\%$
- $N = 30 \rightarrow 58.20\%$

Видимо да се са повећањем вредности N, успешност модела смањује тако да за вредност N бира број 10.

4.1 Унапређење модела

Као главни циљ који класификација има јесте да са што већом успешношћу предвиђа којој класи припада нека инстанца. С обзиром да је добијена тачност од 57%, то је јасни показатељ да има доста простора за унапређење класификационог модела.

Један од начина да се унапреди класификациони модел јесте да се у модел убаци још корисних атрибута који могу да унапреде предвиђање. Унапређење

класификационог модела додавањем нових атрибута, повлачи и унапређење веб трагача.

Конректно везано за фудбалске утакмице, неки од атрибута који могу допринети побољшању модела јесу:

- број шутева у оквир гола;
- број шутева ван оквира гола;
- број фаулова;
- број жутих и црвених картона;
- број корнера;
- посед лопте;
- просечан број гледалаца на утакмици;
- количина новца коју екипе могу да потроше на нове играче;
- број повређених играча у првом тиму;
- тренутна позиција на табели;

Такође исти принцип предвиђања резултата се може применити и на остале спортове као што је бејзбол у ком постоји доста статистичких података и који се могу искористити за креирање доброг класификационог модела. Бејзбол је такође погодан што не може да се деси нерешен резултат па самим тим има и једну класу мање за предвиђање.

Глава 5

Закључак

У овом раду имплементирана је апликација у којој је акценат на два концепта: веб трагач који претражује резултате фудбалских утакмица и класификација која служи да се предвиди коначан исход утакмица.

Веб трагач је представљен као алат који има улогу да претражи велики број интернет страница и да прикупи велики број информација које ће се касније обрадити и анализирати. Такође се може видети да је коришћење веб трагача много брже и ефективније од ручне претраге која је склона људским грешкама при уносу критеријума за претрагу као и при чувању добијених резултата. За предвиђање коначног исхода утакмица коришћена је класификација која за циљ има да предвиди један од могућих коначних исхода.

Добијени класификациони модел има тачност од 57.69%, и овај податак указује да је модел далеко од савршеног. Стога има простора за унапређивање модела додавањем нових атрибута.

Будући кораци у побољшавању модела јесу одабир и додавање нових атрибута који би допринели повећању тачности. Они повлаче за собом и побољшање веб трагача како би могао да прикупи нове атрибуте. Такође један од будућих корака би могло бити и коришћење веб трагача за предвиђање резултата неког другог спорта, нпр. бејзбола који је погодан за овакву врсту анализа због доста статистичких података које садржи.

Кад се говори о побољшању веб трагача, он се може побољшати додавањем методе која ће се извршавати у одређено време (енгл. cron job) и која ће на одређени временски период прикупљати свеже податке. На тај начин ће подаци који се користе за тренинг и тест скуп бити најновији и самим тим ће евалуација класификационог модела бити меродавнија.

Литература

8 proven reasons you need angular for your next development project. (n.d.). Retrieved from https://www.grazitti.com/blog/8-proven-reasons-you-need-angular-for-your-next-development-project/

Ajzenhajmer, N., Bukurov, A., & Stanković, V. (2017). *Istraživanje podataka.* https://www.nikolaajzenhamer.rs/assets/pdf/ip.pdf.

Buursma, D. (2017). *Predicting sports events from past results towards effective betting on football matches.* University of Twente.

Nikolić, M., & Zečević, A. (2019). *Mašinsko učenje.* http://ml.matf.bg.ac.rs/readings/ml.pdf.

Olson, C., & Najork, M. (2010). *Web crawling.* Foundations and Trends in Information Retrieval.

Pros and cons of using spring boot. (n.d.). Retrieved from https://bambooagile.eu/insights/pros-and-cons-of-using-spring-boot/

State management in angular using ngrx: Pt. 1. (n.d.). Retrieved from https://auth0.com/blog/state-management-in-angular-with-ngrx-1/