Adaptív tanító rendszer

Bödő Lajos

Frontó András Levente

Konzulens:

Benedek Zoltán

Budapest, 2020.05.26

1. Bevezetés

Az önálló laboratóriumunk során szerettünk volna egyszerre foglalkozni webes technológiákkal és mesterséges intelligenciával. A két témakör összekapcsolásának megvalósítását egy olyan webalkalmazásban képzeltük el, ami implementál egy mesterséges intelligenciával működő funkciót.

Ötletelésünk során az adaptív tanítás témaköre keltette fel az érdeklődésünket. A személyre szabott tanulás alkalmazása automatizált, gépi technológiákkal egy fejlődő terület, aminek jelentős haszna lenne. Így a cél az volt, hogy az elkészítendő webalkalmazás a felhasználó számára javaslatokat tegyen, hogy milyen irányba folytassa a tanulást. Ennek az ötletnek eredményeként adott volt, hogy szükséges valamilyen ajánló rendszer készítése.

A téma amiben az ajánló rendszer működni fog akkor dőlt el, amikor találtunk egy, a teljes Stack Overflow-t lefedő adat szettet. Ezen adat szett segítségével lehetőségünk van, hogy a felhasználó által már eddig átnézett posztok alapján a számára legrelevánsabbakat ajánljuk.

Ez a megközelítés felülszárnyalja a Stack Overflow által jelenleg javasolt kérdéseket, mert az csak az éppen vizsgálthoz leghasonlóbbakat ajánlja. Ezen felül egy ilyen alkalmazás nem csak a Stack Overflow posztjait veheti alapul az ajánlásnál hanem akár más az alkalmazásból származó adatokat is pl.: a felhasználó adatai, személyes igényei, szakértelme.

Az alkalmazást egy react és redux alapú webalkalmazás formájában szerettük volna megvalósítani ASP.NET Core backend-vel.

1.1. Webalkalmazás célok

Az alkalmazás egy többfelhasználós alkalmazás kell hogy legyen hogy minden felhasználó számára külön javaslatokat tehessen. Az alkalmazásnak implementálni kell a beléptető, regisztrációs és azonosító folyamatokat.

A felhasználók a kérdéseiket szálakba fűzve rendszerezve látják. A szálak teljesen elkülönülnek és az alkalmazás mindig egy adott szálhoz ajánl újabb Stack Overflow posztokat. Az alkalmazás meg tud jeleníteni a felhasználó számára könnyen átlátható formában egy adott szálat. Valamint a felhasználó képes elfogadni javaslatokat illetve törölni a szálból már régebben elfogadott posztokat.

1.2. Mesterséges intelligencia célok

A mesterséges intelligenciával azt terveztük elérni, hogy a bemenetként megadott több kérdésre tudjon több ajánlást is adni. Az ajánlott kérdések kapcsolódjanak a bemeneti kérdésekhez. A különböző lehetőségeket is terveztük megvizsgálni, hogy milyen módszerekkel lehet végezni a tanítást.

2. Mesterséges intelligencia

A mesterséges intelligenciával kapcsolatos feladatokat, az adatfeldolgozást és a tanítást pythonban végeztük.

2.1. Adaptív tanítás

Az adaptív tanulás legnagyobb előnye a személyre szabhatóság. A hagyományos módszerekkel készült kurzusok általános módon vannak tervezve, hogy mindenki számára megfelelő legyenek. Ilyen formában természetesen nincsenek figyelembe véve az adott egyén erősségei és gyengeségei, bizonyos témák könnyűek lesznek a felhasználó számára és a kurzus túl sok időt fordít rá, más témák nehezek lehetnek, a felhasználó úgy érezheti, nem volt elég jól kifejtve az elsajátítandó témakör.

Szintén ritkán kerül felmérésre, hogy milyen előzetes tudással rendelkezik a felhasználó. Ez egymásra épülő kurzusok esetén ritkán probléma, mivel a korábbi kurzusok felkészítik a felhasználót a későbbiekre. A magukban álló tanfolyamok esetében viszont előfordulhat, hogy a tanuló egyes készségekben kezdő másokban viszont már tapasztalt. Ilyen esetekben a tanuló kénytelen lehet más forrásokat keresni, hogy pótolja hiányosságait.

Az imént említett problémák főleg akkor jöhetnek elő a legjobban, ha az ember az interneten keres információt, forrásokat egy adott témakör esetén. Különösen igaz lehet ez, ha a témakörök programozáshoz kapcsolódóak. rengeteg oktató, bemutató anyag található, ezek rendszerezett felhasználása nem egyszerű. Bevett módszer, hogy az ember utánanéz egy kérdésnek,melyre a válasz gyakran található meg a Stack Overflow oldalán. Itt a válaszhoz kapcsolódó további kérdések is fel vannak tüntetve, amik segítik a kérdezőt tovább keresni. Ugyanakkor ezek az ajánlások nem veszik figyelembe milyen más kérdéseket kerestett a felhasználó a témában, a mi web alkalmazásunknak pont ez a célja, hogy ne csak a közvetlen kapcsolatokat használjuk fel, hanem több korábbi kérdést is.

2.2. Adatok

A Stack Overflow posztoknak adatai és a posztokhoz kapcsolódó egyéb adatok könnyen elérhetőek. Egyrészt a Google Big Query szolgáltatásnak köszönhetően a kért információk lekérhetőek a web alkalmazás futása közben, a teljes alapadat feldolgozása és

tárolása nélkül. Másrészt az adatok letölthetőek az archive.org oldalról, így felhasználhatóak az offline tanításhoz. Az adatok negyedévente frissülnek, mi a 2019 decemberi adatokat használtuk.

2.2.1. Adatfeldolgozás

Az adatok több különböző táblában voltak reprezenálva. Mi kettő használtunk fel közülük, a *stackoverflow_posts* és a *tags* táblákat. A *stackoverflow_posts* tábla tartalmazta a posztokhoz kapcsolódó legfontosabb információkat. A tábla körül-belül 31 millió rekordot tartalmaz, a kérdező posztokon kívül tartalmazza a posztokra adott válaszokat is. A *tags* tábla az összes előforduló tag-et tartalmazza.

Hogy fel tudjuk használni az adatokat a mesterséges intelligencia betanítására fel kellett dolgoznunk a *stackoverflow_posts* táblát kisebb darabokra, amikben már csak a számunkra fontos információval bírós adat tulajdonságok szerepelnek. Illetve a nagy méret miatt, egy-egy darabban csak a posztok egy része szerepel.

Az alapadat xml fájlformátumban volt, ezért az alapadat parszolását python-ban végeztük az xml.sax és xml.etree modulokkal. A kimenetek szintén xml fájlok voltak. A parszolás során a ki lettek szűrve azok a posztok, amik nem kérdező posztok voltak, hanem válaszok, illetve az adathibások posztok is. Az általunk fontosnak talált adatok a poszt címe, szövege, tag-jei, a poszt pontszáma, a kedvenc jelölések száma, a nézettség, a válaszok és kommentek száma.

A posztok feldolgozása mellett, készült egy csv fájl, ami tartalmazta az összes előforduló tag-et az adott méretű adatban, illetve egy másik csv fájl is,ami pedig azt tárolta, hogy az adatban előforduló egyes posztokhoz, mely tag-ek tartoznak.

2.3. Tanítás

A mesterséges intelligencia tanítását Google Colab segítségével végeztük. A Google Colab lehetővé teszi, hogy egy Jupyter Notebook-ban felhasználjuk a Google által nyújtott erőforrásokat neurális háló tanítására. A korábban elkészített kisebb adatfájlokat fel lehet tölteni a Colab-ba, vagy egy Google Drive-t csatlakoztatni lehet a Colab futáshoz és így elérhetővé válnak az adatok a tanításhoz. Mi az utóbbi módszert alkalmaztuk.

A feltöltött xml fájlokat újraparszoltuk és ezúttal egy pandas dataframe-be alakítottuk át. A pandas egy python modul, amit a gépi tanulás során széles körben alkalmaznak adatkezelésre. A készített pandas dataframe-n egyszerűen lehet műveleteket végezni, eldobni oszlopokat (tulajdonságokat), ha nincs rájuk szükség, fel lehet darabolni a dataframe-t ha kisebb egységre lenne szükség, normalizációs műveleteket lehet végezni a kiválasztott tulajdonságokon és az egész dataframe-t könnyen át lehet alakítani egy numpy tömbbé.

A felhasznált adatokat normalizáltuk. A normalizálást kétféleképpen is megvizsgáltuk, az sklearn.preprocessing MinMaxScaler és StandardScaler függvényeivel. Előbbi nulla és

egy közé normalizálja az adatokat, az utóbbi a középérték és a szórás segítségével normalizál. Mi a MinMaxScaler alkalmazása mellett döntöttünk.

2.3.1. Dimenzió redukció

A tanításhoz azt a módszert választottuk, hogy a posztok közötti hasonlóságot a köztük lévő távolság függvényében értelmezzük. A posztokat ennek megfelelően egy öt dimenziós vektorként értelmezzük. Az öt dimenzió a számszerű tulajdonságokat foglalja magába, ahogy ez az alábbi ábrán látható.

	Id	Score	FavoriteCount	ViewCount	AnswerCount	CommentCount
0	4	645	49	45103	13	3
1	6	290	11	18713	6	0
2	9	1754	438	574705	61	5
3	11	1461	537	152779	37	3
4	13	597	147	182042	24	10
...	...	...	...	...	...	...
999861	4234765	1	0	2151	3	0
999862	4234767	0	1	1358	1	0
999863	4234769	1	0	1971	3	3
999864	4234771	0	0	912	2	1
999865	4234774	2	0	557	2	7

A felhasznált tulajdonságok

A szövegfeldolgozásra jövőbeli terveink a Word2Vec alkalmazása, ami azt jelenti, hogy az egyes szavakat egy n dimenziós vektorrá képezzük majd le, és a szöveg ezeknek a vektoroknak az összege lesz, amit hozzáillesztünk a posztokat jelképező vektorhoz.

A dimenzió redukció lehetővé teszi, hogy a posztoknak a lényeges információ tartalmát tartsuk meg, illetve a hangsúlyt a tulajdonságok közötti kapcsolatokatra helyezi .

2.3.2. Autoencoder

A dimenzió redukció elvégzésére két módszert vizsgáltunk meg. Az első ezek közül az autoencoder használata volt. Az autoencoder célja, hogy a bemenetet redukálja, kisebb méretre, úgy hogy a tömörítésből minél pontosabban visszaállítható legyen az eredeti alak. Az autoencoder háló két részből áll, egy encoderből, ami fokozatosan csökkenti a bemenet dimenzióját és egy decoderből, ami visszaállítja az eredeti alakot. A tanítás során a célfüggvény, hogy a bemenethez minél közelebb legyen a kimenet. A tanítás után az encoder rész felhasználható, hogy az összes adatot redukálja.

Az autoencoder megvalósításához a keras modult használtuk.

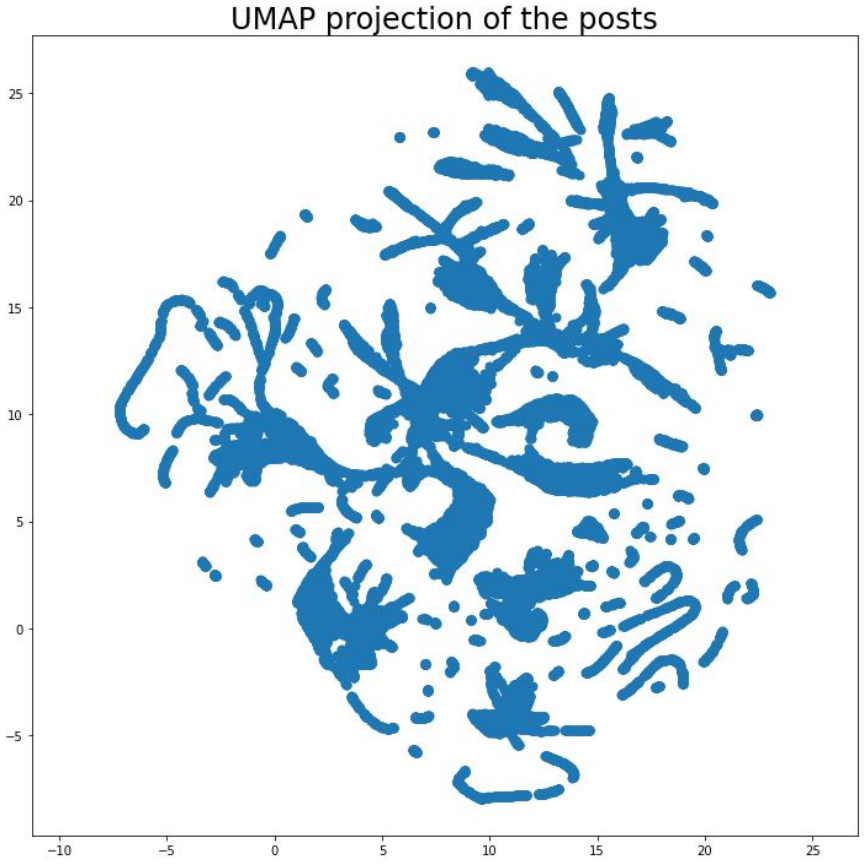
```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
encoder_1 (Dense)            (None, 5)                 30
_________________________________________________________________
encoder_2 (Dense)            (None, 5)                 30
_________________________________________________________________
encoder_3 (Dense)            (None, 4)                 24
_________________________________________________________________
encoder_4 (Dense)            (None, 4)                 20
_________________________________________________________________
bottle_neck (Dense)          (None, 3)                 15
_________________________________________________________________
decoder_1 (Dense)            (None, 4)                 16
_________________________________________________________________
decoder_2 (Dense)            (None, 4)                 20
_________________________________________________________________
decoder_3 (Dense)            (None, 5)                 25
_________________________________________________________________
decoder_4 (Dense)            (None, 5)                 30
=================================================================
Total params: 210
Trainable params: 210
Non-trainable params: 0
```

A készített autoencoder felépítése

2.3.3. UMAP

A dimenzió redukció megvalósítására használt második módszerünk a UMAP volt. A UMAP egy vizualizációs és dimenzió redukciós eszköz. A UMAP működési elve, hogy készít egy magasabb dimenziójú gráfot a bemeneti adatokból, majd készít egy alacsonyabb dimenziójut, úgy hogy az alacsony dimenziójú strukturálisan a lehető legjobban hasonlítson az magasabbra. A kimenet az elkészített alacsonyabb dimenziójú pontok halmaza.

Mi két dimenzióba redukáltuk az ötdimenziós, normalizált posztokat. A UMAP használatának legnagyobb előnye, hogy egy jól bevált eszköz, ami megbízható eredményeket produkál. A UMAP egyben dolgozza fel a bemenetet, emiatt a nagyobb adatmennyiségeknél memóriahiány lépett fel a Google Colab-ban, ezért jelenleg csak 30000 posztot dolgozunk fel UMAP-pal. A UMAP egyszerű használhatósága miatt, ezt használtuk a dimenzió redukcióra, az autoencoderrel szemben.



A UMAP által készített vizualizáció 30 000 posztról

2.3.4. Hierarchikus klaszterezés

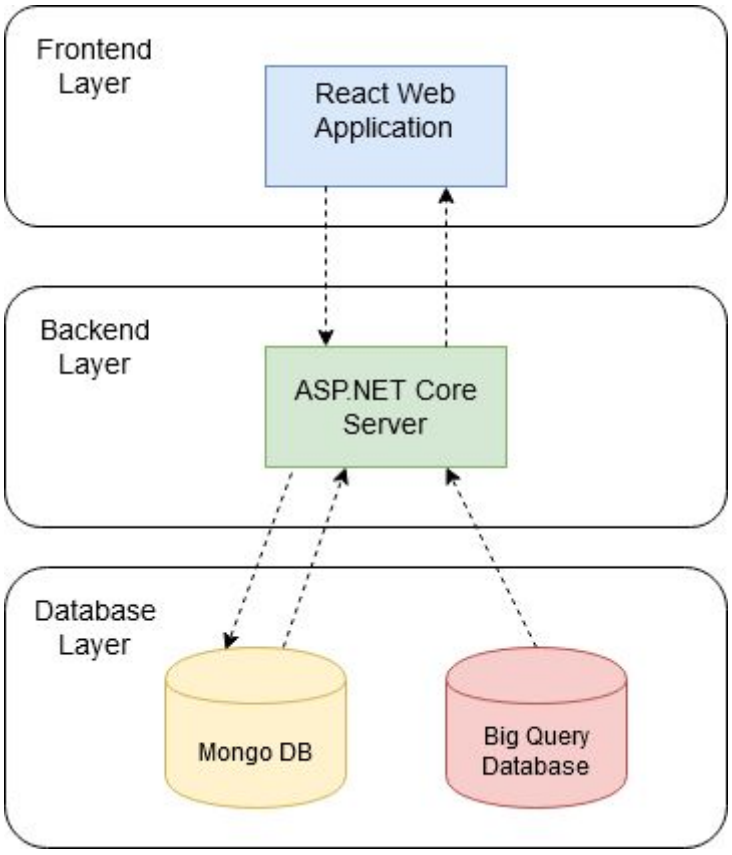
A UMAP dimenzió redukció után rendelkezésünkre áll a posztok kétdimenziós leképezése. Ezeket a pontok felhasználva csoportosíthatjuk a posztokat a hasonlóság alapján. A csoportosítás megvalósítására a hierarchikus klaszterezést használtuk.

A hierarchikus klaszterezés lényege, hogy a bemenet pontjait először önmagukban álló klaszteréknek tekintjük, majd mindig megkeressük az egymáshoz legközelebb lévő két klasztert és azokat összevonjuk egy közös klaszterré. A végén egy csoport lesz, ami az összes pontot tartalmazza. A pontok egy hierarchikus nézet szerint tartoznak bele az egyes csoportokba, minél magasabb rangú egy csoport, annál kevésbé hasonlítanak egymásra a tagjai. A csoportosítás lehetőségét ad arra is, hogy egy adott távolság alapján megvizsgáljuk, hány olyan csoport van, amiknek tangjaik ezen az adott távolságon belül vannak.

A klaszterezésre a scipy.cluster.hierarchy modul linkage függvényét használtuk, ami a klaszterezést egy kapcsolat mátrix formájában adta vissza. A kapcsolat mátrix az tartalmazta, hogy sorban hányas számú klaszterek lettek összevonva egészen addig, amíg csak egy klaszter maradt. A kapott eredményeket a Google Colabból egy .csv fájlba mentettük ki, amit majd a szerver dolgoz fel.

3. A Webalkalmazás

3.1. Architektúra



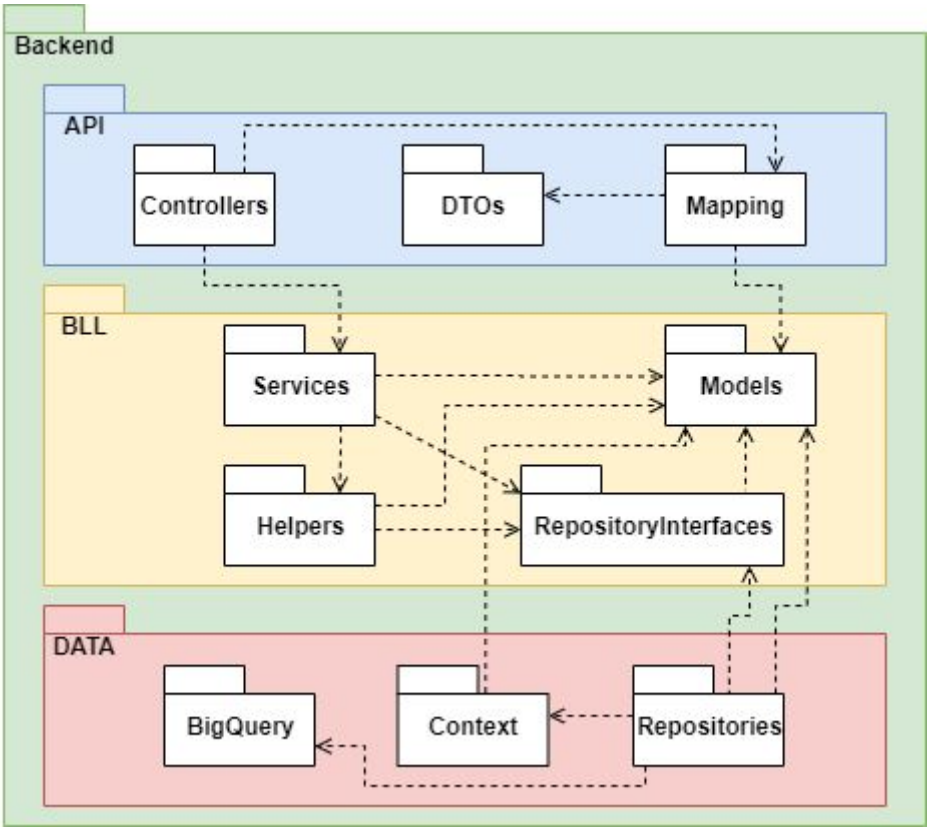
A teljes rendszert ábrázoló absztrakt ábra

Az alkalmazás egy három rétegű architektúrával rendelkezik. Adatbázis ként használja a Google BigQuery adatbázisát, ami tartalmazza a teljes Stack Overflow adat szettet. Emellett egy saját adat bázist is használ ami alkalmazás specifikus és kiegészítő adatokat tárol. A saját adatbázisát MongoDB segítségével valósítja meg. Ez viszont az ASP.NET Core Code First megoldásával van elkészítve tehát a kiegészítő adatbázis bármikor cserélhető az alkalmazás alatt.

A Backend egy ASP.NET Core alapú szerver. Itt a DDD (Domain Driven Design) megközelítést alkalmaztuk. Három rétegre bomlik a backend alkalmazás. A rétegek közül a legértékesebb a BLL, tehát ez teljesen független a többi rétegtől és azok is csak a BLL-től függenek a DDD értelmében.

A klienst typescript-ben React és Redux segítségével készítettük el. Ez magával vonja a React komponensek használatát. A React hook alapú megközelítését alkalmaztuk. Így a komponensek hookok-val vannak megvalósítva. Az állapot tárolására a Reduxot alkalmaztuk így az elkülönül. Ezek mellet egy külön álló csomag foglalkozik a szerver API hívásával.

3.1.1. Backend



A backend package diagram

A backend három rétegre különül el. Minden rétegben függőség befecskendezéssel biztosítjuk a szükséges függőségek meglétét.

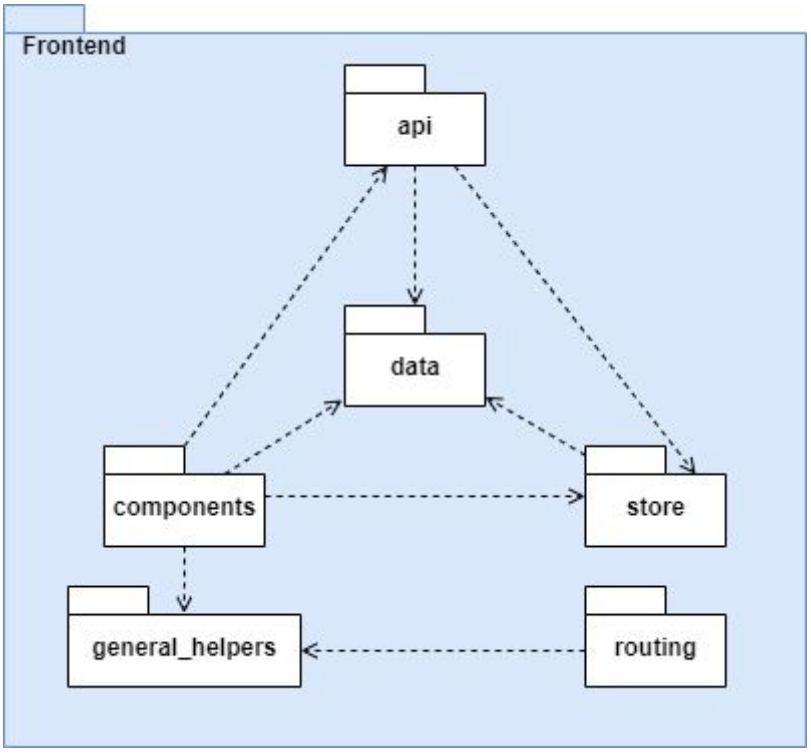
Az API réteg a klienssel való kommunikációért felelős. A szerver a klienssel https kommunikációt folytat. A kliens kérései a kontrollerekhez esnek be. A kontrollerek feldolgozzák, esetleg validálják a bejövő adatot és egy automapper segítségével biznisz objektummá alakítják azt.

A BLL az alkalmazás legfontosabb rétege. Itt szolgáltatások találhatók, amelyek az API rétegtől érkező kéréseket szolgálják ki. Ezek a szolgáltatások a szintén BLL-ben definiált biznisz objektumokon végeznek műveleteket. Vannak műveletek amik adatok adatbázisba való kiírását igénylik vagy onnan való adat beolvasással járnak. Azért, hogy megszüntessük az ebből fakadó függőséget a BLL-ben a DATA réteg felé, a függőség megfordításának elvét alkalmaztuk. A BLL elemei ezek után csak a repository interfészektől függenek és ezek a BLL-en belül találhatók, így a DDD nem sérül. Az utolsó csomag ebben a rétegben a helpereket tartalmazza. Az alkalmazás egyetlen helpere jelenleg a SuggestionsHelpere. Ezt a logikát több szolgáltatásnak is el kell érnie és előnyös ha singleton módon van megvalósítva ezért lett külön kiszervezve.

A DATA rétegben a repository mintát alkalmaztuk. Ez a minta lehetővé teszi, hogy egy általános repository definiálásával el végezzük az összes CRUD műveletet egységesen az összes ebből leszármazó különböző repository-ra. A DATA rétegben két elkülönülő

repository típus van két teljesen független ős repository-val. Az egyik a BigQuerry-hez készült itt csak adatok lekérésére van szükség nem is kell az összes CRUD funkció. A másik pedig minden funkciót megvalósít és a saját MongoDB segéd adatbázisunkhoz tartozó repository-k őse. Ez a két fajta repository elkülönülő kontextusokkal is dolgozik az egyik a Context csomagból szerzi be a MongoDB-hez tartozó kontextust, míg a másik a BigQuery csoagbólszerez kontextust.

3.1.2. Frontend



A frontend package diagram

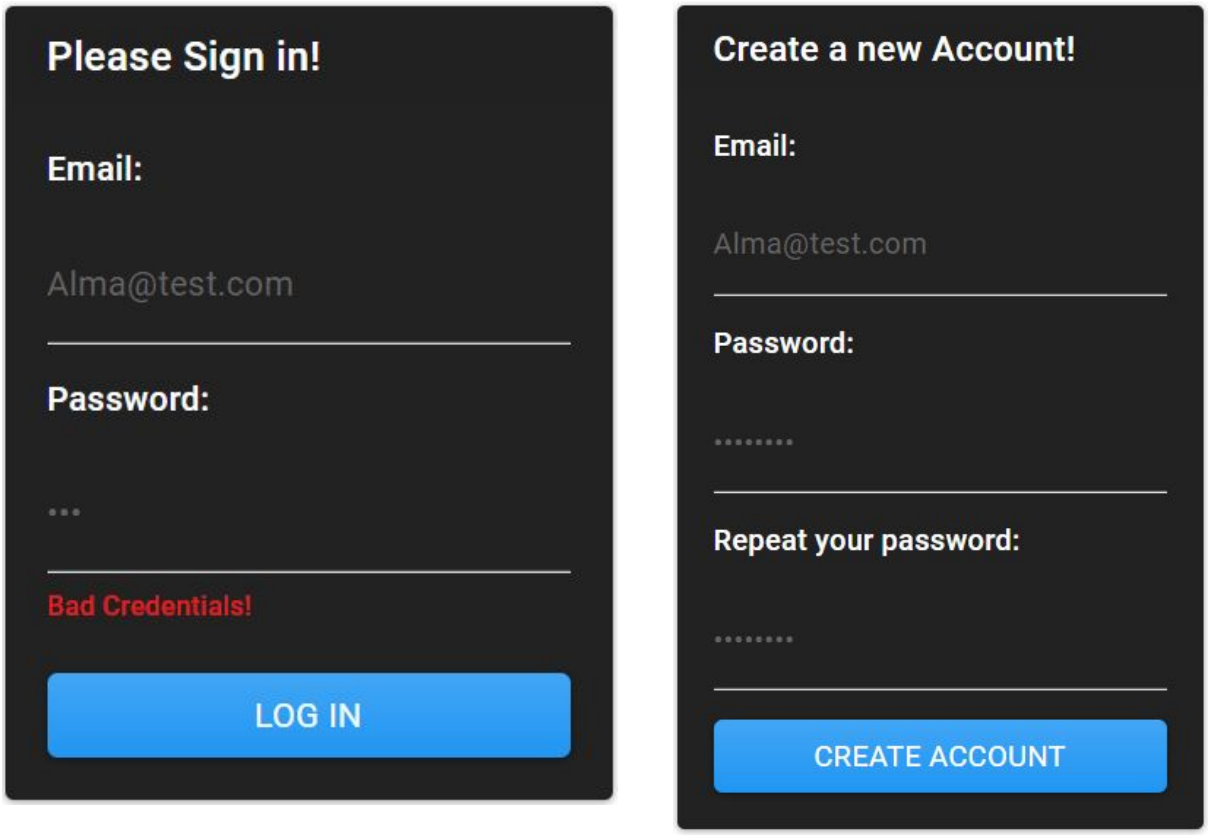
A frontendnek három kiemelkedő csomagja van az api, components és a store. Ebben a három csomagban elkülönülnek a kirajzolandó komponens elemek, amiket react hook-okba szerveztünk, a redux állapot tároló store, amit a Redux-toolkit segítségével valósítottunk meg és az axios-val elkészített api hívások.

A szerver oldalról érkező adat a data csomagban definiált modelleken keresztül érhető el az összes fontosabb csomag számára.

3.2. Megvalósítási ötletek

3.2.1. Authentikáció

Az alkalmazás egy többfelhasználós alkalmazás. Egy bejelentkező képernyővel indít a kliens. Itt vagy be tud jelentkezni a felhasználó vagy regisztrálni tud egy új fiókot.



A belépő és a regisztrációs képernyők

A beléptető UI-hoz a react-hook-form-ot használtuk. Ez egy egyszerűen kezelhető és kliens oldalon is validált form-ot biztosított. Szerver oldalon az AuthController dolgozza fel a bejelentkezést és a regisztrációt. A felhasználók kezeléséhez az ASP.NET Identity rendszerét használjuk, kiegészítve egy MongoDB csomaggal így a felhasználóink egyszerűen kerülnek elmentésre az adatbázisban. A jelszavakat hashelve tároljuk.

Amikor egy felhasználó regisztrál vagy bejelentkezik készítünk számára egy JWT tokent. Ez a token 30 napig érvényes és addig ezzel tudja magát azonosítani a felhasználó. Ezt elmenti a böngésző lokális tárjába a kliens és innentől kezdve amíg a 30 nap le nem jár vagy a felhasználó ki nem lép az alkalmazás bejelentkezve marad. Lejárt JWT token esetén a szerver figyelmezteti a klienst, ami ilyenkor eldobja a lejárt tokent és újra bejelentkezést kér a felhasználótól.

A szerver minden további kontrollere csak bejelentkezés után érhető el. Az identity segítségével a szerver meg tudja állapítani az aktuális felhasználót és a szolgáltatások ellenőrzik, hogy a felhasználó jogosult e olvasni vagy módosítani egy egy adott erőforrást.

3.2.2. SuggestionHelper

Az üzleti logika legfontosabb eleme a javaslatokat előállító SuggestionHelper. Ez egy singleton osztály. A szerver indításakor egyetlen példány jön belőle létre. A konstruktora egy választást kínál fel szerver indításkor hogy be akarunk e tölteni adatokat az adatbázisba vagy már meglévő táblákkal dolgozunk e. Ez azért lényeges, mert jelenleg 30000 posztal dolgozunk és már ezt is lassú fájlból beolvasni.

Az adatbázisban két tábla van amik csak a javaslatok kiszolgálása miatt léteznek. Az egyik a Tag tábla ez a 30000 poszton előforduló összes lehetséges tag-et tartalmazza. Ez azért fontos, mert így ha az alkalmazás ezeket a tag-eket használja fel akkor biztos nem léphet ki a javaslatok által lefedett tag térből. A másik tábla pedig a ClusteredPost tábla. Ez a tábla a 30000 felhasznált post-ot tárolja. A BigQuery-ben használt azonosítót, a poszthoz rendelt clustereket, amiket hierarchikus klaszterezéssel állapítottunk meg, és a poszthoz tartozó tageket tároljuk. Ez utóbbi bár egy a BigQuery-ből is könnyen kinyerhető adat lenne, de a BigQuery elérése nem a leggyorsabb és a javaslatok képzése mint látni fogjuk amúgy sem a leggyorsabb, tehát ahol lehet spórolni kell az idővel.

A javaslat képzés a következőképpen történik:

- Leválasztjuk a beérkező posztok közül az utolsó ötöt. Ezek azok a posztok amiket a felhasználó utoljára látott.
- Megkeressük a legkisebb közös klasztert ezen öt poszt között. Ha nincs öt poszt még a listában akkor a lehető legnagyobb klasztert használjuk azért, hogy elkerüljük, hogy túl korán le szűküljön a javaslatok lehetséges tere.
- Készítünk egy Dictionary-t ennek a klaszternek az elemeiből. ahol a kulcs a poszt azonosító az érték pedig a poszt tag-jeinek listája. Ez a lépés fontos, mert a klaszter elemeinek lekérése a leglassabb lépés a javaslatok elkészítésekor. Ebben a Dictionary-be rendezett formában könnyű és gyors a poszt azonosítóval való indexelés miatt.
- Ezek után csökkenő sorba rendezzük a posztokat az alapján, hogy mennyi közös tag-je van az adott posztnak az öt utoljára nézettel, illetve a szálon beállítottakkal.
- A három elsőt ezek után leválasztjuk és visszaadjuk javaslatként.

A ajánló logika tag hasonlóságot kereső kódja:

```
var orderedPosts = posts.OrderByDescending(postId =>
{
    var tagMatchCountList = mostImportantIncomingPost
            .Select(mII => mII.TagList
            .Intersect(allPostFromCluster[postId])
            .Count()).ToList();
    tagMatchCountList
            .Add(tagsFromThread
                    .Intersect(allPostFromCluster[postId])
                    .Count());
    return tagMatchCountList.Sum(x => x);
});
```

3.2.3. BigQuery elérése

Az alkalmazás a posztoknak csak a BigQuery azonosítóját tárolja. A szöveges adatokat a BigQuery-től szerzi. Ezzel egyrészt lényegesen csökkentjük a saját adatbázisunk méretét és az adatok is folyamatosan frissülnek. A BigQuery-hez saját repositoryt definiáltunk illetve a modellek egy speciális ősből származnak le.

A BQModel rendelkezik egy `FromRow(BigQueryRow row)` függvénnyel. Ez azért fontos mert a BigQuery lekérések `BigQueryRow` típussal térnek vissza. Ennek a függvénynek a segítségével ez mindíg a repository-nak megfelelő modellé alakítható. Például a következőképpen kérjük le az adatokat a BigQuery-től:

```
var rows = bigQuery.GetRows(query);
return rows.Select(r =>
{
    var element = new TEntity();
    element.FromRow(r);
    return element;
}).ToList();
```

A query egy SQL lekérdezés string formában amit BigQuery feldolgoz. Az alkalmazásban mindenhol arra törekszünk, hogy minél kevesebb kérést intézzünk a BigQuery-hez. A kérések kiszolgálásának ideje nagyon változó lehet.

Erre a legjobb példa, amikor betöltünk egy thread-et teljesen a felhasználó számára. Ilyenkor a szál elemei mellet le kell kérjük az aktuális javaslatokat is. A javaslatok kérésére van külön szerver oldali API, de hogy ne kelljen két elkülönülő kérést intézni a BigQuery-hez a szál lekéréséért felelős hívás egyben elkészíti a javaslatokat is és egy csomagban lekérdezi az összes szükséges posztot.

3.2.4. Redux store és Redux toolkit

A kliens alkalmazás állapotának tárolását redux segítségével oldottuk meg. A reduxot pedig a redux-toolkit csomaggal egészítettük ki. Ez a csomag egy jól átlátható hook-okat támogató megközelítést biztosít a redux-hoz.

A redux-toolkit-ből slice-okat használunk. Egy slice összefog egy adott állapotot és az ahhoz kapcsolódó reducer-eket. A slice le generálja a reducerek-hez tartozó akciókat is. Példa a slice-okra:

```
const allTagsSlice = createSlice({
  name: "tags",
  initialState: [] as TagData[],
  reducers: {
    loadAllTags(_state, action) {
      return action.payload.tags;
    },
  },
});
```

A reducereket egy `combineReducers()` függvénnyel kapcsoljuk össze. Az összes reducer összefogásából pedig felépül a teljes store. A store-ba bevezettük a routert is így bármilyen oldal váltás előidézhető és nyomon is követhető azon keresztül.

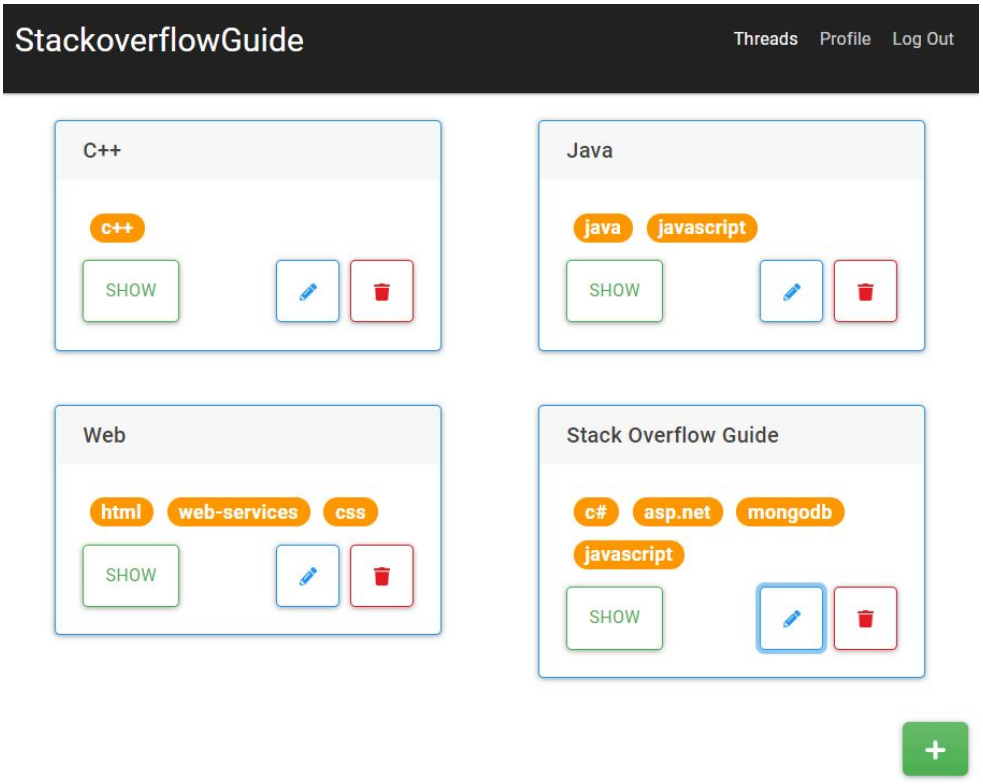
A store-re különböző middleware-ket lehet felvenni. Ilyen a thunk is ami lehetővé teszi a store-ban az aszinkron dispatch-et. Ezt akkor használjuk ki amikor az API hívások eredményeire várunk és betöltjük őket a storeba.

3.3. Az alkalmazás működése

A weboldal megnyitásakor a felhasználónak lehetősége van a bejelentkezésre, illetve a regisztrációra. Miután a felhasználó belépett, a Threads oldal fog megjelenni.

3.3.1. A szálakat tartalmazó oldal

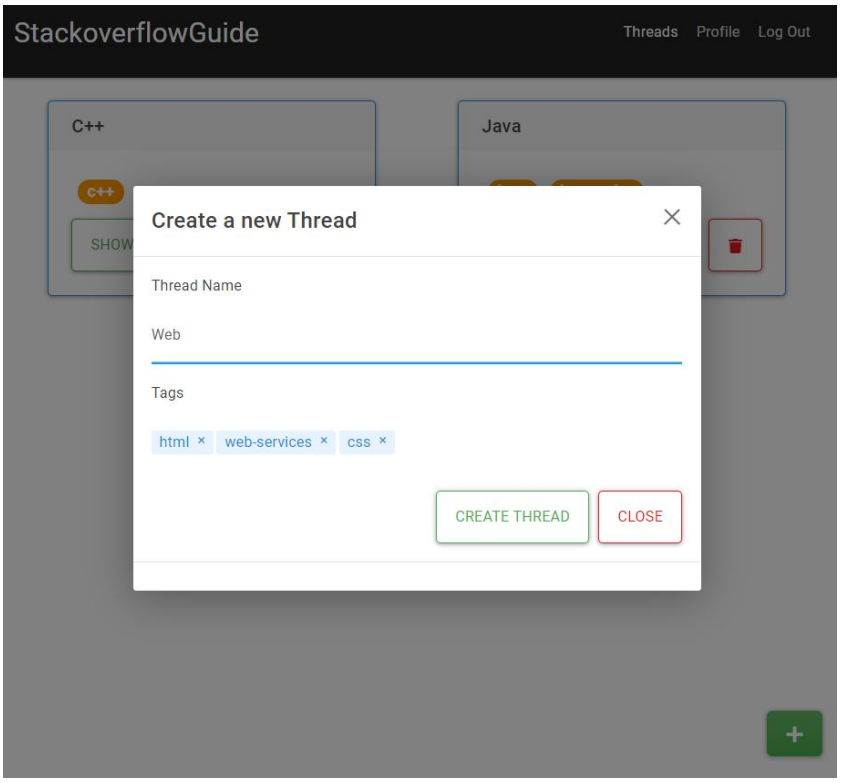
A Threads oldalon láthatja a felhasználó a száljait, amik az egyes témaköreihez tartalmazzák a kérdéseket és az új ajánlásokat. A szálakhoz tartozó tag-eket a narancssárga címkék jelentik.



A felhasználó szálait bemutató oldal

Új szálat az oldal jobb alsó sarkában található, zöld gombra kattintva lehet létrehozni. Ilyenkor feljön egy felugró ablak, ahol a felhasználó megadhatja az új szál címét és kiválaszthatja a szálhoz tartozó tag-eket. A felhasználó a Create Thread gombra kattintva

fogadhatja el az új szálat, a Close gombbal pedig elvetheti. Mindkét esetben vissza kerül a Threads oldalra.



Az új szálat létrehozó felugró ablak

A szálak módosítása a szálak jobb alján lévő, tollat ábrázoló gomb lenyomásával lehetséges. Ekkor feljön az új szál létrehozásához használt felugró ablak, csak a fejléce fog megváltozni az “Edit the Thread” szövegre, illetve a szál eddigi tulajdonságai szerepelni fognak az ablakon. Emelett a Create Thread gomb helyett Save Changes gomb fog szerepelni, aminek megnyomásával a felhasználó elfogadja a változásokat és visszatér a Threads oldalra.

A szálakon a Show gombra kattintva léphetünk át az ajánló oldalra, ahol a szálhoz tartozó kérdéseket és ajánlásokat tekinthetjük meg.

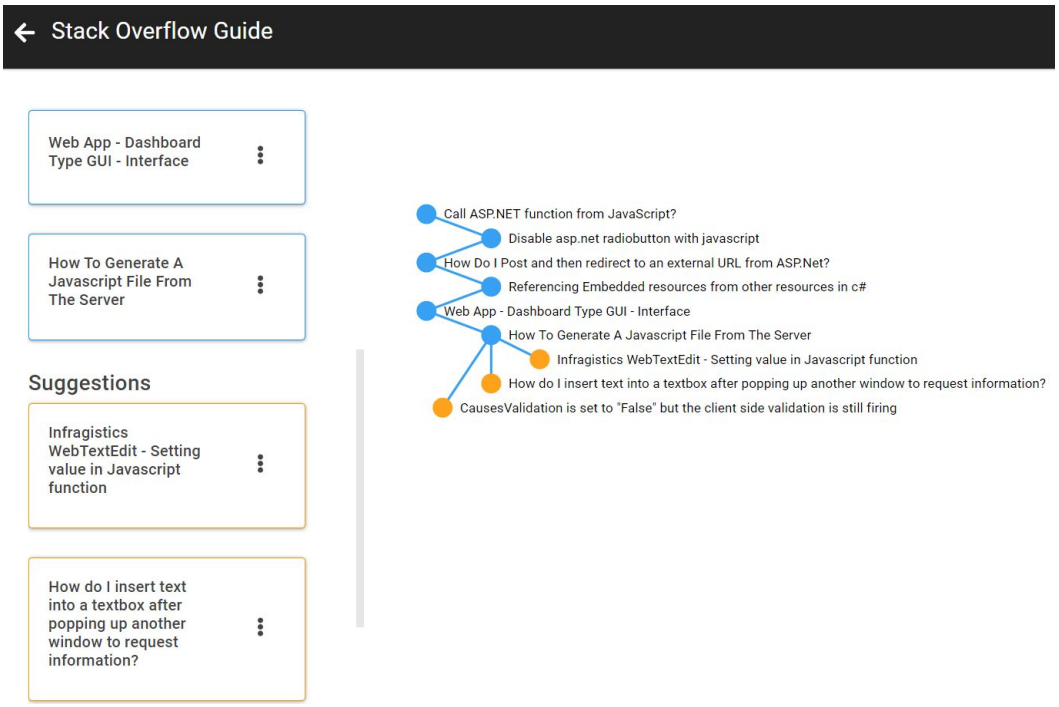
Szálat törölni a szálak jobb alsó sarkában látható, piros kuka gombbal lehet.

3.3.2. Az ajánló oldal

A szálakon a Show gombra kattintva léphetünk át az ajánló oldalra, ahol a szálhoz tartozó kérdéseket és ajánlásokat tekinthetjük meg. Az oldal baloldalán láthatóak a kártyák, amik a kérdések címet mutatják. Felül, kék kerettel vannak a korábbi kérdések, alul narancssárga kerettel pedig az ajánlott kérdések szerepelnek. A kérdések görgethetőek.

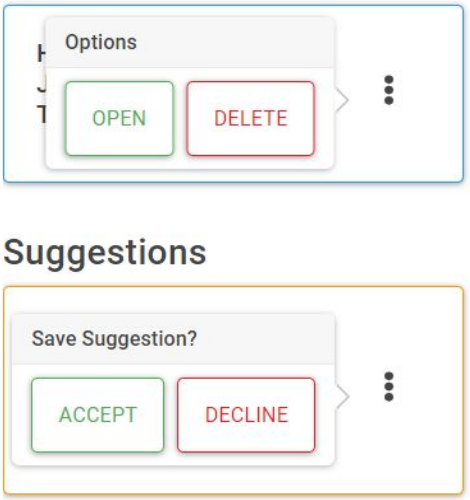
Az oldal jobb oldalán látható egy gráf ábrázolás, ami vizualizálja a korábbi kérdéseket és az új ajánlásokat. A gráf egy vonalra fűzve mutatja, hogy a felhasználó milyen sorrendben tette fel a kérdéseket. A gráf végén találhatók az új ajánlások. A felhasználó

tetszés szerint átrendezheti a gráf pontjainak a helyét, illetve nagyíthatja, kicsinyítheti a gráfot. Egy adott pontra kattintva a baloldali kérdéssor a kiválasztott ponthoz tartozó kérdéshez ugrik és rövid ideig színesen kiemeli.



Az ajánló oldal

A korábbi kérdések oldalán található menüpontra kattintva lenyíló menü, amivel a felhasználó eldobhatja az adott kérdést, ha szeretné kivenni az előzményekből. Hasonló módon az ajánló kérdések esetén ugyanez a felugró menü segítségével lehet elfogadni vagy elutasítani az ajánlást. Miután megtörtént az elfogadás vagy az elutasítás, a weboldal újra generál három ajánlást a felhasználó számára és frissíti a gráfot a változásoknak megfelelően.



A kérdés kezelő menük

4. Jövőbeli tervek

A önálló laboratórium projektünkben még rengeteg potenciál van a fejlődésre. Szeretnénk majd tovább fejleszteni az ajánló rendszerünket és a webalkalmazásunkat is, illetve célunk a Stack Overflow adat szettet minél nagyobb mértékben feldolgozni.

Az ajánló rendszerhez tervezzük majd felhasználni a posztok szövegét és címét. Ezen a szöveg elemek feldolgozásához elsőkörben a Word2Vec módszer alkalmazását gondoltuk. Így a szavakat n dimenziós vektorokká alakítjuk. A szövegfeldolgozás más módjainak is érdemes lehet utánanézni. A tag-ek felhasználása is kapcsolódhatna szorosabban a posztok kategorizálásához, hogy ne csak utólag rangsoroljunk a közös tag-ek alapján.

A posztok dimenzió redukálásához bővebben megvizsgálnánk az autoencoder modelleket, mivel precízebb, testreszabottabb eredményeket érhetünk el vele, mint a UMAP-pal. Az autoencoder-rel a teljes adatot fel tudnánk majd dolgozni. További ajánló módszereket is érdemes lehet megvizsgálni és összehasonlítani őket egymással, hogy minél jobb ajánlásokat tudjunk adni. Az ajánló rendszereknek nagyon sok különböző fajtája létezik hasznos lehet különböző megközelítéseknek is utána járni.

A webalkalmazáshoz sok további funkciót tudnánk még hozzáadni. A felhasználó a profil ablakán megadhatná, hogy milyen ismeretekkel és érdeklődéssel rendelkezik, vagy hogy milyen szintűnek érzi a tudását. Ezek az információk segíthetnek majd, hogy még jobban személyre szabott ajánlásokat tegyünk. A suggestion oldalon megjelenő kérdéseknél tervezünk opciót, hogy a felhasználó megjeleníthesse majd a poszt teljes tartalmát. A felhasználó számára lehetőséget adnánk, hogy ne csak egy ajánlás elfogadásával tudjon hozzáadni egy új posztot, hanem tudjon közvetlenül is újat hozzáadni.