PRIMJENA STROJNOG UČENJA U KONTEKSTU FUNKCIJSKOG JEZIKA HASKELL

Hadžiegrić, Luka

Undergraduate thesis / Završni rad

2019

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Algebra University College / Visoko učilište Algebra**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:225:002358

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-11-29**

ALGEBRA SVEUČILIŠTE

Repository / Repozitorij:

Algebra Univerity - Repository of Algebra Univerity

zir.nsk.hr

dabar
DIGITALNI AKADEMSKI ARHIVI I REPOZITORIJI

VISOKO UČILIŠTE ALGEBRA

ZAVRŠNI RAD

PRIMJENA STROJNOG UČENJA U KONTEKSTU FUNKCIJSKOG JEZIKA HASKELL

Luka Hadžiegrić

Zagreb, Veljača 2019.

„Pod punom odgovornošću pismeno potvrđujem da je ovo moj autorski rad čiji niti jedan dio nije nastao kopiranjem ili plagiranjem tuđeg sadržaja. Prilikom izrade rada koristio sam tuđe materijale navedene u popisu literature, ali nisam kopirao niti jedan njihov dio, osim citata za koje sam naveo autora i izvor, te ih jasno označio znakovima navodnika. U slučaju da se u bilo kojem trenutku dokaže suprotno, spreman sam snositi sve posljedice uključivo i poništenje javne isprave stečene dijelom i na temelju ovoga rada".

U Zagrebu, 26.02.2019.

Predgovor

Ovaj rad posvećujem svojem djedu Josipu Badriću koji na žalost više nije s nama i svojoj baki Ljudmili koja nestrpljivo iščekuje moju diplomu.

Želim se zahvaliti mentoru dr. sc. Goranu Đambiću koji je preuzeo moj rad, svojoj obitelji na podršci, pogotovo svojoj majci Aleksandri te svim svojim učiteljima i profesorima koji su me vodili kroz dosadašnje obrazovanje.

Prilikom uvezivanja rada, Umjesto ove stranice ne zaboravite umetnuti original potvrde o prihvaćanju teme završnog rada kojeg ste preuzeli u studentskoj referadi

Sažetak

Porastom kompleksnosti programskih rješenja funkcijska paradigma postaje sve značajnija u razvoju softvera jer obećava veću modularnost, stabilnost i razumljivost programskog koda.

Ovaj rad istražuje funkcijski pristup izradi softvera kroz izradu fiktivne pametne kuharice i primjenu algoritma strojnog učenja.

Ključne riječi: haskell, strojno učenje, sql, obrada podataka, k-means, aplikacija

Sadržaj

1. Uvod 4

2. Opis projektnog zadatka 5

3. Haskell 6

   3.1. Osnovni koncepti 6

4. Prikupljanje podataka 8

   4.1. Podatkovni model 8

   4.2. Pohrana i čitanje pohranjenih podataka 10

   4.3. Strugači sadržaja 11

   4.4. Raščlanjivanje JSON-a 13

   4.5. Dohvaćanje sadržaja 14

   4.6. Algoritam preuzimanja podataka 16

   4.7. Rezultati prikupljanja podataka 16

5. Obrada prikupljenih podataka 17

   5.1. Izbacivanje nepovoljnih kategorija i njihovih proizvoda 18

   5.2. Stvaranje potpune baze sastojaka sa cijenama i mjernim jedinicama 20

6. Izrada umjetnih podataka 22

   6.1. Profil korisnika 22

   6.2. Funkcija sklonosti 23

   6.3. Stvaranje recepata 27

   6.4. Određivanje favorita 28

   6.5. Stvaranje korisnika 30

7. Algoritam za preporuku recepata 31

   7.1. K-means 31

   7.2. Implementacija 32

7.3. Grupiranje korisnika 34
7.3.1. Jednostavna inicijalizacija 35
7.3.2. K-Means++ inicijalizacija 35
7.4. Provjera kvalitete grupiranja korisnika 37
7.5. Usporedba jednostavne i K-Means++ inicijalizacije 41
8. Korisnička aplikacija 45
8.1. Baza podataka 45
8.2. API server 46
8.3. Korisničko sučelje 46
9. Testiranje 50
Zaključak 52
Popis kratica 53
Popis slika 54
Popis tablica 55
Popis kôdova 56
Literatura 58

1. Uvod

Funkcijska programska paradigma postavlja se kao bolja alternativa *objektno orijentiranom programiranju* (engl. *Object Oriented Programming*, skraćeno OOP) u današnje vrijeme kada je potrebno brzo dostaviti što ispravniji softver. Cilj ovog rada je istražiti tu tvrdnju kroz izradu *pametne* kuharice koja koristeći algoritam strojnog učenja korisnicima predlaže recepte koji bi im se mogli svidjeti.

U drugom poglavlju dan je opis projektnog zadatka koji u kratkim crtama opisuje osnovne funkcionalnosti aplikacije te korake koje je potrebno poduzeti za ispunjenje projektnog zadatka.

Treće poglavlje daje kratak pregled osnovnih koncepata u programskom jeziku Haskell.

Prikupljanje podataka je tema četvrtog poglavlja gdje se obrađuju razni načini za prikupljanje, i pohranu podataka sa Internet servisa.

Peto poglavlje se bavi obradom i pripremom prikupljenih podataka za korištenje u kuharici.

U šestom poglavlju je objašnjen proces izrade umjetnih podataka te simuliranje korisnika aplikacije kako bi se proizveli podaci potrebni za implementaciju inicijalnog algoritma za preporuku recepata.

Vezano na šesto poglavlje, sedmo poglavlje se bavi implementacijom K-means algoritma kao baze za algoritam preporuke. Uz K-means implementirane su i dvije metode inicijalizacije algoritma koje su međusobno uspoređene.

Izrada korisničke aplikacije obrađena je u osmom poglavlju.

Deveto poglavlje bavi se testiranjem te je dan kratak pregled korištenih metoda testiranja napisanog softvera.

Podaci o sastojcima za kuharicu preuzeti su sa Coolinarike[1] i Konzum Klik[2] servisa.

[1] https://coolinarika.com
[2] https://www.konzum.hr/klik/

2. Opis projektnog zadatka

Kao što je u uvodu bilo spomenuto, cilj rada je napraviti kuharicu koja koristi algoritam strojnog učenja za predlaganje recepata te kroz taj proces istražiti funkcijsko programiranje.

Takvi algoritmi očito zahtijevaju podatke nad kojima bi mogli raditi te je potrebno **prikupiti podatke** o sastojcima, **stvoriti nasumične recepate** te **simulirati lažne korisnike** i njihovo služenje kuharicom.

Podatke je potrebno prikupiti sa dva izvora. Sa Coolinarike je potrebno prikupiti osnovne informacije o sastojcima dok je sa Konzum Klik servisa potrebno prikupiti podatke o tržišnim cijenama pojedinih sastojaka. Podatke sa ta dva izvora je potrebno spojiti u jedan set podataka koji će služiti za inicijalizaciju kuharice.

Kuharica je zamišljena kao Internet aplikacija te njen korisnik mora biti u mogućnosti obavljati sljedeće radnje:

1. Registracija, prijava i odjava korisnika
2. Pregled popisa sastojaka i detalja pojedinog sastojka
3. Pregled popisa recepata i detalja pojedinog recepta
4. Bilježenje zanimljivih recepata
5. Stvaranje tjednog jelovnika
6. Pregled potencijalno zanimljivih novih recepata

3. Haskell

Haskell je napredan i lijen čisto funkcijski programski jezik čija je prva verzija objavljena prije gotovo trideset godina (Wadler P. et al., 2007) te je kao takav odabran za ovaj rad koji se između ostalog bavi i proučavanjem funkcijske programske paradigme.

Iako fokus ovog rada nije na teoriji kategorija, bitno je ukratko spomenuti da je ona efektivno matematička osnova Haskella. Riječ je o općenitoj matematičkoj teoriji struktura i sustava struktura koja sve češće svoju primjenu nalazi u računarstvu te nam u principu nudi formalni set oblikovnih obrazaca i apstrakcija sa poznatim svojstvima.

3.1. Osnovni koncepti

Haskell trenutno nije naročito popularan programski jezik te je iz tog razloga u nastavku dan vrlo sažet pregled nekih osnovnih koncepata.

Lijenost (engl. *laziness*) je svojstvo jezika da ne evaluira izraze dok njihove vrijednosti nisu stvarno potrebne. To svojstvo između ostalog omogućava rad sa beskonačnim listama te koncizniji i elegantniji kod.

Čistoća (engl. *purity*) zahtjeva da izrazi uvijek moraju rezultirati nekom vrijednošću te da pri tome ne smiju imati sporednih efekata, npr. ispaliti nuklearne rakete prilikom zbrajanja dva broja.

Referencijalna transparentnost (engl. *referential transparency*) se veže uz čistoću te znači da će neki izraz uvijek biti evaluiran u istu vrijednost. Ovo svojstvo u kombinaciji sa čistoćom omogućava kompajleru da provodi poprilično agresivne optimizacije.

Slijedeći koncepti dolaze iz teorije kategorija te se mogu smatrati standardnim sučeljima (engl. *interfaces*) u Haskellu.

Monoid je ukratko komutativna te lijevo i desno asocijativna binarna operacija sa neutralnom vrijednosti. Njegovo sučelje se sastoji od binarnog operatora `<>` te vrijednosti `mempty` koja predstavlja neutralnu vrijednost.

Funktor (engl. *functor*) daje `fmap` funkciju (alias `<$>`) inače poznatu kao `map` te služi za primjenjivanje neke funkcije nad npr. elementima liste ali je koncept puno generalniji te se može primijeniti na bilo koju strukturu koja *omata* neki drugi podatkovni tip.

Aplikativni funktor (engl. *applicative functor*, skraćeno aplikativ) je nadogradnja na funktor te nam daje sučelje koje omogućava stavljanje vrijednosti unutar neke strukture putem funkcije `pure` te primjenu funkcije unutar neke strukture na vrijednost unutar druge strukture istog tipa putem operatora `<*>` npr. primjena liste funkcija na listu nekih vrijednosti. Iako to možda nije na prvi pogled očito, aplikativ predstavlja paralelno izvršavanje radnji.

Monada (engl. *monad*) je jedan od najbitnijih koncepata te za razliku od aplikativa predstavlja sekvencijalno izvršavanje radnji. Osim toga, monada je vrlo svestrana te nam između ostalog daje takozvanu `do` sintaksu koja omogućava pisanje koda u imperativnom stilu te opisivanje efekata na čist način (pristup bazi, pisanje u datoteku, izmjena varijable[3]).

Jedan od načina korištenja monade bi bio definiranje konteksta unutar kojeg se može pojaviti `null` vrijednost. Unutar tog konteksta moguće je pretvarati se da `null` vrijednost ne postoji te se potencijalna `null` vrijednost treba obraditi samo prilikom izlaska iz tog konteksta što uvelike pojednostavljuje kod te povećava njegovu kvalitetu. Takav kontekst implementira tzv. `Maybe` monada.

Pretvarač monada (engl. *monad transformer*) rješava problem kombiniranja monada koje se same po sebi ne miješaju lako. Vrlo često želimo vršiti neke radnje koje osim što mogu rezultirati `null` vrijednošću mogu i npr. pristupiti bazi, ili uhvatiti grešku. Svi ti efekti su u principu zasebne monade te ih je potrebno spojiti u jednu. To se postiže na način da se željene monade naslažu u *slojeve* te se putem sučelja pretvarača pomičemo između tih slojeva u željeni kontekst.

[3] Haskell nema koncept varijable već se promjena stanja simulira stvaranjem novih vrijednosti baziranih na starim.

4. Prikupljanje podataka

Haskell je izuzetno dobar jezik za raščlanjivanje (engl. *parsing*) i obradu podataka. Jak tipski sustav sprječava velik dio grešaka pri obradi podataka i omogućuje donekle automatizirano pisanje programa. Osim toga, svojstva jezika kao što su funkcije višeg reda i monade omogućuju definiranje vrlo moćnih *monadskih kombinatora raščlanjivača* (engl. *monad parser combinators*) koji omogućuju jednostavno izvlačenje podataka iz tekstualnog oblika.

U svrhu prikupljanja podataka sa Coolinarike i Konzum Klik servisa korištene su programske zbirke *Selda*, *Scalpel*, *Aeson* te *Servant* i *Servant-Client*, te su podaci spremani u *SQLite*[4] bazu podataka budući da je njen format zapisa podataka izrazito lako prenosiv.

4.1. Podatkovni model

Podatkovni model je napravljen korištenjem *algebarskih podatkovnih tipova* (engl. *algebraic data types*, skraćeno ADT). Zovu se *algebarski* jer se njihove definicije mogu shvatiti kao *sume* ili *produkti*.

```
data Unit = L | Kg | Pce
  deriving ( Eq, Show, Generic )
```

Kôd 4.1 Definicija mjerne jedinice.

U Kôd 4.1 dana je definicija koja je primjer čistog sumarnog podatkovnog tipa. Ona govori da vrijednost tipa `Unit` može biti ili *Litra* ili *Kilogram* ili *komad* te je ukupan broj mogućih vrijednosti $1 + 1 + 1 = 3$.

```
data IngredientRow = IngredientRow
  { id          :: ID IngredientRow
  , name        :: Text
  , slug        :: Slug
  , image       :: Image
  , description :: Text
  } deriving ( Eq, Show, Generic, SqlRow )
```

Kôd 4.2 Definicija retka u tablici sastojaka

[4] Jednostavna i lako prenosiva baza podataka, implementirana kao programska zbirka.

U Kôd 4.2 prikazan je produktni podatkovni tip koji predstavlja redak u tablici baze o nekom sastojku preuzetom sa Coolinarike. Konstruktori (engl. *constructors*) podatkovnih tipova mogu u sebi sadržavati više *polja* te se u principu ne trebaju imenovati već je dovoljno samo navesti tipove podataka. Ipak, u ovom slučaju su navedeni nazivi polja konstruktora te se takav podatkovni tip obično naziva zapis (engl. *record*). S obzirom da je `IngredientRow` produktni tip, ukupan broj njegovih vrijednosti se računa na slijedeći način $n(ID) * n(Text) * n(Slug) * n(Image) * n(Text) = n(IngredientRow)$ gdje je $n(a)$ ukupan broj vrijednosti nekog podatkovnog tipa a.

```
newtype ID a = ID
  { unID :: Int
  } deriving ( Eq, Show, Generic )
```

Kôd 4.3 Definicija tipa primarnog ključa

Kod definicije tipa primarnog ključa (engl. *primary key*) dane u Kôd 4.3 može se primijetiti *fantomska tipska varijabla* (engl. *phantom type variable*) `a` koja se ne pojavljuje nigdje u konstruktoru vrijednosti sa desne strane jednakosti. U ovom slučaju ona služi kako bi kompajleru naznačili da neka funkcija radi sa primarnim ključevima npr. sastojka te ne može primiti kao ulaznu vrijednost npr. primarni ključ proizvoda što sprječava potencijalne greške.

U ovom primjeru pojavljuje se i ključna riječ `newtype`. Razlika između `newtype` i dosadašnjeg `data` koji je korišten za definiranje novih podatkovnih tipova je u tome što `data` u principu u sebi sadrži pokazivače (engl. *pointers*) na vrijednosti dok `newtype` omata konkretnu vrijednost te postoji samo za vrijeme faze kompiliranja gdje služi za dodjeljivanje ispravnih instanci tipskih klasa (engl. *type class*) i provjeru tipova, te se nakon faze kompiliranja efektivno briše što je poznato kao brisanje tipova (engl. *type erasure*). Osim toga, `newtype` smije imati samo jedno polje i jedan konstruktor vrijednosti.

Uz `data` i `newtpe` u dosadašnjim primjerima možemo primijetiti i ključnu riječ `deriving` koja služi za automatsko pisanje / derivaciju instanci određenih tipskih klasa. U Kôd 4.2 deriviraju se instance za jednakost (`Eq`) i tekstualni prikaz (`Show`) no uz njih se definira i instanca za generičko programiranje nad podatkovnim tipom (`Generic`) koja omogućava derivaciju tipske klase za serijalizaciju i deserijalizaciju podataka (`SqlRow`) iz baze koju pruža programska zbirka *Selda*.

4.2. Pohrana i čitanje pohranjenih podataka

Za komunikaciju sa *SQLite* bazom podataka korištena je programska zbirka *Selda*. Kako bi bilo moguće pisati upite u *strukturiranom upitnom jeziku* (engl. *Structured Query Language*, skraćeno SQL) unutar Haskella potrebno je, osim instanci klasa `SqlRow` i `SqlType` za korištene podatkovne tipove, definirati i vrijednosti koje će predstavljati tablice baze podataka u kodu.

```
ingredients :: Table IngredientRow
ingredients = table "Ingredient"
  [ #id   :- autoPrimary
  , #name :- unique
  , #slug :- unique
  ]
```

Kôd 4.4 *Selda* definicija tablice sastojaka

Tablica se definira uz pomoć funkcije `table` koja prima naziv tablice u tekstualnom obliku te listu parova polja / stupaca i njihovih atributa i ograničenja (engl. *constraints*) što se može vidjeti u Kôd 4.4.

```
createIngredient :: ( MonadCatch m, MonadSelda m )
  => Ingredient
  -> m ( Either SeldaError ( ID IngredientRow ) )
createIngredient Ingredient{..} = try $ insertWithPK
  ingredients [ IngredientRow{ id = def, .. } ]
```

Kôd 4.5 *Selda* funkcija za pohranu novog sastojka u bazi

Sa vrijednosti `ingredients` moguće je definirati funkciju uz pomoću koje se nova vrijednost sprema u bazu podataka što je prikazano u Kôd 4.5.

```
findIngByName :: ( MonadCatch m, MonadSelda m )
  => Text
  -> m ( Either SeldaError ( Maybe Ingredient ) )
findIngByName n = try
  $ fmap ( fmap convert . listToMaybe ) $ query $ do
    ingredient <- select ingredients
    restrict $ ingredient ! #name .== literal n
    pure ingredient
    where convert IngredientRow{..} = Ingredient{..}
```

Kôd 4.6 *Selda* upit za pronalazak sastojka prema imenu

Funkcija `createIngredient` je relativno jednostavna te se *ugrađeni domeni specifičan jezik* (engl. *Embedded Domain Specific Language*, skraćeno EDSL) koji pruža *Selda* puno jasnije vidi na primjeru funkcije `findIngByName` definirane u Kôd 4.6 gdje se funkcijom `select` odabiru sastojci koji se zatim sa funkcijom `restrict` ograničavaju samo na one sastojke kojima je ime jednako parametru `n`.

Iz *tipskog potpisa* (engl. *type signature*) funkcije `createIngredient` vidljivo je da se rezultat vraća unutar neke monade `m` u kojoj je moguće hvatati greške i pristupati bazi putem *Selde* što indiciraju ograničenja `MonadCatch` i `MonadSelda`.

Rezultat se vraća omotan u `Either` i `Maybe` podatkovni tip što indicira da operacija može vratiti grešku (`SeldaError`) ili će **možda** pronaći traženi rezultat u bazi (`Maybe Ingredient`).

```
data Maybe a = Nothing | Just a
data Either a b = Left a | Right b
```

Kôd 4.7 Definicije `Maybe` i `Either` podatkovnih tipova

`Maybe` i `Either` su u principu tipski konstruktori (engl. *type constructor*) te se ne smatraju *pravim* podatkovnim tipovima dok se ne popune njihove tipske varijable, isto kao što `Just`, `Left` i `Right` nisu prave vrijednosti za razliku od `Nothing` dok u sebi ne sadrže neku konkretnu vrijednost.

4.3. Strugači sadržaja

Podaci o sastojcima sa Coolinarike dostupni su samo u obliku *hipertekstualnog označnog jezika* (engl. *hypertext markup language*, skraćeno HTML) tako da je potrebno definirati raščlanjivač koji će iz HTML-a izvući željeni sadržaj. Takav raščlanjivač je još poznat pod nazivom strugač (engl. *scraper*). Za definiranje strugača korištena je programska zbirka *Scalpel* koja pruža niz kombinatora za izradu strugača.

Sa Coolinarike je potrebno prvo preuzeti popis svih sastojaka u obliku pročišćenih naziva često zvanih s*lug*. Ti će nazivi zatim biti korišteni kako bi se dohvatio HTML sa detaljima pojedinog sastojka. Naziv sastojka „Čokolada za kuhanje“ ima *slug* „cokolada-za-kuhanje“.

Slug je format u kojem su izbačeni svi naglasci sa slova, velika slova su promijenjena u mala te su razmaci pretvoreni u znak minus. Slugovi su na HTML-u sa popisom sastojaka već pripremljeni samo ih treba strugačem izvući iz poveznice koja vodi do punog opisa sastojka.

```
newtype Slug = Slug
  { unSlug :: Text
  } deriving ( Eq, Show, Generic )

slugsS :: Scraper Text [ Slug ]
slugsS = do
  uris <- attrs "href"
    $ "h3" @: [ hasClass "h5", hasClass "title" ] // "a"
  pure $ ( Slug . prep ) <$> uris
  where prep = replace "/" mempty
             . replace "/namirnica/" mempty
```

Kôd 4.8 Definicija strugača popisa *slugova*

U Kôd 4.8 definiran je strugač `slugsS`. *Scalpel* pruža podatkovni tip `Scraper` sa dva tipska parametra od kojih prvi određuje na kojem tipu podataka radi a drugi određuje koji je rezultat izvršenog struganja.

Scalpelov EDSL je relativno sličan selektorima *kaskadnih stilova* (engl. *Cascading Style Sheets*, skraćeno CSS) tako da je moguće jednostavno protumačiti kako `slugS` izlučuje `href` atribute iz svih *anchor* oznaka (engl. *tag*) koje se nalaze unutar `h3` oznake sa klasama `h5` i `title`.

```
ingredientS :: Image -> Scraper Text Ingredient
ingredientS image = do
  name <- text $
    AnyTag @: [ "id" @= "content_header" ] // "h1"
  slug <- fmap fx $
    attr "content" $ "meta" @: [ "property" @= "og:url" ]
  description <- attr "content" $
    "meta" @: [ "property" @= "og:description" ]
  pure Ingredient{..}
  where fx = Slug
           . replace "/" mempty
           . replace "https://www.coolinarika.com/namirnica/"
             Mempty
```

Kôd 4.9 Definicija strugača sastojka

Strugač sastojaka `ingredientS` u Kôd 4.9 definiran je nešto drugačije od `slugS` strugača. `Scraper` je monada, no ne podržava ulazno izlazne (engl. *input output*, skraćeno IO) radnje. S obzirom na to da sastojci sadrže i slike, te je slike potrebno preuzeti ranije prije

nego strugač bude primijenjen na HTML od sastojka. Iz tog razloga je `ingredientS` definiran kao funkcija koja prima sliku te vraća strugač kao rezultat.

4.4. Raščlanjivanje JSON-a

Za razliku od Coolinarike Konzum Klik servis ima API koji daje odgovore u JavaScript notaciji objekata (engl. *JavaScript Object Notation*, skraćeno JSON). JSON odgovore potrebno je raščlaniti u konkretne Haskell zapise što je moguće uz korištenje *Aeson* programske zbirke.

```
{ "name"              : "Šampanjac Dom Perignon 0,75 l"
, "statistical_price" : "1,933.32 Kn/L"
, "categories"        :
  [ { "name" : "Cat 01" }
  , { "name" : "Cat 02" }
  , { "name" : "Cat 03" }
  ]
}
```

Kôd 4.10 Primjer JSON odgovora sa Konzum Klik API-ja za pretragu

Primjer JSON-a od jednog proizvoda dobivenog sa API-ja za pretragu dan je u Kôd 4.10. Iako originalni JSON sadrži puno više informacija ovdje su istaknuta samo bitna polja.

```
data Product = Product
  { name :: Text
  , unit :: Unit
  , cost :: Word
  , cats :: [ Category ]
  } deriving ( Eq, Show, Generic )
```

Kôd 4.11 Definicija zapisa proizvoda

Taj JSON proizvoda je raščlanjen u zapis `Product` definiran u Kôd 4.11. Ukoliko nazivi polja u zapisu i u JSON-u odgovaraju jedan na jedan, *Aeson* može automatski napraviti instancu tipske klase `FromJSON`, no budući da to ovdje nije slučaj te da je potrebno dodatno obraditi polje `statistical_price` kako bi se iz njega izvukli cijena i mjerna jedinica u zasebna polja instanca je napisana ručno u Kôd 4.12.

```
instance FromJSON Product where
  parseJSON = withObject "Product" $ \ o -> do
    name <- o .: "name"
```

```
    unit <- fmap unitP $ o .: "statistical_price"
    cost <- fmap costP $ o .: "statistical_price"
    cats <- o .: "categories"
    pure Product{..}
    where unitP :: Text -> Unit
          unitP sp = let [_, u] = prepp sp in readUnit u
          costP :: Text -> Word
          costP sp = let [p, _] = prepp sp in read $ unpack p
          prepp :: Text -> [Text]
          prepp = splitOn " "
                . replace "Kn/" mempty
                . replace ","   mempty
                . replace "."   mempty
                . strip
```

Kôd 4.12 Definicija FromJSON instance za proizvod

4.5. Dohvaćanje sadržaja

Za dohvaćanje podataka sa interneta korištene su programske zbirke *Servant* i *Servant Client*. *Servant* pruža niz *kombinatora tipske razine* (engl. *type level combinators*) koji služe za definiranje serverskog *sučelja za programiranje aplikacija* (engl. *application programming interface*, skraćeno API), dok *Servant Client* omogućuje automatsko stvaranje funkcija za konzumiranje tog API-ja.

```
type CoolinarikaIngredientListAPI
  =  "recepti"
  :> "namirnice"
  :> Get '[HTML] ( Either SlugError [ Slug ] )
type CoolinarikaImageAPI
  = "images"
  :> Capture "d1"     Text
  :> Capture "d2"     Text
  :> Capture "image"  Text
  :> Get     '[IMAGE] Image
type CoolinarikaIngredientAPI
  = "namirnica"
  :> Capture "ingredient_slug" Slug
  :> Get '[HTML]
     ( Manager -> IO ( Either IngredientError Ingredient ) )
```

Kôd 4.13 Tipski potpisi API-ja Coolinarike

Tipski potpisi dani u Kôd 4.13 opisuju bitne dijelove Coolinarike, tako tip `CoolinarikaIngredientListAPI` u principu predstavlja slijedeću poveznicu:

```
https://coolinarika.com/recepti/namirnice/
```

dok tip `CoolinarikaIngredientAPI` predstavlja slijedeći **predložak** poveznice:

```
https://coolinarika.com/namirnica/{ingredient_slug}
```

gdje je `{ingredient_slug}` potrebno zamijeniti sa konkretnim `Slug`-om.

U ovom slučaju `Get` kombinator iz *Servanta* kaže da se ovom djelu API-ja pristupa sa GET zahtjevom te da kao odgovor možemo očekivati HTML iz kojeg će biti izlučena funkcija koja prima `Manager` te izvodi neku `IO` akciju koja može vratiti traženi sastojak ili grešku.

```
instance MimeUnrender HTML ( Either SlugError [Slug] ) where
  mimeUnrender _ bs = do
    let html = parseTags
               $ decodeUtf8With ignore $ toStrict bs
    pure $ case scrape slugsS html of
      Nothing -> Left SlugParseFailed
      Just ss -> Right ss
```

Kôd 4.14 MimeUnrender instanca tipske klase za listu Slugova

Kako bi *Servant* znao ispravno protumačiti odgovor sa servera potrebno je napisati instancu tipske klase `MimeUnrender` koja pruža funkciju `mimeUnrender` koja određuje kako pretvoriti dobiveni tip sadržaja.

U primjeru Kôd 4.14 dana je instanca `MimeUnrender` klase za povratni tip sadržaja HTML kojeg treba interpretirati kao `Either SlugError [ Slug ]`. Budući da `mimeUnrender` funkcija prima dobiveni sadržaj u obliku `ByteString` potrebno ga je pretvoriti u `Text` na koji se zatim može primijeniti strugač `slugS` definiran u Kôd 4.8.

Pisanje `MimeUnrender` instanci za sadržaj sa Konzum Klik API-ja nije potrebno, budući da *Servant* već ima `MimeUnrender` instancu za sve tipove podataka koji implementiraju `FromJSON` te ima slijedeći potpis:

```
instance FromJSON a => MimeUnrender JSON a
```

Servant Client implementira funkciju `client` koja kao argument prima tip API-ja te kao rezultat vraća funkciju za konzumiranje tog API-ja.

```
ingredientListC :: ClientM [ Slug ]
```

```
ingredientListC = client $ Proxy @ CoolinarikaIngredientAPI
```

4.6. Algoritam preuzimanja podataka

Pošto je potrebno na neki način povezati podatke sa Coolinarike i Konzum Klik-a, nakon definiranja svih potrebnih podatkovnih tipova, tipskih klasa i funkcija implementiran je slijedeći algoritam korišten za preuzimanje informacija o sastojcima u funkciji koja:

1. Otvara vezu sa SQLite bazom podataka
2. U bazi stvara potrebne tablice za čuvanje podataka o sastojcima
3. Preuzima popis pročišćenih naziva sastojaka (Slug-ova) sa Coolinarike
4. Iz baze preuzima popis već obrađenih sastojaka (u slučaju da je prijašnji pokušaj preuzimanja bio prekinut)
5. Izrađuje novi popis sastojaka koji se ne nalaze u bazi i koje je potrebno preuzeti
6. Preuzima i sprema jedan po jedan sastojak sa liste sastojaka u bazu podataka
7. Preuzima popis svih kategorija koje postoje na Konzum Klik servisu i sprema ih u bazu podataka
8. Za svaki sastojak preuzet sa Coolinarike pretražuje Konzum Klik putem API-ja po imenu sastojka te rezultate pretrage sprema u bazu

4.7. Rezultati prikupljanja podataka

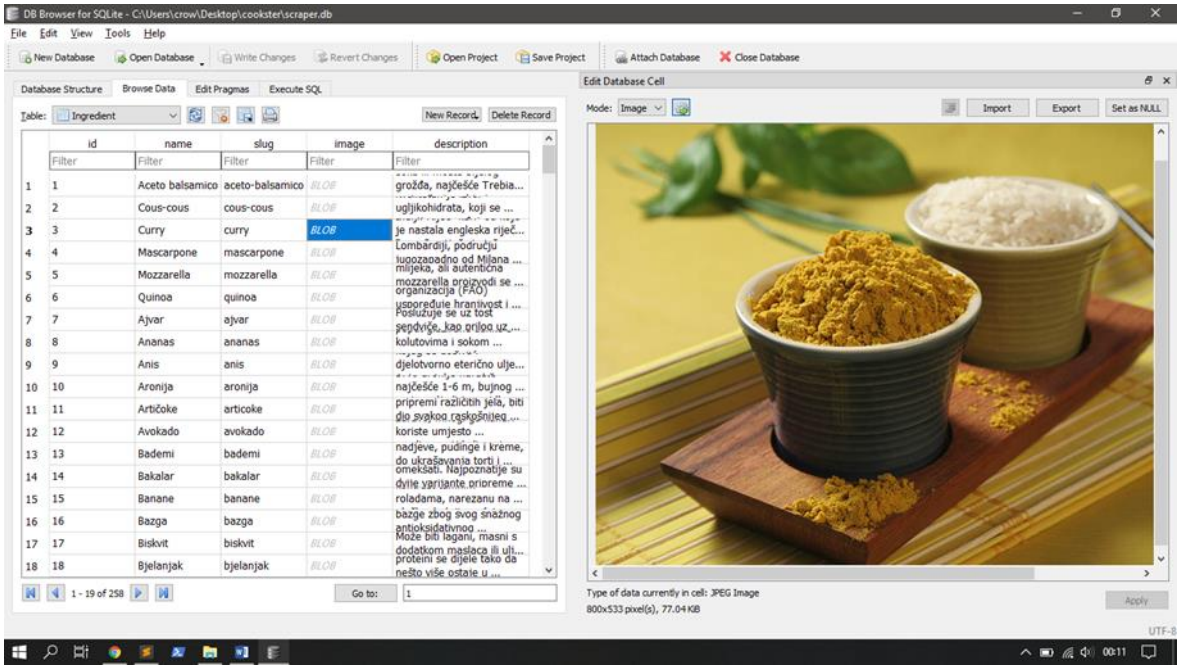
Izvršavanje funkcije za prikupljanje podataka o proizvodima nazvane `executeDataScraper` trajalo je otprilike pet minuta te je rezultiralo sa prikupljenih **258** sastojaka, **201** kategorija proizvoda te **3107** prikupljenih proizvoda.

5. Obrada prikupljenih podataka

Podaci sa Coolinarike i Konzum Klik servisa prikupljani su u svrhu sastavljanja baze podataka o raznim sastojcima koja sadrži njihov:

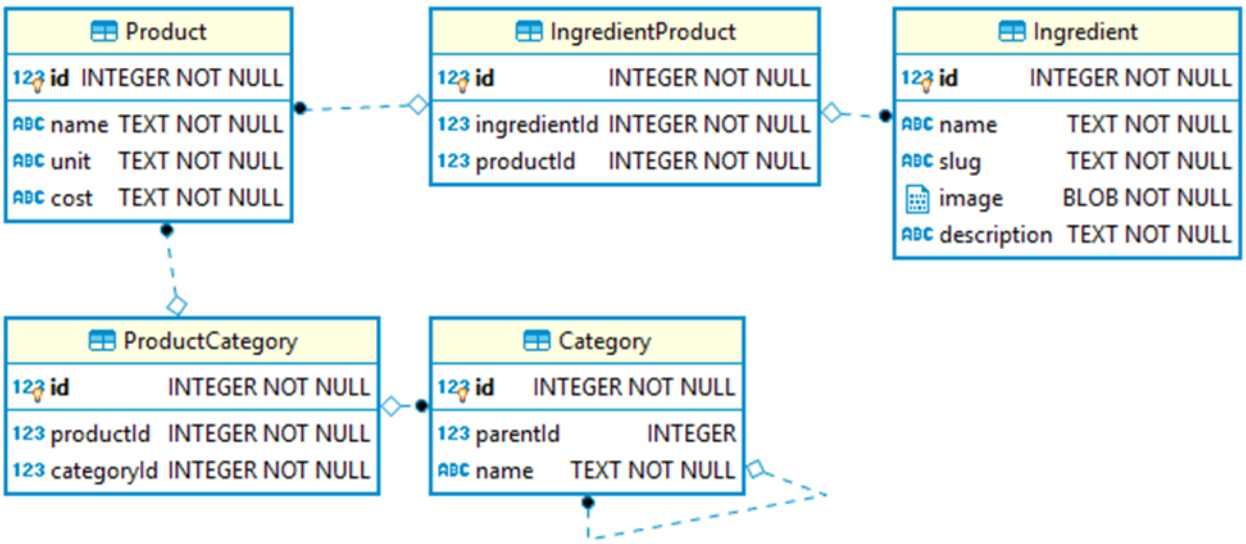
- naziv
- sliku
- kratak opis
- približnu tržišnu cijenu
- mjernu jedinicu

Prije izrade takve baze iz prikupljenih podataka, podatke je bilo potrebno prekontrolirati i očistiti od nevaljalih podataka. Taj je postupak odrađen ručno uz pomoć alata za upravljanje *SQLite* bazama podataka naziva *DB Browser for SQLite*. Ovaj alat je vrlo praktičan te u sebi sadrži i preglednik slika, tako da je moguće pregledavati i prikupljene slike koje su u bazi spremljene u BLOB formatu.



Slika 5.1 Prikaz pregleda prikupljenih podataka u programu *DB Browser for SQLite*

Nakon ručne inspekcije tablice sa sastojcima zaključak je da su prikupljeni podaci zadovoljavajuće kvalitete te ne sadrže nikakve greške. Ipak, prije daljnje obrade podataka korisno je pogledati shemu baze koja sadrži prikupljene podatke kako bi se dobila bolja slika o relacijama između podataka.



Slika 5.2 Relacijska shema baze prikupljenih sastojaka

5.1. Izbacivanje nepovoljnih kategorija i njihovih proizvoda

Budući da su podaci o proizvodima prikupljani kroz API za pretraživanje, prikupljeni su i mnogi proizvodi koji su nazivom donekle slični nazivu pojedinog sastojka ali nisu ono što se tražilo. Na primjer, ukoliko su traženi proizvodi koji odgovaraju sastojku *Čokolada* kao rezultati su mogli biti dobiveni i *Čokoladno mlijeko* ili neki deterdžent za pranje rublja sa aromom čokolade i vanilije. Ideja je da stvari poput deterdženata mogu biti eliminirane prema kategoriji, dok stvari poput čokoladnog mlijeka mogu biti eliminirane prema očekivanoj mjernoj jedinici sastojka. U ovom slučaju očekivana mjerna jedinica čokolade bila bi Kilogram, dok se čokoladno mlijeko očito mjeri u Litrama.

Kategorije proizvoda trebaju biti ručno provjerene i obrisane kroz sučelje alata za upravljanje *SQLite* bazama podataka, no budući da strani i primarni ključevi čuvaju integritet baze podataka, potrebno je napraviti neke izmjene u shemi. Idealno bi bilo kada bi se sve što je ovisno o identifikatoru neke kategorije samo izbrisalo iz baze podataka. Takva opcija postoji u SQLite standardu no dostupna je samo prilikom stvaranja tablice te ne može biti naknadno dodana kroz `alter` SQL naredbu. *Selda* na žalost trenutno nema mogućnost definirati takvu shemu tako da je to problem.

Način na koji se ovaj problem može riješiti je da:

1. Isključimo provjeravanje stranih ključeva

2. Preimenujemo staru tablicu
3. Stvorimo novu tablicu koja ima istu definiciju i naziv kao i stara prije preimenovanja samo uz dodanu `cascade` opciju
4. Kopiramo podatke iz stare tablice u novu
5. Ponovno omogućimo provjeravanje stranih ključeva

Opisani postupak moguće je izvršiti slijedećim SQL naredbama:

```
pragma foreign_keys=off;
alter table "Category" rename to "OldCategory";
create table "Category"
  ( "id" integer not null primary key autoincrement unique
  , "parentId" integer
  , "name" text not null
  , foreign key ( "parentId" ) references "Category" ( "id" )
    on delete cascade
  );
insert into "Category" select * from "OldCategory";
drop table "OldCategory";
pragma foreign_keys=on;
```

Kôd 5.1 Primjer dodavanja `on delete cascade` opcije na već postojeću *SQLite* tablicu

Postupak prikazan u Kôd 5.1 napravljen je za tablice `Category`, `ProductCategory` i `IngredientProduct` kako bi se očuvao integritet baze u slučaju brisanja podataka iz tablice kategorija ili proizvoda.

Sa trenutnim stanjem baze ukoliko se izbriše kategorija nekih proizvoda samo se briše veza između te kategorije i njezinih proizvoda. S obzirom da je cilj ukloniti i proizvode iz nepovoljnih kategorija, nije dovoljno ukloniti samo proizvode koji nakon brisanja kategorija nisu dodijeljeni niti jednoj kategoriji, budući da jedan proizvod može spadati u više kategorija (od kojih jedna može biti i poželjna). Kako bi se obrisali i takvi proizvodi, potrebno je prije brisanja kategorije obrisati i sve proizvode koji spadaju u tu kategoriju. Za tu svrhu stvoren je slijedeći okidač definiran u Kôd 5.2.

```
create trigger "Category_Delete_Trigger"
before delete on "Category" for each row
begin
  delete from "Product"
  where id in
```

```
    (select productId from ProductCategory where categoryId =
old.id);
end;
```

Kôd 5.2 Okidač za brisanje pripadajućih proizvoda pri brisanju kategorije

Nakon ovih promjena na bazi izbrisan je poveći broj kategorija koje iz praktičnih razloga nisu navedene, te je ukupan broj proizvoda u bazi podataka pao sa **3107** na **1971**.

5.2. Stvaranje potpune baze sastojaka sa cijenama i mjernim jedinicama

Iako su neodgovarajući proizvodi većinom izbačeni iz baze, potrebno je još jednom filtrirati rezultate na osnovi mjerne jedinice. Kao što je ranije bilo spomenuto, sa sastojkom *Čokolada* mogu biti vezani proizvodi imena *Čokolada za kuhanje* što je u principu poželjno, no isto tako će i proizvod kao *Čokoladno mlijeko* biti vezan za *Čokoladu*. Ovdje se očekuje da je mjerna jedinica za čokoladu **Kilogram** a za čokoladno mlijeko **Litra** tako da je prema tom kriteriju moguće dodatno filtrirati nepoželjne rezultate.

Za početak je potrebno napisati upit (**Pogreška! Izvor reference nije pronađen.**) koji će povezati sastojke sa proizvodima te grupirati rezultate po atributima sastojka i po mjernoj jedinici povezanih proizvoda te izračunati prosječnu cijenu sastojka iz cijena proizvoda po mjernoj jedinici.

```
select
  I.id, I.name, cast( avg(P.cost) as int ) as cost, P.unit
from Ingredient as I
inner join IngredientProduct as IP on IP.ingredientId = I.id
inner join Product as P on IP.productId = P.id
group by I.id, I.name, P.unit
```

Kôd 5.3 Pomoćni SQL upit za dobivanje liste sastojaka sa cijenama i mjernim jedinicama

Tablica 5.1 Uzorak rezultata pomoćnog upita

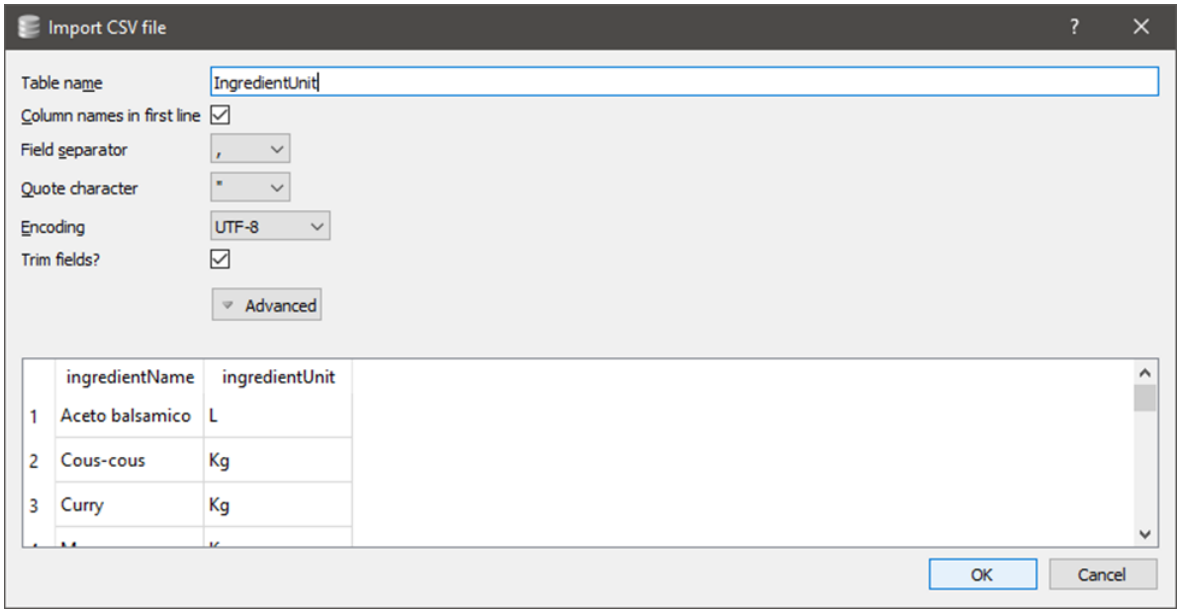
253	Vrhnje	2815	Kg
253	Vrhnje	4301	L
254	Vrhnje za kuhanje	4062	Kg

254	Vrhnje za kuhanje	4178	L

U uzorku Tablica 5.1 Uzorak rezultata pomoćnog upita može se jednostavno zaključiti da su željeni rezultati u litrama. Također, usporedbom ovih rezultata sa konkretnim proizvodima na Konzum Klik servisu dolazi se do zaključka da izračunate cijene donekle odgovaraju realnim cijenama.

Idući korak je ručno napraviti popis naziva sastojaka te očekivane mjerne jedinice. Ovaj rezultat može se spremiti u datoteku zarezom odvojenih vrijednosti (engl. *comma separated values*, skraćeno CSV) koju je onda praktično uređivati u Excelu kako bi se stvorio popis. Taj popis se uvozi nazad u bazu kako bi se mogao iskoristiti za filtriranje neželjenih rezultata u zadnjem upitu koji će poslužiti za stvaranje potpune baze podataka o sastojcima.

CSV datoteka naziva `IngredientUnit.csv` koja sadrži očekivane mjerne jedinice za pojedine sastojke je uvezena kroz sučelje programa *DB Browser for SQLite* kao nova tablica.



Slika 5.3 Stvaranje tablice mjernih jedinica sastojaka putem sučelja programa DB Browser for SQLite

Sa tim podacima moguće je napraviti upit koji će sve podatke povezati u jednu cjelinu koja će poslužiti kao osnova za glavnu aplikaciju. Rezultati tog upita izvezeni su u novu SQLite bazu kao zasebna tablica kako bi se ubrzalo pristupanje podacima.

6. Izrada umjetnih podataka

Značajka aplikacije za predlaganje recepata ovisi o postojećim korisničkim podacima na osnovu kojih se predlažu recepti. S obzirom da je aplikacija tek u začetku te da trenutno ne postoji dovoljan broj korisnika za efektivan rad algoritma strojnog učenja potrebno je napraviti simulaciju korištenja aplikacije kako bi algoritam imao na čemu učiti.

Takvi podaci naravno neće biti jednako dobri kao i pravi podaci o korisnicima, ali uz nekoliko pretpostavki moguće je napraviti dovoljno dobre podatke kako bi se napravila inicijalna verzija algoritma za predlaganje recepata.

Algoritam za predlaganje recepata ovisi o korisnicima i o receptima koje su zabilježili u svoju privatnu kuharicu (prema njima zaključuje koji bi se drugi recepti mogli sviđati korisniku).

Dakle, u procesu izrade umjetnih podataka potrebno je stvoriti recepte, korisnike te simulirati dodavanje pojedinih recepata u privatne kuharice korisnika.

6.1. Profil korisnika

Način na koji algoritam za predlaganje recepata radi je da grupira slične korisnike, stvori popis spremljenih recepata od svih korisnika te zatim svakom korisniku nudi proizvode koji su u toj listi ali nisu u njegovoj privatnoj kuharici (budući da korisnik već zna za te recepte).

Kako bi bilo moguće grupirati slične korisnike osmišljen je koncept profila. Profil u sebi sadrži popis svih sastojaka u bazi te za svaki sastojak ima zabilježenu *sklonost*. Sklonost je vrijednost između 0 i 1.

U svrhu isprobavanja rezultata simulacije stvorena je beskonačna lista naziva sastojaka.

```
type IngredientName = Text

ingredientNames :: [ IngredientName ]
ingredientNames = pack <$> zipWith (<>)
  ( repeat "Ing" ) ( printf "%02d" <$> ( [1..] :: [ Int ] ) )
```

Kôd 6.1 Beskonačna lista naziva sastojaka

Pokušaj ispisivanja vrijednosti `ingredientNames` rezultira beskonačnim nizom testnih naziva sastojaka, npr. `[ Ing001, Ing002, Ing003, …]`. Ovo je vrlo praktična

primjena Haskellove lijenosti jer se beskonačna lista evaluira samo do dijela kojeg koristimo. Iz beskonačne liste možemo jednostavno uzeti npr. deset elemenata sa naredbom `take 10 ingredientNames`.

```
type Preference = Double
type Profile = ( StdGen, Map IngredientName Preference )

profile :: [ IngredientName ] -> Rand StdGen Profile
profile ins = do
  generator <- getGen
  ( c, o )  <- biasParameters $ fromIntegral $ length ins
  pure
   (generator, Map.fromList $ zip ins (bias c o <$> [1,2..]))

profiles :: [ IngredientName ] -> [ Rand StdGen Profile ]
profiles = repeat . profile
```

Kôd 6.2 Generator profila

Profil (`Profile`) je uređeni par nasumičnog generatora brojeva (engl. *random number generator*) `StdGen` koji je korišten za stvaranje profila te rječnika (engl. *dictionary*) `Map` koji za svaki naziv sastojka `IngredientName` čuva sklonost / `Preference` profila ka tom sastojku.

6.2. Funkcija sklonosti

Pojedine sklonosti nisu nasumično određene već ih određuje funkcija sklonosti `bias` definirana u Kôd 6.3 prema rednom broju pojedinog sastojka. Ono što je nasumično određeno su parametri funkcije koji određuju njezin krajnji oblik.

```
bias
  :: Double -- ^ number of elements per cycle / wave
  -> Double -- ^ offset on the @x@ axis
  -> Double -- ^ @x@
  -> Double -- ^ @y@
bias ec os x
  = 0.5 * ( sinc $ 2 * pi * ( 1 / ec ) * ( x - os ) ) + 0.5

biasParameters
  :: Double
  -> Rand StdGen ( Double, Double )
```

```
biasParameters n
  = (,) <$> pure ( n / 4 ) <*> getRandomR ( 0, n )
```

Kôd 6.3 Definicija funkcije sklonosti i funkcije za nasumično generiranje inicijalnih parametara

Funkcija sklonosti nastala je iz potrebe za što realnijim i jednostavnim modelom korisnika kojeg je lako vizualizirati i usporediti sa drugim profilima.

Raditi vizualizaciju u prostoru do tri dimenzije je relativno lako, no za potrebe algoritma za preporuku sastojaka korisnik je predstavljen kao točka u n-dimenzionalnom prostoru gdje broj dimenzija ovisi o ukupnom broju sastojaka u bazi podataka.

Kao rješenje koje zadovoljava navedene potrebe postavila se funkcija *sinc* koja je dana formulom 1.

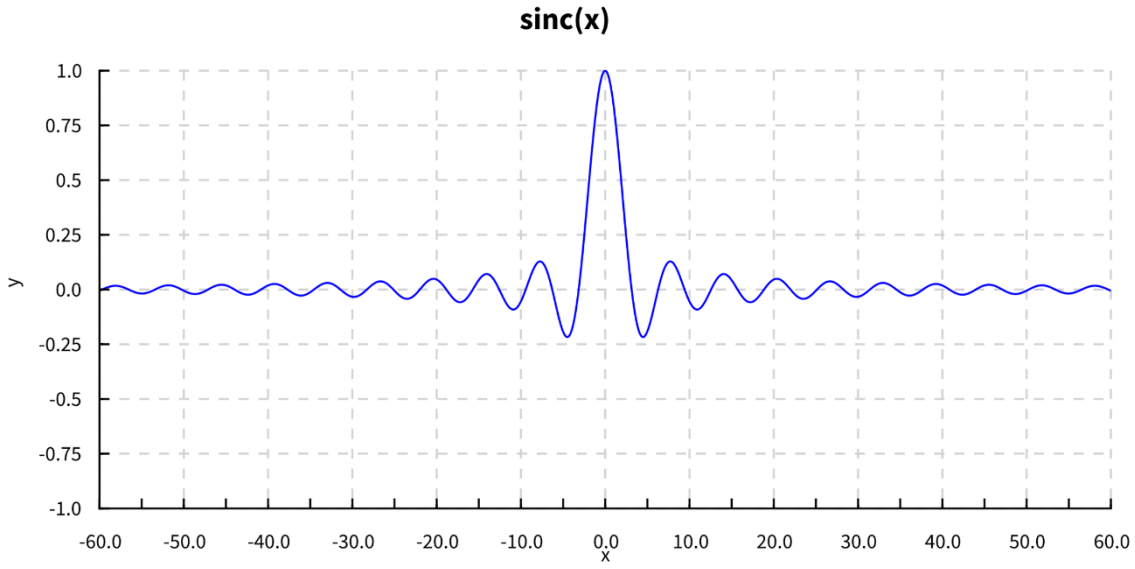
$$sinc(x) = \begin{cases} \frac{\sin(x)}{x}, x \neq 0 \\ 1, x = 0 \end{cases} \quad (1)$$

Te je u Haskellu definirana na slijedeći način:

```
sinc :: ( Eq a, Floating a ) => a -> a
sinc 0 = 1
sinc x = sin x / x
```

Kôd 6.4 `sinc` funkcija

Pretpostavka je da za nekog korisnika očekujemo da će imati određen broj sastojaka koje jako preferira kao i one koje izrazito ne preferira ali će postojati i popriličan broj sastojaka za koje nema neko izrazito mišljenje. Ukoliko pozornije i s namjerom promotrimo graf funkcije `sinc` onda ćemo primijetiti da ju potencijalno možemo iskoristiti za opisivanje sklonosti korisnika jer ima tražena svojstva.



Slika 6.1 Graf `sinc` funkcije

Za graf prikazan u Slika 6.1 možemo zamisliti da *x* os predstavlja sastojke, dok *y* os predstavlja sklonost korisnika nekom sastojku.

Kao što se vidi iz grafa, postoje mnogi brjegovi i dolovi. Dol predstavlja manju sklonost dok brijeg predstavlja veću sklonost. Također je moguće primijetiti da se brjegovi i dolovi smanjuju kako se odmiču od najvećeg brijega te se približavaju nuli.

Ideja je da se ispod najvećeg brijega na osi *x* nalaze sastojci koje pojedini profil korisnika preferira, dok se ispod najdubljih nalaze oni koje ne preferira.

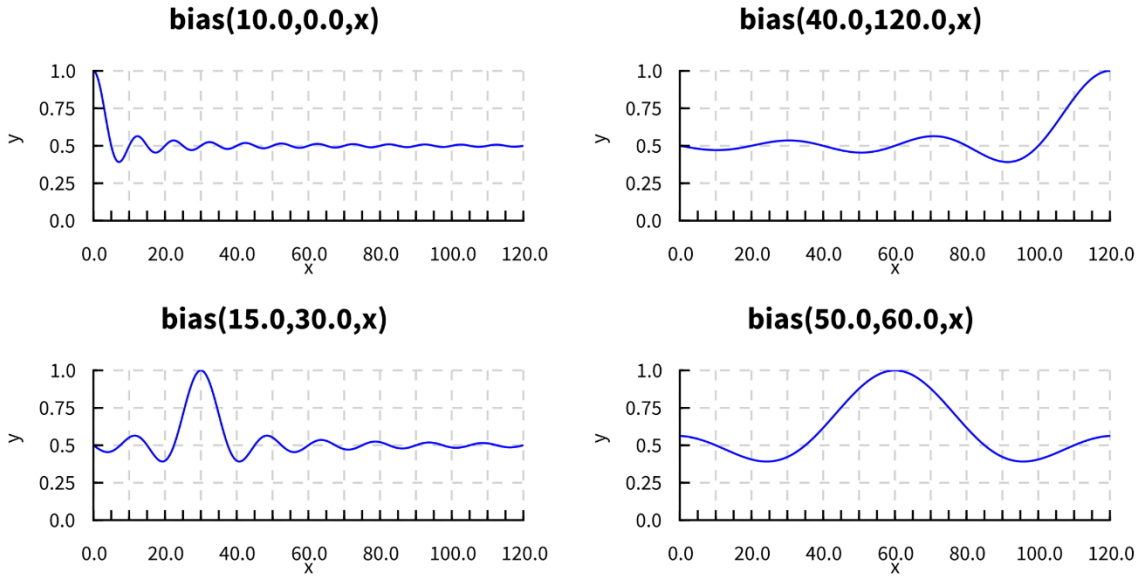
Budući da se vrijednost sklonosti kreće u rasponu od 0 do 1 potrebno je primijeniti neke transformacije nad funkcijom `sinc` kako bi se ponašala ispravno, također je potrebno i dodati par argumenata koji će služiti za stvaranje varijacija funkcije `sinc`. Kako bi se zadovoljile te potrebe osmišljena je slijedeća formula:

$$bias(n, o, x) = 0.5 * sinc\left(\frac{2\pi(x - o)}{n}\right) - 0.5 \quad (2)$$

```
bias
  :: Double -- ^ number of elements per cycle / wave
  -> Double -- ^ offset on the @x@ axis
  -> Double -- ^ @x@
  -> Double -- ^ @y@
bias ec os x = 0.5 * ( sinc $ 2 * pi * ( 1 / ec ) * ( x - os
) ) + 0.5
```

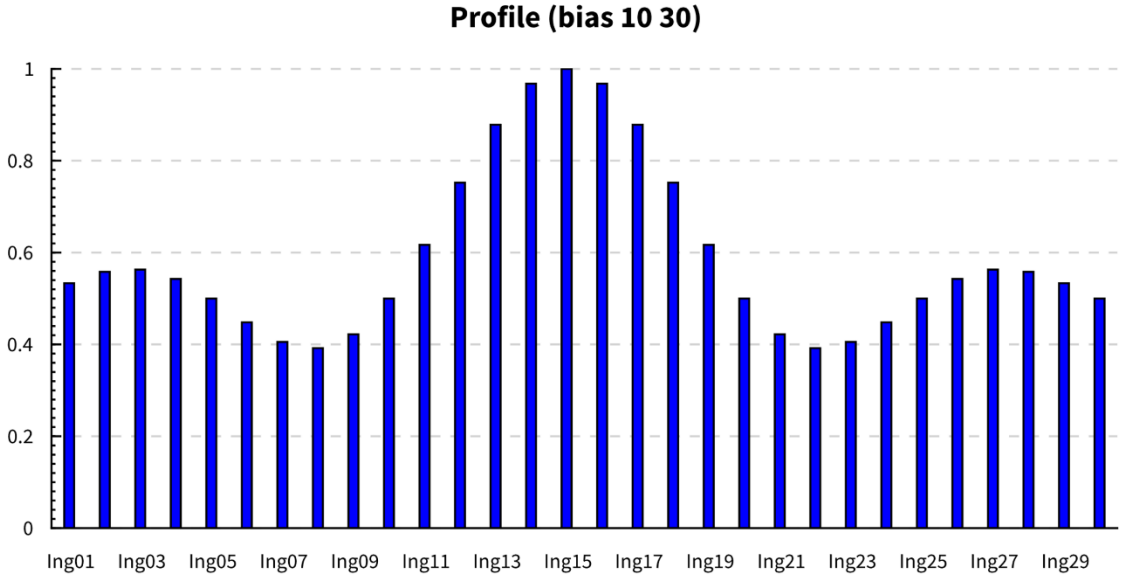
Kôd 6.5 *bias* implementacija u Haskellu

Funkcija `bias` implementirana u Kôd 6.5 se koristi na način da se prvo provede parcijalna primjena (engl. *partial application*) argumenata `ec` i `os` te se zatim dobivena funkcija kojoj je ostao samo jedan argument dalje primjenjuje na neki skup vrijednosti `x`.



Slika 6.2 Primjeri grafova funkcije `bias` sa različitim argumentima

Grafovi prikazani u Slika 6.2 u principu predstavljaju profil korisnika, no tu su ipak u pitanju kontinuirane vrijednosti dok je potpuni profil skup diskretnih vrijednosti, odnosno parova naziva sastojaka te korisničke sklonosti prema tom sastojku.



Slika 6.3 Graf profila

Na grafu u Slika 6.3 prikazani su nazivi sastojaka te sklonosti za pojedini sastojak koja je prikazana kao stupac određene visine. Neki nazivi sastojaka su izostavljeni kako bi stali na graf.

6.3. Stvaranje recepata

Recepti bi u principu mogli biti sasvim nasumično generirani, no budući da je cilj stvoriti podatke koji se mogu grupirati potrebno je primijeniti malo pametniji pristup.

Stvoreni recepti biti će predloženi umjetnim korisnicima te će ti umjetni korisnici morati odlučiti da li da spreme taj recept ili ne. Kako bi povećali šansu da se među stvorenim receptima nađu recepti koji se sviđaju određenom profilu korisnika novi recepti biti će stvoreni prema sklonostima određenog profila.

```
type Amount = Double
type RecipeSchema = Map IngredientName Amount

recipeSchema :: Profile -> Rand StdGen RecipeSchema
recipeSchema ( _, p ) = do
  let biased = Map.keys $ Map.filter (>=0.7) p
  shuffled <- shuffle biased
  selected <- getRandomR ( 2, 6 )
  let ingredients = take selected shuffled
  fmap Map.fromList $ sequence $ fmap amount ingredients
  where amount i = getRandomR ( 0.1, 3 ) >>= pure . (i,)

recipeSchemas :: Profile -> [ Rand StdGen RecipeSchema ]
recipeSchemas = repeat . recipeSchema
```

Kôd 6.6 Generator sheme recepta

U primjeru Kôd 6.6 definiran je podatkovni tip `RecipeSchema` koji je u principu samo alias za rječnik naziva sastojaka i količina tih sastojaka koje idu u recept. Funkcija `recipeSchema` prima profil korisnika te iz njega stvara shemu recepta.

Shema se stvara na način da se iz popisa sastojaka profila izvuku svi sastojci za koje profil ima sklonost veću ili jednaku `0.7`. Ta se probrana lista sastojaka miješa te se iz primiješane liste sastojaka uzima prvih `2` do `6` sastojaka (što znači da će stvoreni recept imati između 2 i 6 sastojaka). Nakon toga, svakom odabranom sastojku pridjeljuje se nasumična količina čiji se raspon vrijednosti kreće između `0.1` i `3` te se napokon stvara shema recepta.

6.4. Određivanje favorita

Jedina aktivnost korisnika koja će se simulirati je spremanje recepata u favorite odnosno osobnu kuharicu. Kako bi to bilo moguće potrebno je definirati funkciju koja određuje sklonost profila određenom receptu / shemi recepta.

```
preference :: Profile -> RecipeSchema -> Preference
preference ( _, p ) s = sum wghtd / sum s
  where wghtd = merge dropMissing dropMissing
                ( zipWithMatched $ const (*) ) p s
```

Kôd 6.7 Funkcija sklonosti profila korisnika ka nekom receptu

Funkcija `preference` definirana u Kôd 6.7 prima profil i shemu recepta te kao rezultat vraća sklonost profila shemi recepta. Budući da profil sadrži rječnik naziva sastojaka sklonosti profila pojedinom sastojku a shema recepta sadrži rječnik naziva sastojaka i njihovih količina, ukupna sklonost profila nekom receptu računa se na način da se prvo pomnože količine sastojaka sa sklonostima sastojaka te se rezultat sumira i podijeli sa sumom količina svih sastojaka.

Funkcija `preference` se može jednostavnije opisati slijedećom formulom:

$$preference = \frac{\sum_{s \in S} k(s) * p(s)}{\sum_{s \in S} k(s)} \tag{3}$$

Gdje S predstavlja skup sastojaka u nekom receptu, $k(s)$ predstavlja količinu sastojka s u receptu a $p(s)$ sklonost profila ka sastojku s.

Pošto sada postoji način za određivanje sklonosti profila pojedinome receptu moguće je provjeriti da li funkcija `recipeSchema` definirana u Kôd 6.6 uistinu stvara recepte koji su pogodniji određenim profilima.

```
averageBiasedPreference
  :: Int                    -- ^ number of ingredients
  -> Int                    -- ^ number of profiles
  -> Int                    -- ^ number of recipes per profile
  -> Rand StdGen Preference -- ^ average preference
averageBiasedPreference ni np nr = do
  ps <- sequence $ take np $ profiles $
        take ni ingredientNames
  fmap average $ forM ps $ \p -> do
    schemas <- sequence $ take nr $ recipeSchemas p
```

```
    pure $ average $ fmap ( preference p ) schemas

averageNonBiasedPreference
  :: Int               -- ^ number of ingredients
  -> Int               -- ^ number of profiles
  -> Int               -- ^ number of recipes per profile
  -> Rand StdGen Preference -- ^ average preference
averageNonBiasedPreference ni np nr = do
  ps <- fmap ( zip [1::Int ..] )
    $ sequence
    $ take np
    $ profiles
    $ take ni ingredientNames
  schemapairs <- forM ps $ \( i, p ) -> do
    schemas <- sequence $ take nr $ recipeSchemas p
    pure ( i, schemas )
  pure $ average $ flip fmap ps $ \( i, p ) ->
    let schemas = concat $ fmap snd $ filter ( (/=i) . fst )
                  schemapairs
    in  average $ fmap ( preference p ) schemas
```

Kôd 6.8 Funkcije za provjeru kvalitete stvorenih recepata

Kako bi provjerili funkcija `recipeSchema` stvarno stvara recepte skrojene po mjeri profila u Kôd 6.8 definirane su funkcije `averageBiasedPreference` i `averageNonBiasedPreference`.

`averageBiasedPreference` stvara niz profila te za svaki od profila stvara određen broj recepata prilagođenih tom profilu. Nakon što je to napravljeno, računa se prosječna sklonost profila za sve njegove recepte te se ta vrijednost vrača kao rezultat.

`averageNonBiasedPreference` stvara niz profila i za svaki od njih stvara određen broj recepata prilagođenih tom profilu, no za razliku od `averageBiasedPreference` ona računa prosječnu sklonost profila za sve recepte **osim** za one koji su njemu prilagođeni.

Ukoliko funkcija `recipeSchema` stvara ispravne recepte, očekivali bi da su vrijednosti koje generira `averageNonBiasedPreference` (skraćeno `anbp`) manje od onih koje generira `averageBiasedPreference` (skraćeno `abp`).

Tablica 6.1 Mjerenja rezultata funkcija `abp` i `anbp`

ni	**np**	**nr**	**abp ni np nr = bp'**	**anbp ni np nr = np'**	**bp' > np'**
50	5	10	0.90	0.59	TRUE
50	5	20	0.90	0.60	TRUE
50	10	10	0.89	0.56	TRUE
100	10	10	0.89	0.56	TRUE
100	10	20	0.89	0.56	TRUE
100	15	10	0.88	0.55	TRUE

Napravljen je niz mjerenja funkcija `abp` i `anbp` te se u Tablica 6.1 vidi da profil korisnika uistinu ima veću sklonost receptima koji su specijalno napravljeni za njega.

6.5. Stvaranje korisnika

Simulirani korisnici su u principu uređeni par profila i odabranih recepata te su ti podaci uvezeni u produkcijsku bazu podataka u koraku inicijalizacije.

```
isPrefered :: Profile -> RecipeSchema -> Rand StdGen Bool
isPrefered p s = getRandomR(0,1)>>= pure . (<=preference p s)
```

Kôd 6.9 Funkcija za odlučivanje o odabiru recepta

Jedina bitna funkcija vezana uz stvaranje korisnika je `isPrefered` definirana u Kôd 6.9 koja odabire nasumičnu vrijednost između 0 i 1 te gleda da li je sklonost korisnika određenom receptu manja ili jednaka toj nasumičnoj vrijednosti.

Ovaj element nasumičnosti je bitan kako bi se spriječilo da svi korisnici napravljeni iz istog profila ne odaberu iste recepte.

7. Algoritam za preporuku recepata

Algoritam za preporuke ne treba biti pretjerano precizan niti je njegova svrha da sa sto postotnom točnošću predviđa koji se recepti sviđaju korisnicima. Ideja je zapravo da korisniku ponudi nešto što bi mu se moglo svidjeti ali da ga i upozna sa nečim novim za što možda još niti ne znamo da li bi mu se svidjelo.

Princip na kojem funkcionira algoritam za preporuku recepata je da prvo grupira sve *slične* korisnike zajedno. U ovom slučaju sličnost se određuje prema receptima koje su pojedini korisnici spremili u svoje osobne kuharice. Nakon što su korisnici grupirani, za svaku grupu se stvara zajednički popis svih recepata koje su korisnici u grupi spremili, zatim se svakom od korisnika u grupi nude recepti sa ukupnog popisa uz izuzetak onih koji se nalaze u njegovoj osobnoj kuharici.

7.1. K-means

Iako kod umjetno stvorenih podataka točno znamo koji korisnik pripada kojoj grupi (odnosno koji je korisnik stvoren iz određenog profila) to neće biti moguće unaprijed znati sa realnim podacima i stvarnim korisnicima. „U takvim slučajevima moramo se osloniti na **nenadzirano učenje** (engl. *unsupervised learning*). Tri tipična zadatka nenadziranog učenja su **grupiranje** (engl. *clustering*) podataka, otkrivanje novih vrijednosti ili vrijednosti koje odskaču (engl. *novelty/outlier detection*) i **smanjenje dimenzionalnosti** (engl. *dimensionality reduction*)“ (Šnajder et al., 2014).

S obzirom da je **algoritam k-srednjih vrijednosti** (engl. *k-means algorithm*) algoritam grupiranja, to ga čini idealnim za implementaciju sustava preporuka budući da je potrebno grupirati slične korisnike.

Princip na kojem algoritam funkcionira je da se podaci koji se grupiraju moraju prikazati kao točke u nekom n-dimenzionalnom prostoru. Nakon toga se na neki način odabire k točaka u tom prostoru gdje k predstavlja broj očekivanih grupa. Te točke predstavljaju centar pojedine grupe te se nazivaju centroidi (engl. *centroids*). Nakon odabira inicijalnih centroida ostale točke (odnosno podaci) se grupiraju sa najbližim centroidom prema nekom kriteriju udaljenosti te se za svaku novonastalu grupu (engl. *cluster*) računa prosječna vrijednost svih točaka čime nastaje novi centroid te grupe oko kojeg se ponavlja cijeli postupak grupiranja

podataka. Ovi koraci se ponavljaju sve dok se centroidi ne ustabile odnosno dok centroidi iz prijašnje iteracije ne budu jednaki novo nastalim centroidima.

7.2. Implementacija

Algoritam k-srednjih vrijednosti zahtjeva da su ulazni podaci vektori koji predstavljaju točku u n-dimenzionalnom prostoru tako da je u principu potrebno korisnika svesti na točku. Način na koji je to učinjeno je da se za svakog korisnika odredi ukupan zbroj količina za svaki sastojak i za sve recepte koje je zabilježio. Time dobivamo sumarni popis sastojaka jednog korisnika gdje je npr. ukupna količina soli za sve recepte od jednog korisnika 20 kila, ili ukupna količina papra 5 kila. Taj popis možemo iskoristiti kao vektor koji predstavlja korisnika te ga je prije grupiranja potrebno normalizirati što će biti obrađeno malo kasnije.

Budući da algoritam k-srednjih vrijednosti radi samo sa vektorima, pitanje je kako nazad povezati korisnike sa njihovim vektorima nakon grupiranja budući da transformacija iz korisnika u vektor nije bijektivna.

```
type Vector = [ Double ]

data Projection a = Projection
  { source :: !a
  , vector :: !Vector
  } deriving ( Eq, Show )
```

Kôd 7.1 Podatkovni tip za projekciju podataka u vektorski prostor

Kako bi se riješio taj problem, definiran je podatkovni tip `Projection` koji predstavlja projekciju nekog podatkovnog tipa u vektorski prostor.

```
data Cluster a = Cluster
  { centroid :: !Vector
  , cmembers :: ![ Projection a ]
  } deriving ( Show )

instance Eq ( Cluster a ) where
  c1 == c2 = centroid c1 == centroid c2
```

Kôd 7.2 Podatkovni tip grupe podataka

Uz podatkovni tip `Projection` definiran je i podatkovni tip grupe `Cluster` koji u sebi sadrži listu projekcija podataka i centroid.

```
kmeans :: [ Cluster a ] -> [ Cluster a ]
kmeans cs | cs == cs' = cs | otherwise = kmeans cs'
  where cs' = recluster cs

recluster :: [ Cluster a ] -> [ Cluster a ]
recluster cs = cluster cs' ps
  where ps  = concat $ fmap cmembers cs
        cs' = fmap ( mean . fmap vector . cmembers ) cs

mean :: [ Vector ] -> Vector
mean ps = fmap (/genericLength ps) $ foldl1' (zipWith (+)) ps
```

Kôd 7.3 Implementacija algoritma k-srednjih vrijednosti

Nakon definiranja pomoćnih tipova `Cluster` i `Projection` moguće je dobiti poprilično elegantnu definiciju algoritma k-srednjih vrijednosti koja je prikazana u primjeru Kôd 7.3. Funkcija `kmeans` prima neku inicijalnu listu grupa `cs` te na nju primjenjuje funkciju `recluster`. Ukoliko je lista grupa `cs` jednaka onoj koju vrati funkcija `recluster` tada je to znak da je konfiguracija stabilna te da proces grupiranja može završiti i vratiti `cs` kao rezultat. Ukoliko to nije slučaj, funkcija `kmeans` se rekurzivno poziva sa rezultatom funkcije `recluster` kao ulaznim parametrom.

S druge strane funkcija `recluster` računa nove centroide svih grupa uz pomoć funkcije `mean` te zatim izvlači projekcije iz svih grupa u odvojenu listu. Nakon toga funkcija `recluster` poziva funkciju `cluster` koja prima listu centroida i listu projekcija te pokušava grupirati projekcije oko centroida u listi.

```
cluster :: [ Vector ] -> [ Projection a ] -> [ Cluster a ]
cluster cs = Map.elems . foldl' go mempty
  where add p c = c { cmembers = p : cmembers c }
        go m p =
          let k = minimumBy
                  ( comparing $ distance $ vector p ) cs
          in  Map.alter
              ( Just . maybe ( Cluster k [ p ] ) ( add p ) )
              k m
```

Kôd 7.4 Implementacija funkcije grupiranja

Funkcija `cluster` koristi rječnik kao optimalnu strukturu za ovaj proces koji kao ključeve ima centroide a kao vrijednosti čuva grupe, te prolazi kroz listu projekcija i svakoj traži najbliži centroid. Nakon toga, po ključu (odnosno centroidu) dodaje projekciju u rječnik iz kojeg na kraju izvlači grupe i vraća ih kao rezultat. Za određivanje udaljenosti projekcije od centroida korištena je Euklidska udaljenost te je njena implementacija dana u Kôd 7.5.

```
distance :: Vector -> Vector -> Double
distance v1 v2
  = sqrt $ sum $ fmap (^(2::Int)) $ zipWith (-) v1 v2
```

Kôd 7.5 Funkcija udaljenosti

7.3. Grupiranje korisnika

Kao što je to ranije bilo spomenuto, kako bi mogli grupirati korisnike uz pomoć algoritma k-srednjih vrijednosti potrebno ih je prikazati u vektorskom obliku koji predstavlja točku u nekom n-dimenzionalnom prostoru.

Vektorska reprezentacija korisnika stvara se na način da se prikupe svi recepti koje je korisnik spremio u svoju kuharicu. Iz njih se izvlače sastojci i njihove količine te se za svaki sastojak bilježi ukupna pojavljena količina u svim receptima. Time se dobiva lista sa popisom ukupnih količina sastojaka za pojedinog korisnika koja se nadopunjava sa ostalim postojećim sastojcima iz baze koji se nisu pojavili niti u jednom spremljenom receptu te se njihova ukupna količina postavlja na nulu.

Budući da će se sol kao sastojak recepta obično pojavljivati u gramima a meso u kilogramima to dovodi do velike razlike između te dvije vrijednosti. Algoritam k-srednjih vrijednosti je osjetljiv na takve razlike u vrijednostima te je za kvalitetne rezultate potrebno normalizirati te vrijednosti prije grupiranja (Mohamad B. et al., 2013).

Normalizacija se radi na način da se u skupu korisničkih vektora, za svaki sastojak pronađe minimalna i maksimalna vrijednost te se zatim sve vrijednosti koje predstavljaju pojedini sastojak skaliraju na raspon između 0 i 1.

```
scale :: (Eq a, Fractional a) => (a,a) -> (a,a) -> a -> a
scale ( mn, mx ) ( mn', mx' ) n
  | mx - mn == 0 = 0
  | otherwise = ( (mx' - mn') * (n - mn) / (mx - mn) ) + mn'
```

Kôd 7.6 Funkcija za skaliranje vrijednosti

7.3.1. Jednostavna inicijalizacija

„Algoritam k-srednjih vrijednosti je pohlepan i pronalazi lokalno optimalno rješenje. Hoće li to rješenje biti i globalno optimalno, ovisi o izboru početnih srednjih vrijednosti.“ (Šnajder J. et al. 2014)

Iz potpisa funkcije `kmeans` definirane u Kôd 7.3 vidi se da kao ulaznu vrijednost očekuje već definiranu listu klastera sa srednjim vrijednostima. Funkcija `kmeans` je neovisna o algoritmu inicijalizacije te se on odvija zasebno nakon čega se rezultat daje kao ulazni argument `kmeans`-u.

```
initSP :: K -> [ Projection a ] -> [ Cluster a ]
initSP k ps = map cluster' $ chunksOf n ps
  where n = fromIntegral $ (genericLength ps + k - 1) `div` k
        cluster' ps' = Cluster ( mean $ fmap vector ps' ) ps'
```

Kôd 7.7 Jednostavni algoritam *k-means* inicijalizacije

U Kôd 7.7 je dana definicija jednostavnog algoritma inicijalizacije koji se svodi na to da se lista ulaznih podataka raspodijeli na k dijelova koji se automatski smatraju grupama te se te grupe predaju funkciji `kmeans` kao ulazni parametar.

Ovaj pristup daje zadovoljavajuće rezultate, no može se primijetiti da malo spor pri grupiranju.

7.3.2. K-Means++ inicijalizacija

K-Means++ je algoritam za odabir početnih centroida koji potencijalno može ubrzati proces traženja stabilne konfiguracije te poboljšati preciznost algoritma k-srednjih vrijednosti.

Algoritam je odabran kao alternativa jednostavnom i intuitivnom pristupu inicijalizaciji. Za razliku od jednostavne inicijalizacije K-Means++ predlaže slijedeće, u slučaju gdje je $D(x)$ najkraća udaljenost od podatkovne točke (engl. *data point*) do najbližeg već odabranog centroida možemo definirati slijedeći algoritam koji predstavlja K-Means++:

1. Odaberi jedan centroid c_1, odabran nasumično iz X
2. Odaberi novi centroid c_i, birajući $x \in X$ sa vjerojatnošću $\frac{D(x)^2}{\sum_{x \in X} D(x)^2}$
3. Ponavljaj korak 2 dok nije odabrano k centroida
4. Nastavi normalno sa klasičnim k-means algoritmom

(Arthur D., et al. 2007).

```
initPP :: K -> [ Projection a ] -> Rand StdGen [ Cluster a ]
initPP k ps = flip cluster ps <$> ctroids
  where vectors :: [ Vector ]
        vectors = fmap vector ps

        initial :: Rand StdGen [ Vector ]
        initial =   getRandomR ( 0, fromIntegral $ k - 1 )
                >>= pure . (:[]) . (vectors!!)

        bigDist :: [ Vector ] -> Vector -> Double
        bigDist cs p
          = distance p
          $ minimumBy ( comparing $ distance p ) cs

        weights :: [ Vector ] -> [ ( Vector, Double ) ]
        weights cs = fmap
          ( \p -> ( p, bigDist cs p ^ ( 2 :: Int ) ) )
          vectors

        nextctr :: [ Vector ] -> Rand StdGen [ Vector ]
        nextctr cs = (:cs) <$> wpicker totalwgh weighted
          where weighted = weights cs
                totalwgh = sum $ fmap snd weighted

        wpicker
         :: Double -> [(Vector,Double)] -> Rand StdGen Vector
        wpicker t ws = getRandomR ( 0, t ) >>= pure . walk ws
          where walk [] _ = error "can't work on empty list"
                walk ((x, w):xws) random
                  | random - w < 0 = x
                  | otherwise      = walk xws $ random - w

        ctroids :: Rand StdGen [ Vector ]
        ctroids = iterate ( >>= nextctr ) initial
                  !! ( fromIntegral $ k - 1 )
```

Kôd 7.8 K-Means++ algoritam inicijalizacije

Za razliku od postupka inicijalizacije `initSP` prikazanog u Kôd 7.7 K-Means++ algoritam prikazan u Kôd 7.8 mora se izvršavati u monadi koja podržava stvaranje nasumičnih brojeva.

7.4. Provjera kvalitete grupiranja korisnika

Provjeravanje ispravnosti i kvalitete grupiranja algoritma k-srednjih vrijednosti obično nije lak zadatak s obzirom da se on obično koristi na podacima o kojima ne znamo mnogo, pa tako niti točan broj kategorija i njihovu konfiguraciju. Još jedna nepovoljna okolnost provjere ispravnosti je ta što ovisno o postupku inicijalizacije algoritam može imati i nasumičnu komponentu.

Na sreću, u ovom slučaju postoji poprilična kontrola nad podacima koje grupiramo te je unaprijed poznat broj grupa korisnika.

Kako bi se testira kvaliteta algoritma stvorena je funkcija sa slijedećim tipskim potpisom:

```
userClusteringTest
  :: StdGen -- ^ generator nasumičnih brojeva
  -> Int    -- ^ broj sastojaka
  -> Int    -- ^ broj profila
  -> Int    -- ^ broj recepata po profilu
  -> Int    -- ^ broj korisnika po profilu
  -> IO ()
```

Implementacija funkcije `userClusteringTest` je poveća tako da nije prikazana u ovom radu. Funkcija prima generator nasumičnih brojeva kako bi se mjerenja mogla ponoviti više puta ali dati isti rezultat. Ono što funkcija radi može se opisati na slijedeći način:

1. Stvara određen broj sastojaka (`ns`)
2. Stvara određen broj profila (`np`)
3. Za svaki profil stvara određen broj recepata iz stvorenih sastojaka (`nr`)
4. Za svaki profil stvara određen broj korisnika iz stvorenih recepata (`nk`)
5. Grupira korisnike u `np` kategorija
6. Ispisuje korisnike po grupama
7. Računa te ispisuje kvalitetu grupiranja
8. Generira grafove svih grupa radi lakše vizualne inspekcije

Nakon izvršavanja `userClusteringTest (mkStdGen 10) 60 6 10 5` dobivena je slijedeća tablica rezultata.

Tablica 7.1 Tablica rezultata grupiranja korisnika

Grupa 01	Grupa 02	Grupa 03	Grupa 04	Grupa 05	Grupa 06
Pr 4 Kor 1	Pr 1 Kor 2	Pr 5 Kor 1	Pr 1 Kor 3	Pr 3 Kor 1	Pr 1 Kor 1
Pr 4 Kor 2	Pr 1 Kor 4	Pr 5 Kor 2	Pr 1 Kor 5	Pr 3 Kor 2	Pr 4 Kor 3
Pr 6 Kor 2	Pr 2 Kor 4	Pr 5 Kor 3	Pr 2 Kor 1	Pr 3 Kor 3	Pr 4 Kor 4
Pr 6 Kor 3	Pr 2 Kor 5	Pr 5 Kor 5	Pr 2 Kor 2	Pr 3 Kor 4	Pr 4 Kor 5
Pr 6 Kor 4	Pr 6 Kor 1		Pr 2 Kor 3	Pr 3 Kor 5	
			Pr 5 Kor 4	Pr 6 Kor 5	

Ime korisnika je stvoreno na način da indicira njegov profil (tj. očekivanu grupu) te njegov indeks odnosno koji je po redu bio generiran iz određenog profila.

Ocjena odnosno kvaliteta ovog grupiranja iznosi 0.666 što je poprilično dobro, ali se i može primijetiti iz tablice ukoliko se koncentriramo na ***Pr*** dio naziva.

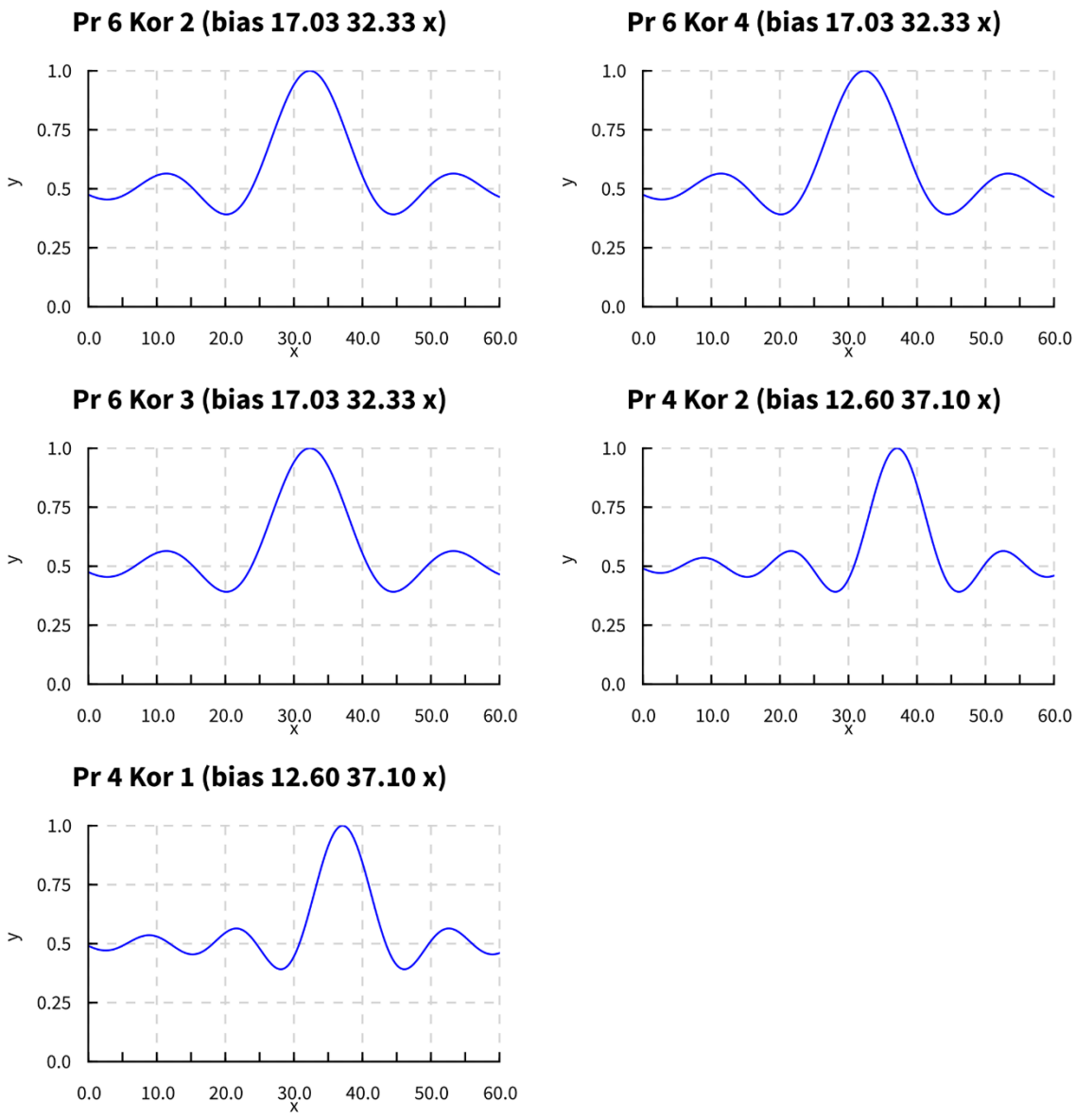
Budući da su grupe i pripadnost korisnika tim grupama unaprijed poznate ocjena se računa na slijedeći način:

1. Za svaki profil pronađe se najveći broj pripadajućih korisnika grupiranih zajedno u nekoj od grupa. Za slučaj profila 5 najveći broj korisnika koji su grupirani zajedno bi bio 4 ako pogledamo u stupac Grupa 03 u Tablica 7.1.
2. Zatim se tih vrijednosti izračuna prosjek
3. Te se taj prosjek podijeli sa brojem korisnika generiranih po profilu

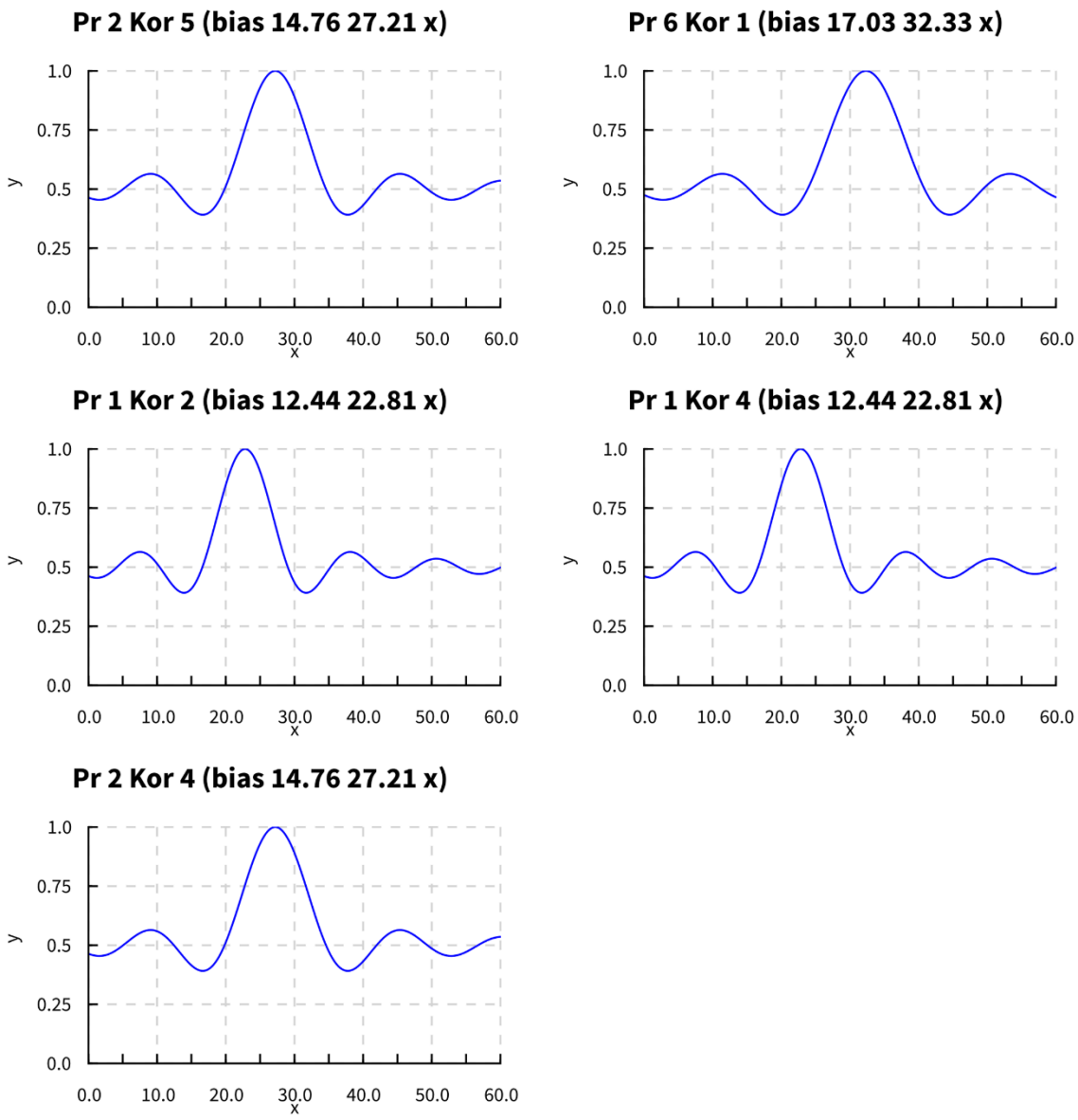
Treba primijetiti da je ovo u principu „najniža“ moguća ocjena grupiranja. Stvar je u tome da iako smo mi odredili broj grupa u generiranim podacima sve jedno postoji element nasumičnosti koji u teoriji može stvoriti dva identična profila korisnika (dok će u praksi to vjerojatnije biti dva vrlo slična profila korisnika) zbog čega bi u principu trebali umanjiti očekivani *k* za 1 (makar to u praksi ne možemo jednostavno saznati).

Taj slučaj je vrlo lako moguć, te ako pogledamo u Tablica 7.1 možemo primijetiti da su profili 6 i 4 grupirani zajedno nemaloj količini. Isto možemo primijetiti i za profile 1 i 2.

Ukoliko pogledamo grafikone grupa 01 i 02 primijetit ćemo da su profili 6 i 4 te 1 i 2 uistinu poprilično slični.



Slika 7.1 Graf grupe 01



Slika 7.2 Graf grupe 02

7.5. Usporedba jednostavne i K-Means++ inicijalizacije

Sa prethodno definiranim mjerilima moguće je usporediti performanse jednostavne inicijalizacije te K-Means++ inicijalizacije.

```
measurements :: [(Int, Int, Int, Int)]
measurements = do
  numOfIngredients       <- [50, 100, 200]
  numOfProfiles          <- [10, 15]
  numOfRecipesPerProfile <- [10, 30]
  numOfUsersPerProfile   <- [10, 40]
  pure
    ( numOfIngredients
    , numOfProfiles
    , numOfRecipesPerProfile
    , numOfUsersPerProfile
    )
runMeasurement
  :: StdGen
  -> (Int, Int, Int, Int)
  -> IO ( Double, Double, Double, Double, Double, Double )
runMeasurements :: FilePath -> IO ()
```

Kôd 7.9 Pomoćne funkcije za usporedbu algoritama inicijalizacije grupa

Za tu svrhu dani su potpisi funkcija te definicije u Kôd 7.9. Vrijednost `measurements` definira permutacije ulaznih parametara za funkciju `runMeasurement`. Lista je monada te se tu može koristiti monadsko sučelje i `do` sintaksa kako bi se na jednostavan način stvorile permutacije raznih vrijednosti. Ovdje se konkretno spominju četiri liste sa po 3, 2, 2 te 2 elemenata. To znači da postoji ukupno 3*2*2*2 = 24 setova argumenata koji će se primijeniti na funkciju `runMeasurement`. Funkcija to sve objedinjuje te rezultate zapisuje u CSV datoteku na disku te sadrži slijedeća zaglavlja:

Tablica 7.2 Zaglavlja datoteke rezultata mjerenja

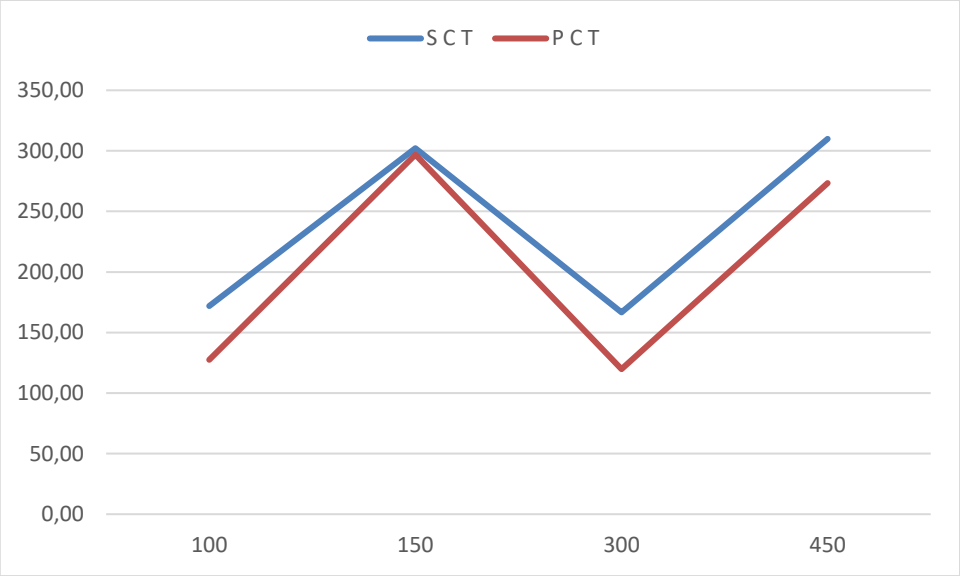
N Ing	Broj generiranih sastojaka
N Pr	Broj generiranih profila
N R / P	Broj generiranih recepata po profilu

N U / P	Broj generiranih korisnika po profilu
S S	Ocjena algoritma grupiranja za jednostavnu inicijalizaciju
S T	Vrijeme potrebno za jednostavnu inicijalizaciju (ms)
S C T	Vrijeme potrebno za grupiranje nakon jednostavne inicijalizacije (ms)
P S	Ocjena algoritma grupiranja za K-Mean++ inicijalizaciju
P T	Vrijeme potrebno za K-Mean++ inicijalizaciju (ms)
P C T	Vrijeme potrebno za grupiranje nakon K-Mean++ inicijalizacije (ms)

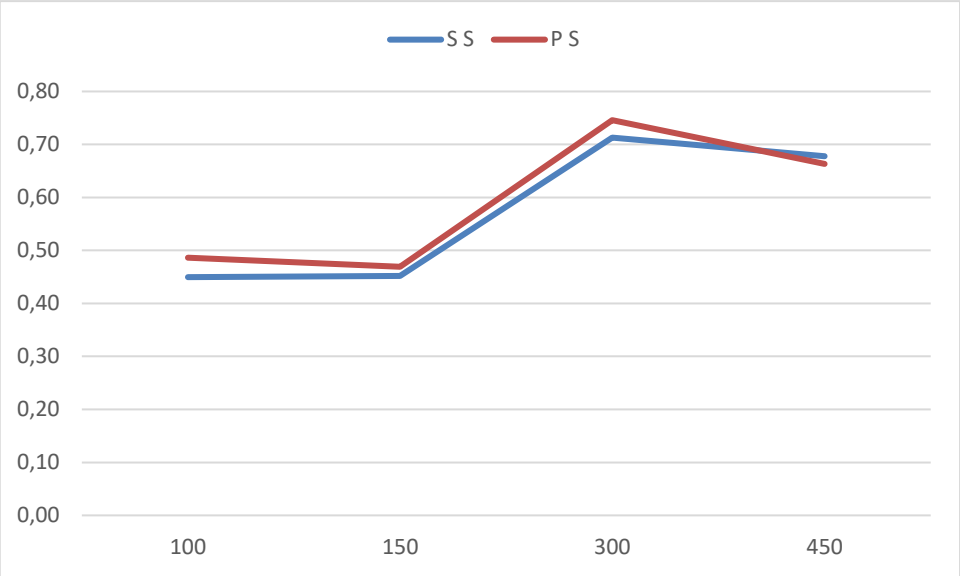
Tablica 7.3 Rezultati mjerenja K-Means++ i jednostavnog algoritma inicijalizacije

N Ing	N Pr	N R / P	N U / P	S S	S T	S C T	P S	P T	P C T
50	10	10	10	0.45	0.00	15.63	0.47	0.00	15.63
50	10	10	40	0.48	0.00	78.13	0.41	31.25	78.13
50	10	30	10	0.76	0.00	15.63	0.80	15.63	0.00
50	10	30	40	0.72	15.63	93.75	0.79	31.25	62.50
50	15	10	10	0.45	15.63	31.25	0.47	15.63	31.25
50	15	10	40	0.43	15.63	281.25	0.38	125.00	218.75
50	15	30	10	0.70	0.00	31.25	0.67	15.63	31.25
50	15	30	40	0.81	31.25	156.25	0.71	109.38	343.75
100	10	10	10	0.46	0.00	31.25	0.48	15.63	15.63
100	10	10	40	0.48	15.63	375.00	0.50	62.50	171.88
100	10	30	10	0.71	15.63	15.63	0.74	0.00	15.63
100	10	30	40	0.71	15.63	281.25	0.67	78.13	265.63
100	15	10	10	0.52	0.00	62.50	0.55	46.88	31.25
100	15	10	40	0.45	15.63	421.88	0.46	234.38	390.63
100	15	30	10	0.63	15.63	46.88	0.72	62.50	46.88
100	15	30	40	0.66	31.25	484.38	0.66	265.63	359.38
200	10	10	10	0.37	0.00	46.88	0.52	46.88	46.88
200	10	10	40	0.46	15.63	484.38	0.53	156.25	437.50
200	10	30	10	0.67	0.00	46.88	0.75	46.88	46.88
200	10	30	40	0.72	15.63	546.88	0.72	171.88	328.13
200	15	10	10	0.37	0.00	109.38	0.48	125.00	62.50
200	15	10	40	0.48	15.63	906.25	0.47	500.00	1046.88
200	15	30	10	0.57	0.00	156.25	0.61	125.00	62.50
200	15	30	40	0.69	46.88	984.38	0.62	500.00	796.88
Srednje vrijednosti				0.57	11.72	237.63	0.59	115.89	204.43

Iz Tablica 7.3 može se primijetiti da je kod K-Means++ algoritma vrijeme inicijalizacije dosta lošije u usporedbi sa jednostavnim algoritmom, no u prosjeku daje nešto bolje rezultate te brže dostiže stabilnu konfiguraciju. Iz toga se da zaključiti da bi u slučaju ovih podataka K-Means++ bio pogodniji kod manjih broja klastera te velikog seta podataka.

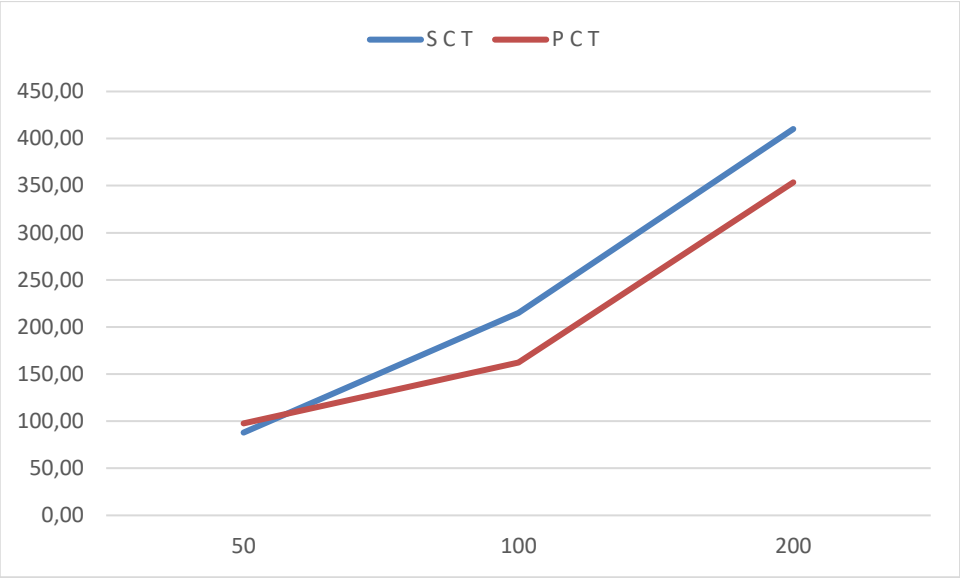


Slika 7.3 Usporedba prosječnih SCT i PCT vrijednosti prema broju podataka

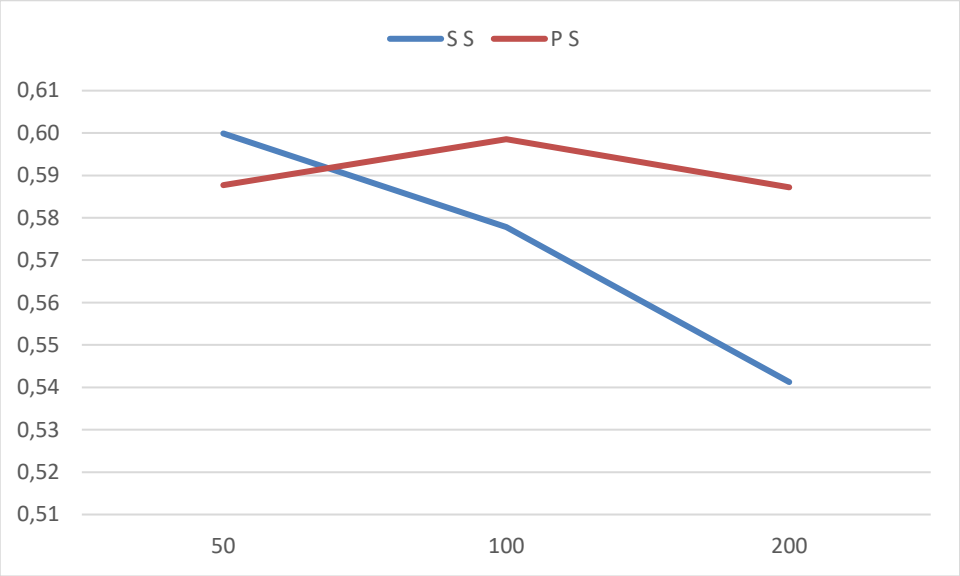


Slika 7.4 Usporedba prosječnih SS i PS vrijednosti prema broju podataka

Ukoliko pogledamo prosječne vrijednosti vremena grupiranja te kvalitete grupiranja prema količini obrađivanih podataka možemo primijetiti da je K-Means++ nešto bolji.



Slika 7.5 Usporedba prosječnih SCT i PCT vrijednosti prema broju atributa



Slika 7.6 Usporedba prosječnih SS i PS vrijednosti prema broju atributa

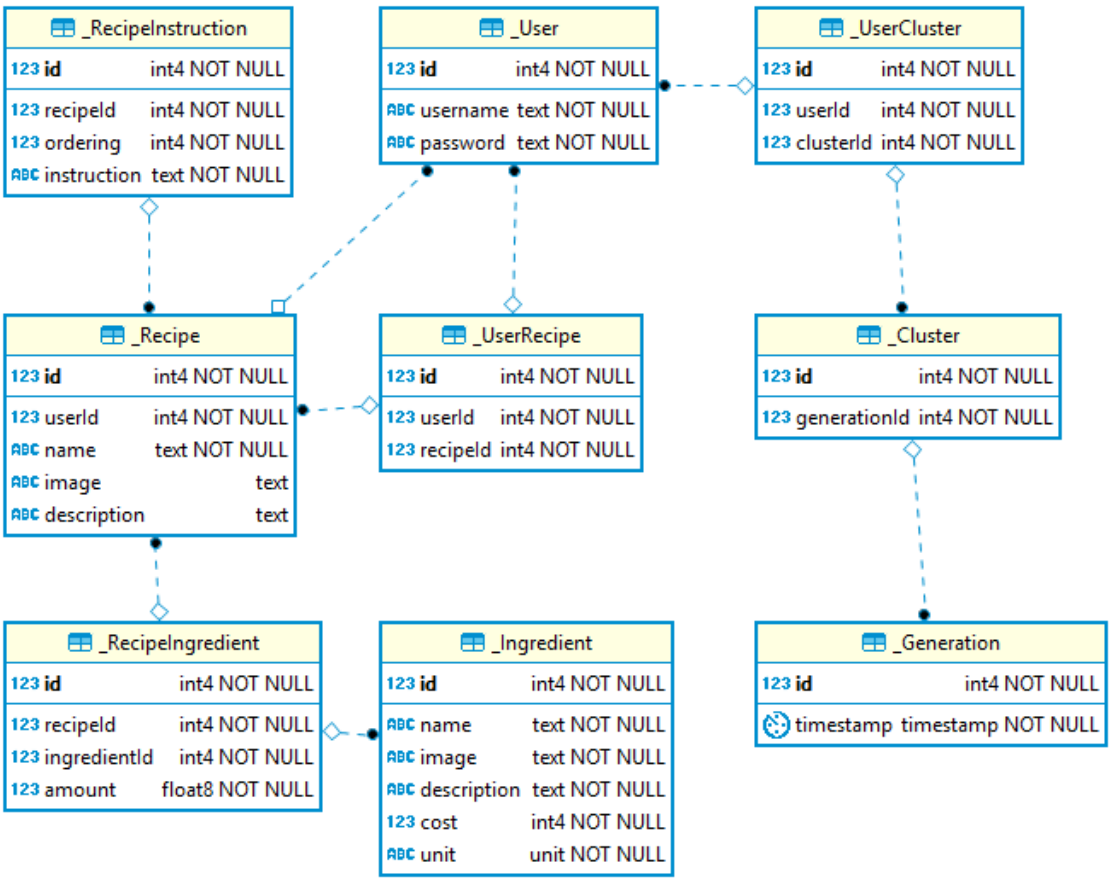
Ako usporedimo prosječne vrijednosti vremena grupiranja i kvalitete grupiranja prema broju atributa, i tu je K-Means++ opet neznatno bolji.

Dolazi se do zaključka da K-Means++ nije bio vrijedan uloženog vremena u ovome slučaju.

8. Korisnička aplikacija

Korisnička aplikacija zamišljena je kao *jedno-stranična aplikacija* (engl. *Single Page Application*, skraćeno SPA) koja u pozadini komunicira sa JSON API serverom te preko njega dobiva podatke o korisnicima, sastojcima i receptima iz *PostgreSQL* baze podataka.

8.1. Baza podataka



Slika 8.1 ER dijagram produkcijske baze podataka

U produkcijskoj bazi tablica recepata povezana je sa tablicom sastojaka preko vezne tablice u kojoj se bilježi količina određenog sastojka u receptu, uz to tablica je još povezana i sa tablicom uputa za pripremu tog recepta te svaki recept može imati svog autora i druge korisnike koji su ga spremili kroz tablicu korisničkih recepata. Radi sustava preporuka napravljene su tablice generacija, klastera i korisnika koji pripadaju pojedinom klasteru te se pri svakom pokretanju algoritma za preporuke stvara nova generacija klastera korisnika.

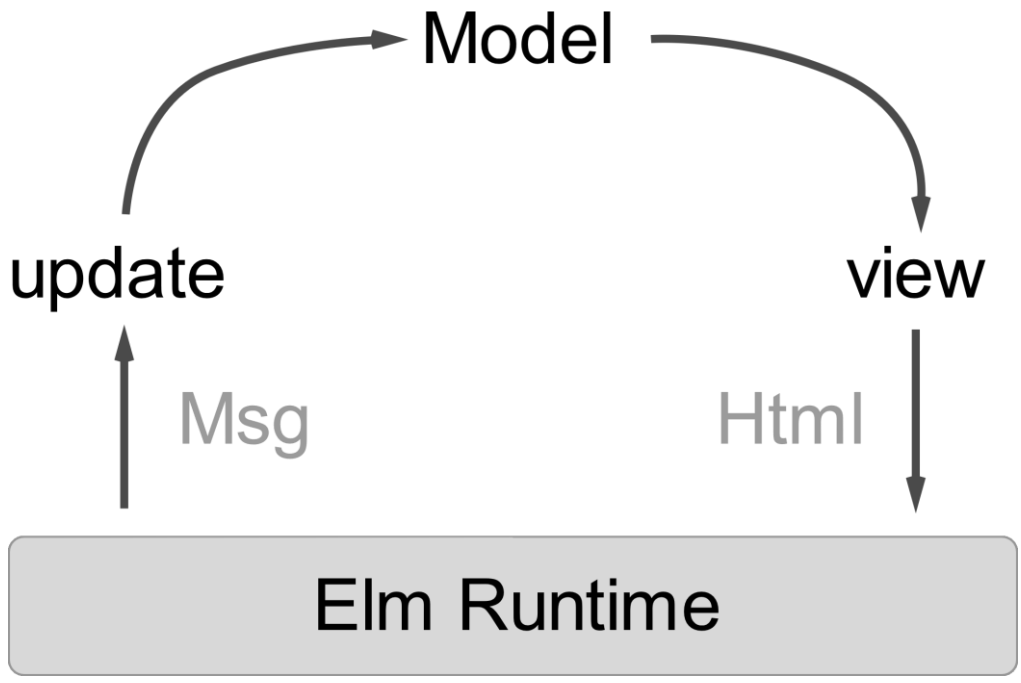
8.2. API server

API server definiran je uz pomoć *Servanta* na sličan način kao što je to ranije napravljeno za Coolinariku u Kôd 4.13, osim što je ovoga puta bilo potrebno i implementirati taj API. Implementacija se u principu svodi na obične funkcije za dohvaćanje sadržaja iz baze podataka kao što je napravljeno u Kôd 4.6. Ukoliko funkcija vraća na primjer listu sastojaka iz baze, potrebno je samo derivirati instance `FromJSON` i `ToJSON` te će *Servant* automatski pretvoriti vraćenu listu sastojaka u JSON i proslijediti ga kao odgovor na upit.

8.3. Korisničko sučelje

Korisničko sučelje je napravljeno sa *Miso* programskom zbirkom. Ta programska zbirka napisana je u Haskellu te je namijenjena da se kompilira sa GHCJS kompajlerom. GHCJS je kompajler koji Haskell prevodi u JavaScript što je izrazito korisno jer omogućava da se isti kod koristi i u Internet pregledniku i na serveru što znači da se nije potrebno zamarati rukovanjem JSON upitima i odgovorima već je moguće koristiti *Servant Client* za jednostavnu komunikaciju sa serverom.

Miso je baziran na takozvanoj Elm arhitekturi koja dolazi iz Elm-a, funkcijskog programskog jezika baziranog na Haskellu te specijalno dizajniranog za prevođenje u JavaScript i izradu Internet aplikacija.



Slika 8.2 Elm arhitektura

Miso koristi *funkcijsko reaktivno programiranje* (engl. *Functional Reactive Programming*, skraćeno FRP) kao način za opisivanje ponašanja korisničkog sučelja. FRP služi za modeliranje promjena kroz vrijeme te se može smatrati alternativnim načinom opisivanja ponašanja.

```
data Model = Model
  { currentURI :: URI
  , account :: Maybe UserToken
  , ingredient :: Either Text Ingredient
  , ingredients :: Either Text ( Paginated [Ingredient] )
  , …
  }
```

Kôd 8.1 Primjer modela korisničke aplikacije

Kako bi Elm arhitektura bila zadovoljena potrebno je prvo definirati model aplikacije čiji je dio prikazan u Kôd 8.1. On sadrži u sebi trenutnu putanju tj. lokaciju unutar aplikacije, ukoliko je korisnik ulogiran onda sadrži i njegov korisnički token koji se koristi za pristup zaštićenim dijelovima API-ja te razne druge detalje poput liste sastojaka, detalja pojedinog sastojka ili čak poruke o greškama.

```
data Action
  = GoHome
  | GetIngList Pagination
  | SetIngList ( Either Text [ Ingredient ] )
  | AddToFavourites ( ID Recipe )
  …
  | NoOp
```

Kôd 8.2 Primjer definicije akcija koje je moguće izvršiti unutar aplikacije

Nakon modela potrebno je definirati akcije koje je moguće izvršiti unutar aplikacije čiji je primjer dan u Kôd 8.2. Akcije se obično definiraju kao sumarni tip te su izrazito korisne jer na jednom mjestu pružaju dokumentaciju o svim mogućim akcijama koje aplikacija može napraviti. One su u principu samo *nosači* informacije te se konkretne radnje izvršavaju unutar `update` funkcije definirane u **Pogreška! Izvor reference nije pronađen.**.

```
update :: Action -> Model -> Effect Action Model
update (GetIngList p) m = m <# do SetIngList <$> getIngList p
update (SetIngList eis) m = noEff m { ingredients = eis }
update (ChangeURI uri) m = m <# do pushURI uri >> pure NoOp
```

Kôd 8.3 Dio definicije update funkcije

Funkcija `update` prima akciju i trenutni model te vraća efekt koji će, nakon što bude izvršen od strane Miso runtime-a, promijeniti model aplikacije. Korištenjem podudaranja uzoraka (engl. *pattern matching*) moguće je vrlo lako prepoznati o kojoj se akciji radi te izvršiti potrebne radnje za promjenu modela kao što je prikazano u Kôd 8.3.

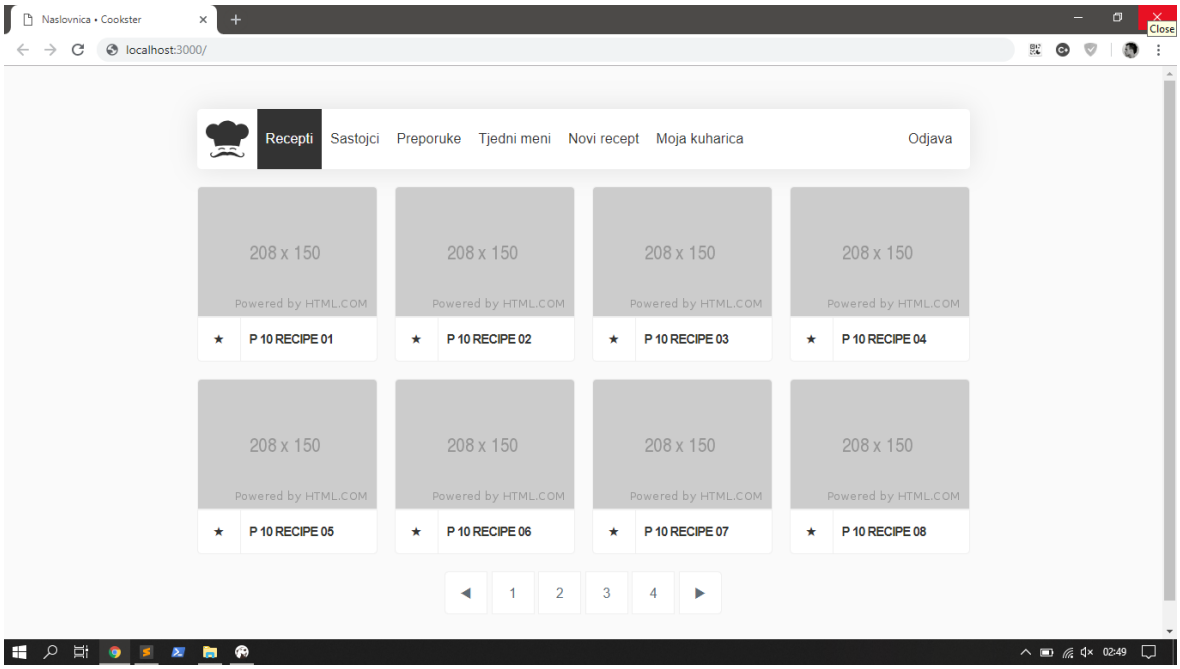
```
loginForm :: Maybe Credentials -> View action
loginForm mc = form_ []
  [ label_ [ for_ "username" ] [ text "Korisničko ime" ]
  , input_
    [ type_ "text", id_ "username"
    , value_ $ text $ maybe "" username mc ]
  , label_ [ for_ "password" ] [ text "Lozinka"]
  , input_
    [ type_ "text", id_ "password"
    , value_ $ text $ maybe "" password mc ]
  , input_
    [ type_ "submit", value_ "Prijava", onClick LogIn ]
  ]
```

Kôd 8.4 Primjer pogleda forme za prijavu

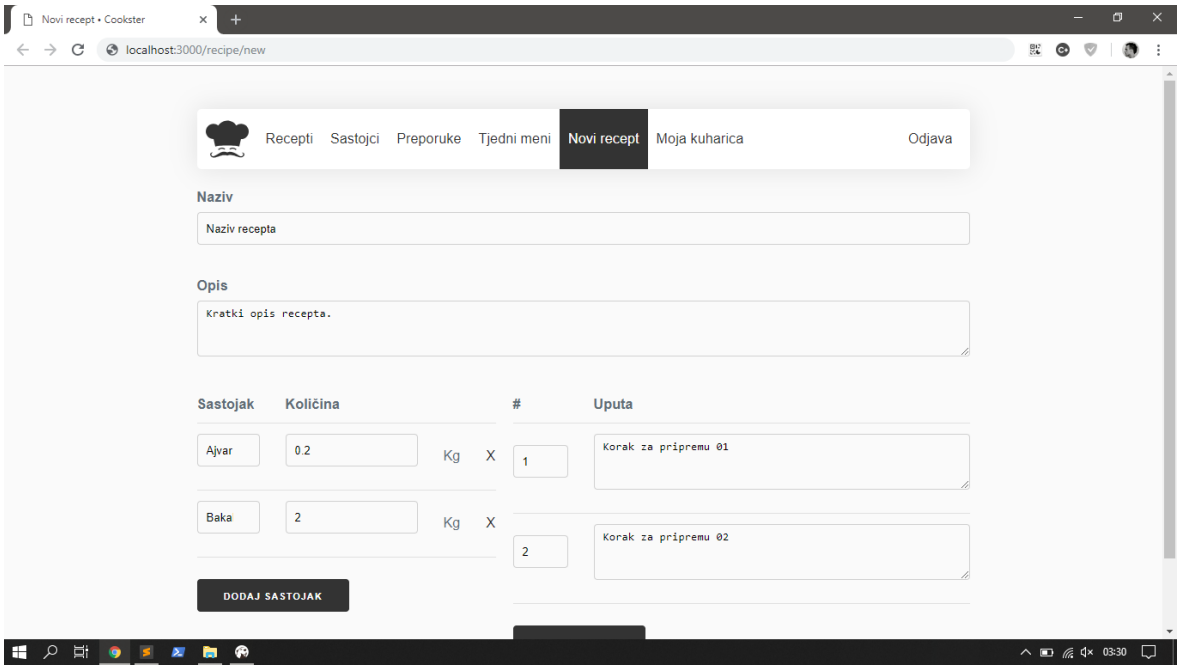
Zadnji dio kojeg je potrebno definirati je pogled (engl. *view*) koji je u principu rezultat trenutnog modela. Nakon što *runtime* izvrši efekt i promijeni model, taj model (ili njegov dio) se šalje u odgovarajući pogled što je u slučaju Kôd 8.4 pogled forme za prijavu korisnika.

U pogledu se koristi EDSL za HTML kako bi se definiralo korisničko sučelje te se putem funkcije `onClick` (i raznih drugih) određuje koja će akcija biti izvršena na klik gumba za prijavu. U ovom slučaju je to akcija `LogIn`.

Kako bi se olakšalo određivanje ispravnog pogleda opet je korišten *Servant* kojim je moguće na jednostavan način definirati rute unutar *Miso* aplikacije. Klikom na neki od izbornika prikazanih na Slika 8.3 i Slika 8.4 mijenja se adresa u adresnoj traci. Tu promjenu hvata *Servant* te raščlanjuje novu adresu i postavlja odgovarajući pogled.



Slika 8.3 Korisničko sučelje za pregled recepata



Slika 8.4 Korisničko sučelje za stvaranje novog recepta

9. Testiranje

Za provjeru ispravnosti napisanog koda korištene su programske zbirke *Hspec*, *HUnit* i *QuickCheck*. *Hspec* je EDSL za pisanje testova koji objedinjuje *HUnit* koji služi za pisanje *jediničnih testova* (engl. *unit test*) i *QuickCheck* koji služi za svojstveno testiranje (engl. *property based testing*).

```
spec :: Spec
spec = describe "Coolinarika" $ do
  describe "Scrapers" $ do
    describe "slugsS" $ do
      it "scrapes list of ingredient slugs from HTML" $ do
        shouldReturn
          ( ph slugsS "…/IngredientList.html" )
          $ Just [ Slug "aceto-balsamico" ]
    describe "ingredientS" $ do
    ...
```

Kôd 9.1 Primjer jediničnih testova za strugače

Primjer jediničnog testa dan u Kôd 9.1 učitava lokalno spremljenu HTML datoteku te na nju primjenjuje `slugsS` strugač definiran u Kôd 4.8 i provjerava da li je dobiven očekivan rezultat.

Osim jediničnih testova moguće je provesti i svojstveno testiranje nad funkcijama.

```
spec :: Spec
spec = do
  describe "biased vs non-biased" $ do
    it "biased score should be higher than non-biased" $
      forAll ( choose ( 50, 250 ) ) $ \ni ->
      forAll ( choose ( 5 , 20  ) ) $ \np ->
      forAll ( choose ( 5 , 20  ) ) $ \nr ->
      property                      $ \s  ->
        let
          g = mkStdGen s
          bis = evalRand (averageBiasedPref ni np nr) g
          non = evalRand (averageNonBiasedPref ni np nr) g
        in  bis >= non
```

Kôd 9.2 Primjer svojstvenog testiranja

U Kôd 9.2 dan je primjer svojstvenog testiranja korištenjem *QuickCheck*-a. Tu se testira da li umjetno stvoreni korisnici imaju veću sklonost receptima koji su stvoreni prema njihovom „ukusu“. Svojstvo je u ovom slučaju to da sklonost specijalno pripremljenim receptima mora uvijek biti veća od sklonosti prema ostalim receptima. *QuickCheck* sam puni argumente funkcija te pokušava pronaći rubni slučaj koji poništava zadano svojstvo koje mora biti zadovoljeno a ako ne uspije pronaći takav slučaj onda je test zadovoljen.

Zaključak

Cilj ovog rada bio je istražiti funkcijsku paradigmu kroz izradu pametne kuharice. Funkcijski programski jezik Haskell pokazao se kao izrazito elegantan i moćan alat za razvoj softvera. Jak tipski sustav, lijenost, čistoća, funkcije višeg reda i sl. su pridonijeli ugodnom iskustvu prilikom razvoja pametne kuharice.

Prikupljanje podataka prošlo je bez većih problema uglavnom zbog vrlo moćnih zbirki za raščlanjivanje koje su uglavnom jaka strana funkcijskih programskih jezika.

Podrška za istraživačku analizu i obradu podataka u Haskellu nije baš jaka kao što je to recimo slučaj sa Pythonom. Postoje integracije sa Jupyterom no nisu na zadovoljavajućoj razini tako da je obrada prikupljenih podataka bila odrađena ručno kroz SQL upite radi praktičnosti.

Izrada umjetnih podataka bila je poprilično zahtjevna no na kraju je pronađen odgovarajući model za korisnike aplikacije koji je jednostavan za vizualizirati i usporediti.

Algoritam za preporuku recepata baziran je na K-Means algoritmu te se korisnicima u istoj grupi nude recepti od njihovih susjeda koje oni sami ne posjeduju. Pretpostavka je da bi se korisnicima više sviđali recepti koje posjeduju njima slični ljudi. Uz korištenje rekurzije te funkcija višeg reda postignuta je vrlo elegantna i koncizna definicija K-Means algoritma a uz njega su također implementirane dvije inicijalizacijske metode no njihove performanse na kraju nisu bile previše različite.

Korisnička aplikacija, opisana u projektnom zadatku, napravljena je kao SPA s time da je cijela napisana u Haskellu te je klijentski dio kompiliran u JavaScript. Ovaj pristup smanjio je kognitivni stres i potencijalne greške budući da je kod između servera i klijenta bio dijeljen te su ujedno sve prednosti Haskella dobivene i u klijentskom kodu.

Testiranje koda je bilo provedeno kroz jedinične i svojstvene testove. Svojstveni testovi su bili poprilično praktični jer je bilo dovoljno opisati očekivana svojstva neke funkcije na što bi sustav za testiranje sam napravio i proveo niz testova.

Ukratko, Haskell i funkcijsko programiranje pružaju vrlo moćan set alata i apstrakcija koje garantiraju veću ispravnost koda no to dolazi sa cijenom. Potrebno je naučiti poprilično drugačiju paradigmu od uobičajene a relativno mali broj korisnika povlači i relativno ne razvijen eko sustav što može utjecati na sposobnost razvoja softvera.

Popis kratica

ADT	*Algebraic Data Type*	Algebarski podatkovni tip
API	*Application Programming Interface*	Sučelje za programiranje aplikacija
CSS	*Cascading Style Sheets*	Kaskadni stil
CSV	*Comma Separated Values*	Zarezom odvojene vrijednosti
EDSL	*Embedded Domain Specific Language*	Ugrađeni domeni specifičan jezik
FRP	*Functional Reactive Programming*	Funkcijsko reaktivno programiranje
HTML	*HyperText Markup Language*	Hipertekstualni označni jezik
IO	*Input Output*	Ulazno izlazno
JSON	*JavaScript Object Notation*	JavaScript notacija objekata
OOP	*Object Oriented Programming*	Objektno orijentirano programiranje
SPA	*Single Page Application*	Jedno-stranična aplikacija
SQL	*Structured Query Language*	Strukturirani upitni jezik

Popis slika

Slika 5.1 Prikaz pregleda prikupljenih podataka u programu *DB Browser for SQLite* 17

Slika 5.2 Relacijska shema baze prikupljenih sastojaka 18

Slika 5.3 Stvaranje tablice mjernih jedinica sastojaka putem sučelja programa DB Browser for SQLite 21

Slika 6.1 Graf `sinc` funkcije 25

Slika 6.2 Primjeri grafova funkcije `bias` sa različitim argumentima 26

Slika 6.3 Graf profila 26

Slika 7.1 Graf grupe 01 39

Slika 7.2 Graf grupe 02 40

Slika 7.3 Usporedba prosječnih SCT i PCT vrijednosti prema broju podataka 43

Slika 7.4 Usporedba prosječnih SS i PS vrijednosti prema broju podataka 43

Slika 7.5 Usporedba prosječnih SCT i PCT vrijednosti prema broju atributa 44

Slika 7.6 Usporedba prosječnih SS i PS vrijednosti prema broju atributa 44

Slika 8.1 ER dijagram produkcijske baze podataka 45

Slika 8.2 Elm arhitektura 46

Slika 8.3 Korisničko sučelje za pregled recepata 49

Slika 8.4 Korisničko sučelje za stvaranje novog recepta 49

Popis tablica

Tablica 5.1 Uzorak rezultata pomoćnog upita 20

Tablica 6.1 Mjerenja rezultata funkcija `abp` i `anbp` 30

Tablica 7.1 Tablica rezultata grupiranja korisnika 38

Tablica 7.2 Zaglavlja datoteke rezultata mjerenja 41

Tablica 7.3 Rezultati mjerenja K-Means++ i jednostavnog algoritma inicijalizacije 42

Popis kôdova

Kôd 4.1 Definicija mjerne jedinice. 8

Kôd 4.2 Definicija retka u tablici sastojaka 8

Kôd 4.3 Definicija tipa primarnog ključa 9

Kôd 4.4 *Selda* definicija tablice sastojaka 10

Kôd 4.5 *Selda* funkcija za pohranu novog sastojka u bazi 10

Kôd 4.6 *Selda* upit za pronalazak sastojka prema imenu 10

Kôd 4.7 Definicije `Maybe` i `Either` podatkovnih tipova 11

Kôd 4.8 Definicija strugača popisa *slugova* 12

Kôd 4.9 Definicija strugača sastojka 12

Kôd 4.10 Primjer JSON odgovora sa Konzum Klik API-ja za pretragu 13

Kôd 4.11 Definicija zapisa proizvoda 13

Kôd 4.12 Definicija FromJSON instance za proizvod 14

Kôd 4.13 Tipski potpisi API-ja Coolinarike 15

Kôd 4.14 MimeUnrender instanca tipske klase za listu Slugova 15

Kôd 5.1 Primjer dodavanja `on delete cascade` opcije na već postojeću *SQLite* tablicu 19

Kôd 5.2 Okidač za brisanje pripadajućih proizvoda pri brisanju kategorije 20

Kôd 5.3 Pomoćni SQL upit za dobivanje liste sastojaka sa cijenama i mjernim jedinicama 20

Kôd 6.1 Beskonačna lista naziva sastojaka 22

Kôd 6.2 Generator profila 23

Kôd 6.3 Definicija funkcije sklonosti i funkcije za nasumično generiranje inicijalnih parametara 24

Kôd 6.4 `sinc` funkcija 24

Kôd 6.5 *bias* implementacija u Haskellu 25

Kôd 6.6 Generator sheme recepta 27
Kôd 6.7 Funkcija sklonosti profila korisnika ka nekom receptu 28
Kôd 6.8 Funkcije za provjeru kvalitete stvorenih recepata 29
Kôd 6.9 Funkcija za odlučivanje o odabiru recepta 30
Kôd 7.1 Podatkovni tip za projekciju podataka u vektorski prostor 32
Kôd 7.2 Podatkovni tip grupe podataka 32
Kôd 7.3 Implementacija algoritma k-srednjih vrijednosti 33
Kôd 7.4 Implementacija funkcije grupiranja 33
Kôd 7.5 Funkcija udaljenosti 34
Kôd 7.6 Funkcija za skaliranje vrijednosti 34
Kôd 7.7 Jednostavni algoritam *k-means* inicijalizacije 35
Kôd 7.8 K-Means++ algoritam inicijalizacije 36
Kôd 7.9 Pomoćne funkcije za usporedbu algoritama inicijalizacije grupa 41
Kôd 8.1 Primjer modela korisničke aplikacije 47
Kôd 8.2 Primjer definicije akcija koje je moguće izvršiti unutar aplikacije 47
Kôd 8.3 Dio definicije update funkcije 47
Kôd 8.4 Primjer pogleda forme za prijavu 48
Kôd 9.1 Primjer jediničnih testova za strugače 50
Kôd 9.2 Primjer svojstvenog testiranja 50

Literatura

[1] ŠNAJDER J., BAŠIĆ B. D. *Strojno učenje, Skripta.* Sveučilište u Zagrebu, 2014.

[2] MOHAMAD B.I., USMAN D., *Standardization and Its Effects on K-Means Clustering Algorithm*, Research Journal of Applied Sciences, Engineering and Technology, 2013.

[3] ARTHUR D., VASSILVITSKII S., *k-means++: The Advantages of Careful Seeding*, Proc. of the Annu. ACM-SIAM Symp. on Discrete Algorithms, 2007.

[4] WADLER P., HUDAK P., HUGES J., JONES S.P., A History of Haskell: Being Lazy With Class, Proceedings - Third ACM SIGPLAN History of Programming Languages Conference, HOPL-III. 1-55. 10.1145/1238844.1238856., 2007.