Spark MLlib

modul za mašinsko učenje nad velikim skupovima podataka

Mitar Perović
Fakultet tehničkih nauka
Novi Sad, Srbija

***Abstract*— (*Abstract*) Apache Spark i njegova arhitektura kao odgovor na potrebu za pouzdanim softverom za obradu ogromne količine podataka. Proširenje Sparka za distribuirane algoritme mašinskog učenja nad velikim skupovima podacima kroz modul MLlib.**

Keywords—veliki podaci, spark, mašinsko učenje, spark mllib, paketna obrada

I. Uvod

Podaci su nova nafta. Krilaticu možemo čuti skoro svake godine, još od njenog nastanka (2006. godine). Količina podataka koju generišemo svakog dana se ubrzano uvećava. Kada pogledamo trend količine kreiranih podataka na godišnjem nivou [1] možemo ustanoviti eksponencijalan trend rasta. Brojke su enormne, i estimacija za 2022. godinu je da će biti kreirano oko 97 zetabajta podataka. Radi perspektive jedan zetabajt iznosi milijardu terabajta. Drukčija perspektiva, koja govori o udvostručavanju ljudskog znanja predstavlja ilustrativniji pregled. Ako pisane tragove podataka posmatramo kao znanje, estimirano je da se do dvadesetog vijeka, ljudsko znanje udvostručavalo svakih 100 godina. Danas se taj isti vremenski okvir spustio na samo 13 sati [2]. Uzevši navedeno u obzir, javlja se i prijeka potreba za novim, adekvatnim pristupima za čuvanje i obradu enormne količine podataka. U ovom radu ćemo se baviti isključivo obradom, konkretnije držaćemo se okvira Apache Spark alata. Glavni cilj obrade podataka je pokušaj izvlačenja znanja. Život u eri "vještačke inteligencije", gdje nam u velikoj mjeri algoritmi određuju sadržaj koji konzumiramo i olakšavaju segmente života, ne bi bio moguć bez tehnika mašinskog učenja. Spajanje velike količine podatka i standardnih algoritama mašinskog učenja je veliki izazov. Najpopularnije rešenje su Spark alat i njegova biblioteka *MLlib*.

A. Apache Spark

Jedan od trenutno najpoznatijih alata koji je nastao iz iz potrebe za obradom ogromne količine podataka, jeste upravo Apache Spark. Projekat je otvorenog koda (*open source*) i nastao je na univerzitetu *Berkeley (University of California)*. Sa razvojem je otpočeto 2009. godine na univerzitetu, da bi 2013. bio predat Apache fondaciji [3], i trenutno je aktuelna treća verzija alata. Popularnosti ovog alata svedoči činjenica da 80% *Fortune 500* kompanija koristi ovaj alat za obradu svojih podataka. Glavne karakteristike Sparka iz kojih je proistekla njegova popularnost su brzina i dostupnost. Neki od modula, poput modula za mašinsko učenje dostižu ubrzanje i do 100 puta u odnosu na alternativni program iste funkcionalnosti (*Hadoop MapReduce*). Dostupan je u više programskih jezika, i to: *Python, Java, Scala, C#, F#, R i SQL.* Sam Spark sadrži nekoliko modula za specifične slučajeve korišćenja, kao što su: Spark *SQL* (izvršenje upita), *GraphX* (za distribuiranu obradu grafova), *Structured Streaming* (za obradu tokova podataka), kao i *MLlib* - modul za mašinsko učenje kojim se bavimo u ovom radu.

Slika 1.1 Prikaz Sparkovih različitih modula na lijevoj strani, i programskim jezicima u kojima je dostupan za korišćenje, na destnoj strani. Izvor: link

B. Mašinsko učenje

Najčešće mašinsko učenje definišemo kao primjenu vještačke inteligencije, kako bi program mogao da uči sam kroz iskustvo, bez eksplicitnog programiranja, i to na osnovu podataka. Možemo posmatrati kao drukčiju paradigmu u odnosu na programiranje. U klasičnom programiranju zadajemo set koraka, koji će na osnovu određenog ulaza dati određeni izlaz. Kod mašinskog učenja imamo već konkretne ulaze, tačnije trening skup podataka, i konkretan krajnji rezultat (na primjer godišnja zarada neke osobe). Cilj nam je da pronađemo određene obrasce u podacima, na osnovu kojih

možemo raditi buduća predviđanja. Gruba podjela mašinskog učenja je na nadgledano (kada imamo oznake/labele svake instance podatka) i nenadgledano (kada ne postoje oznake). U poglavlju o funckionalnostima MLlib modula, biće više riječi o specifičnim zadacima mašinskog učenja poput klasifikacije, regresije, klasterovanja, itd.

U nastavku će biti dat pregled samog alata Spark, kao i kratak uvid u mašinsko učenje. Nakon toga, u drugom poglavlju će biti detaljnije opisana sama arhitektura alata Spark, kao i principi funkcionisanja i uklapanje modula za mašinsko učenje u Spark Core. Različite funkcionalnosti modula MLlib biće opisane u trećem poglavlju. Nakon uvida u mogućnosti MLliba i arhitekturu Sparka, biće opisan konkretan primjer korišćenja MLlib biblioteke u domenu procesiranja prirodnog jezika (NLP). Poslednje poglavlje predstavlja sumarizaciju cjelokupnog rada.

II. Spark Arhitektura

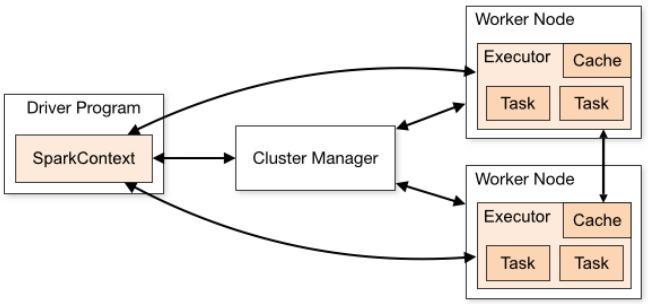
Kako je ranije rečeno, Spark je program za distribuiranu obradu podataka. Samu srž predstavlja Spark Core, na koji se nadograđuju svi prethodno navedeni dodatni moduli. Bavi se različitim ključnim operacijama. Konkretno, bavi se upravljanjem memorijom, oporavkom od otkaza, *scheduling*-om, distribuiranjem i monitoringom poslova na klasteru kao i interakcijom sa sistemima za skladištenje podataka.

Spark se izvršava na klasteru, gdje imamo jedan glavni čvor (*master node)* i proizvoljan broj radnih čvorova (*slave/worker nodes*). Na master čvoru se izvršava Spark Driver. Tu se kreira Spark Context, koji predstavlja neku vrstu veze korisničkih instrukcija ka samom Spark programu. Analogno možemo posmatrati kao bazu podataka, tačnije DBMS, i otvorenu konekciju ka toj bazi. Spark Driver ima različite komponente poput *DAGSchedulera*, *TaskSchedulera*, *BackendSchedulera*, *BlockManagera* koji su zaduženi za prebacivanje korisničkog programskog koda u zadatke koje izvršavaju Spark radni čvorovo na klasteru. Spark Driver radi dva zadatka, tačnije konvertuje korisničke programe u zadatke i planira izvršenje zadataka kod izvršioca.

Driver na master čvoru radi otprilike sledeći niz zadataka:

1. Zakazuje izvršenje programa i pregovara sa Klaster menadžerom
2. Prevodi RDDjeve u graf izvršenja, koji razbija na više faza
3. Drajver čuva metapodatke o svim *RDDj*evima i njihovim particijama
4. Razbija zadatak na taskove
5. Nakon završetka taskova svi radnici šalju svoje rezultate Drajveru.
6. Drajver izlaže rezultate

Dvije glavne apstrakcije pomoću kojih je definisana arhitektura su *RDD* (*Resilient Distributed Datasets*) i *DAG* (*Directed Acyclic Graph*). O njima će biti riječi u nastavku.



Slika 2.1 Prikaz arhitekture izdijeljene na čvorove

Spark podržava dvije vrste operacija. Podržava transformacije i akcije. Transformacije, kao što sam naziv govori, su vrste operacija koje od početnog *RDD*-a, vraćaju nov, izmijenjen *RDD*. U nastavku ćemo opisati par primjera.

Primjeri transformacija su:

- *map()* - operacija koja prima funkciju i *RDD*, zatim nad svakim elementom *RDD*a primjenjuje funkciju i vraća novi *RDD*. Za svaki element povratna vrijednost je jedan element
- *flatMap()* - slično kao *map,* uz razliku da povratna vrijednost može biti više elemenata, a ne samo jedan kao kod *map* operacije

Bitno ja napomenti da su transformacije *lazy*, odnosno da se rezultat evaluira tek nakon poziva neke od akcija.

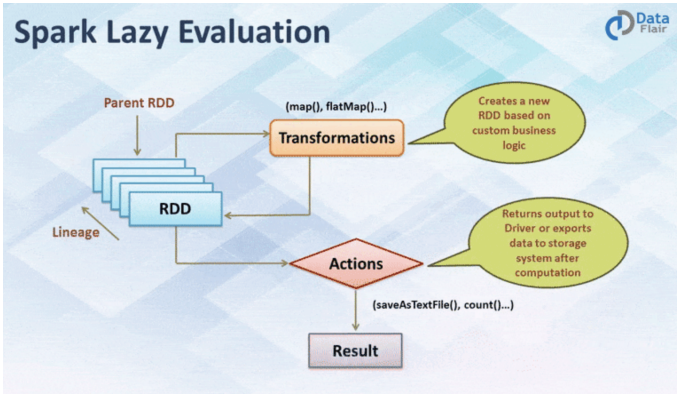
Postoje dvije vrste transformacija:

- uske (*narrow*) - kod ovakvih vrsta transformacija, svi elementi koji su potrebni za izračunavanje sledeće particije *RDD*a se nalaze u istoj particiji roditeljskog *RDD*a.
- široke (*wide*) - naspram uskih, kod širokih transformacija, svi elementi particije roditeljskog *RDD*a se mogu naći u više particija *RDD*a potomka.

S druge strane, neki od primjera akcija su:

- *saveAsTextFile(path)* - upisuje elemente dataseta u tekstualni fajl (ili skup tekstualnih fajlova) u zadati direktorijum. Spark poziva *toString* metodu svakog elementa kako bi ga konvertovao u liniju teksta
- *count()* - vraća broj elemenata u *RDD*u

Rezultat akcija se se smješta na park Driver ili na eksterni fajl sistem, poput HDFSa.

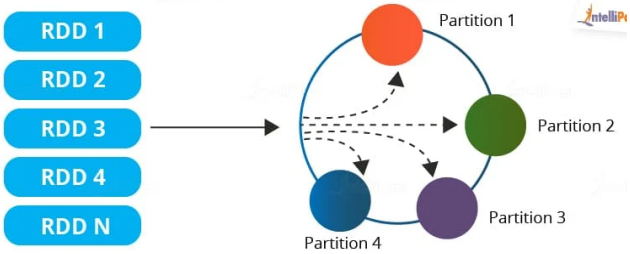


Slika 2.2 Grafički prikaz ciklusa obrade podataka pomoću Spark operacija. Izvor: link

A. *Model podataka*

Prvobitni način rezprezentovanja podataka u Sparku je pomoću *Resilient Distributed Datasets (RDD)* tipa podataka. On je dostupan od verzije 1.0, dok je u verziji 1.3 uveden nov tip *DataFrame*, i u verziji 1.6 i tip podataka *DataSets*.

Glavna odlika *RDD*jeva je njihova otpornost na otkaze. U suštini predstavljaju particionisane skupove podataka koji su replicirani na različite čvorove. Podaci se ne mogu mijenjati, particije su *read-only*. U njih možemo smještati i struktuirane, kao i nestruktuirane podatke, ali ne postoji šema podataka. Još neke od mana su što ne možemo koristiti Sparkove optimizatore izvršenja/upita. Kako su podaci u *RDD*-ju obični *Java* objekti, implicitno se koristi *Garbage collector,* za upravljanje memorijom, kao i serijalizacija *Java* objekata.



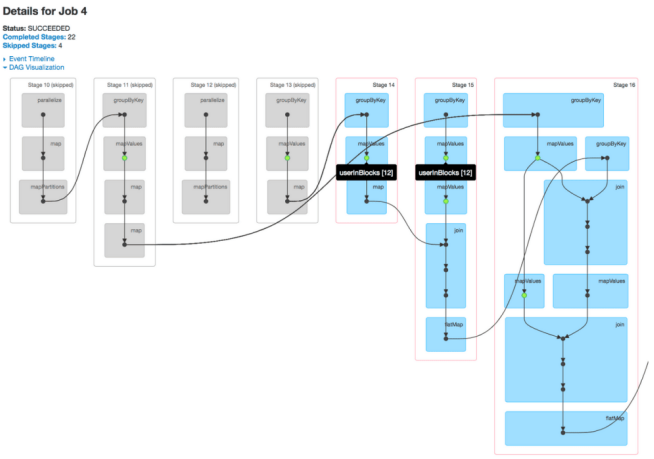
Slika 2.3 Prikaz više RDDjeva i podjela jednog na particije. Izvor: link

Dve godine kasnije, u verziji Sparka 1.3, uvedeni su *DataFrame*-ovi. Glavni cilj je bilo poboljšanje performansi, kao i intuitivnije korišćenje. Podaci su organizovani u kolone i redove, imamo tabelaran pregled podataka. Svaka kolona u *DataFrame*-u ima asocirano ime i tip. Koristi se *Catalyst*, kao optimizator pri unosu. Podaci se čuvaju van glavnog *heap*-a u Javi, ali i dalje su u RAMu.

Poslednje poboljšanje u manipulaciji podataka u Sparku su *DataSet*-ovi. Predstavljaju nadogradnju *DataFrame*-a. Omogućavaju interfejs objektno orijentisanog programiranja, uvođenjem koncepata klasa i objekata.

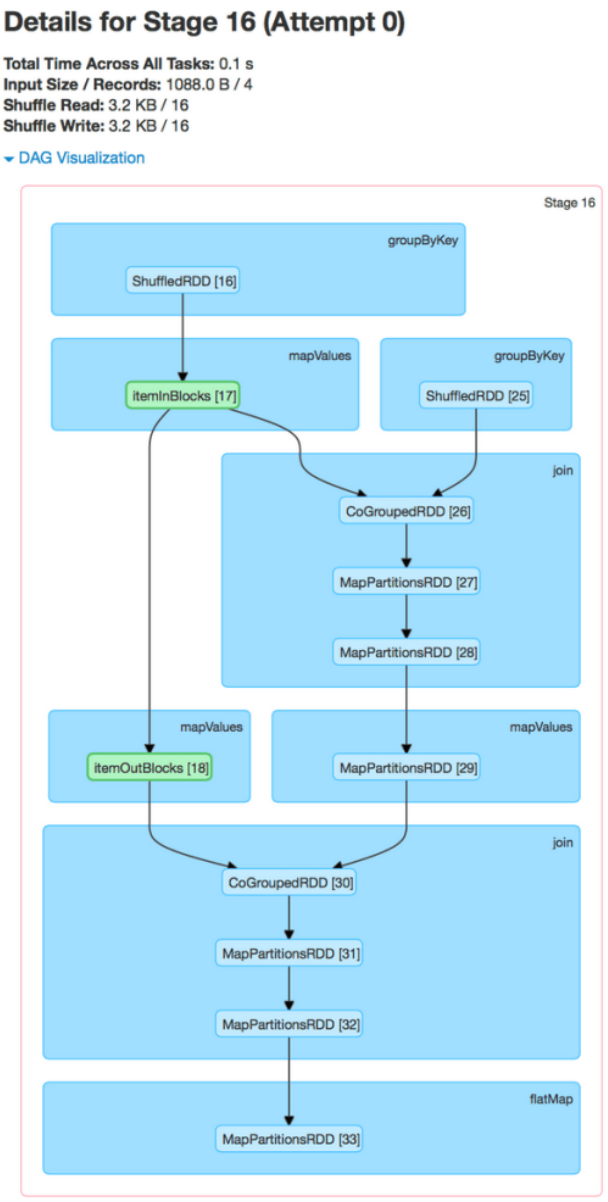
B. *MLlib u okviru Sparka*

Sparkov MLlib modul koristi primarno *DataFrame API*, od verzije Sparka 2.0. Podatke učitavamo u *DataFrame,* nebitno da li iz nekog postojećeg *RDD*-a, da li iz *Hive* tabele, ili obične CSV datoteke. Kako koristimo *DataFrame API,* koji je dostupan preko Spark SQL modula, za kreiranje strukture podataka i šeme, potrebni su nam elementi iz Spark SQLa. Na početku izvšavanja ML pipeline-a, MLlib modul komunicira sa Spark Driverom, koji sadrži Spark Context. Kada se podaci proslijede u napravljeni ML pipeline, interno se sve svodi na već opisanu arhitekturu rada Spark programa. Podaci iz *DataFrame*-a se svode na *RDD*-jeve, nad kojima se vrši veći broj transformacija i akcija. Zadatak se transformiše u DAG, zatim se razbija na faze. Spark Driver iskomunicira sa Klaster menadžerom, na koji radni čvor ide koji zadatak. U nastavku je prikazan primjer ALS (*alternating least squares*) algoritma u mašinskom učenju. Ovaj algoritam se koristi kod sistema za preporuku, i u suštini radi faktorizaciju matrice na proizvod dvije manje matrice. Razbija se na veći broj *map, join, groupByKey* operacija interno.



Slika 2.4 Prikaz internog rada Spark MLlib modula i ALS algoritma. Izvor: link

Ovaj prikaz je dostupan u Spark UI dodatku. Klikom na bilo koju fazu, možemo dobiti još detaljniji prikaz, kao na slici ispod.



Slika 2.5 Detaljni prikaz jedne faze internog rada Spark MLlib modula i ALS algoritma. Izvor: link

III. Funckionalnosti MLlib modula

Komponente koje čine modul *MLlib* možemo izdjeliti na četiri podmodula:

- Algoritme
- *Pipeline* - "protočna obrada"
- *Featurization* - rad sa obilježjima
- *Utilities* - dodatne statističke obrade

A. *Algoritmi*

Upotrebom tehnika mašinskog učenja, najčešće nam je cilj da predvidimo buduću numeričku vrijednost ili pripadnost klasi nekog entiteta, odnosno formalnije, vršimo regresiju i klasifikaciju. U *MLlib*-u je podržan veliki broj popularnih algoritama koji se koristi u ove svrhe.

Jedan od osnovnih "slabih" klasifikatora je logistička regresija. Podržane su i binarna kao i višeklasna klasifikacija kroz logističku regresiju. Takođe, putem parametara je moguće optimizovati regularizaciju, kao npr. kroz *elasticNetParam*. Zadajemo takođe i maksimalan broj iteracija, kao kriterijum zaustavljanja. Moguće je pristupiti i detaljima istreniranog modela na kraju.

Kao proširenje logističke regresije imamo i višeslojni *Perceptron* model. On je osnova neuronskih mreža. Njegov API za korišćenje je vrlo jednostavan u MLlibu. Zadajemo maksimalan broj iteracija, listu koja redom sadrži broj neurona u svakom sloju, kao i *seed* za početno generisanje težina. Specijalni slučajevi neuronskih mreža, kao što su konvolutivne neuronske mreže ili rekurentne mreže, nisu direktno podržane. Samim tim rad sa nestruktuiranim podacima pomoću *state-of-the-art* modela je ograničen u osnovnoj verziji *MLlib*-a. Od verzije Sparka 3.0, podržana je integracija sa *Tensorflow* bibliotekom [4].

Pored ovih modela, podržani su i detaljno dokumentovani i algoritmi poput SVM (*support vector machines*), Naivnog Bajesa, kao i algoritmi zasnovani na stablima.

S druge strane, problemi vezani za struktuirane podatke se uglavnom rešavaju upotrebom *gradient boosted* algoritama zasnovanih na stablima odluke (*decision trees*). Iako nisu implementirani najpoznatiji algoritmi poput *XGBoost*-a, mogu se lako proširiti [5]. S druge strane implementirani su "obični" ansamlb algoritmi koji rade sa stablima odlučivanja. Kroz model *GradientBoostedTrees* možemo koristiti model i za regresiju putem *trainRegressor()* konstruktora, kao i za klasifikaciju putem *trainClassifier()* konstruktora.

Pored najčešćih zadataka kao što su regresija i klasifikacija, modul *MLlib* podržava i algoritme za klasterovanje. Podržani su neki od najpopularnijih algoritama koji su koriste u ove svrhe, kao što su *K-means, LDA (latent dirichlet allocation), Gaussian mixture models*, i *Power iteration clustering*.

Još jedan od zanimljivih slučajeva korišćenja mašinskog učenja su sistemi za preporuku. U *MLlib*-u takođe podržani najpopularniji algoritmi koji su koriste u ove svrhe. Možemo raditi kolaborativno filtriranje kroz prije pomenuti *ALS* algoritam. Takođe je implementiran *FP-growth (frequent pattern mining)* algoritam, koji nam služi za izvlačenje pravila, odnosno implikacija iz skupa podataka.

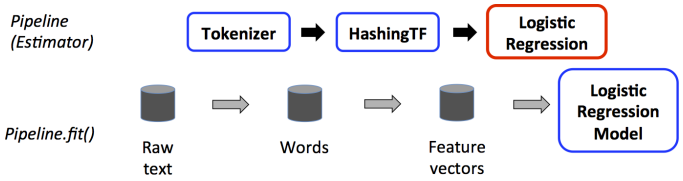
B. *Pipeline*

Koncept *pipeline*-a nam omogućava da izgradimo niz koraka obrade nad *DataFrame*-om. Kombinujemo više koraka i algoritama obrade u jedan uniforman niz, koji možemo lakše optimizovati kasnije. Osnovna jedinica koja sadrži podatke je

DataFrame. Nad podacima koristimo određene *transformere,* koji nam služe za transformaciju početnih atributa (ne radi se o modelu zasnovanom na *attention*-u). Kako sam *Dataframe* nije promjenjiv, rezultat transformacije nam je novi *DataFrame*, koji npr. ima novu kolonu. Transformacija se vrši pozivom metode *transform()*.

Sledeći korak u *pipeline*-u je dodavanje *estimator*-a, odnosno modela koji će vršiti neko predviđanje. Ovo je samo apstrakcija algoritma pomenutih u prošloj sekciji. Svaki *estimator* ima metodu *fit(),* koja prima *DataFrame* sa podacima i kao rezultat vraća istrenirani model.

Jedan od primjera koji su koristi u procesiranju prirodnog jezika je niz obrada, gdje na ulazu imamo dokument (može biti i rečenica/paragraf, ovo je čisto konvencija) koji razbijamo na riječi. Sva velika početna slova konvertujemo u mala i zatim ih konvertujemo u numerički reprezent, kako bi algoritmi mogli raditi s njima. Kada imamo vektorizovan tekst, možemo obučiti *estimator,* kao što je npr. SVM ili logistička regresija za predviđanje sentimenta. Na slici ispod se nalazi takva ilustracija.



Slika 3.1 Primjer NLP pipeline-a u obradi teksta. Izvor: link

Najčešći benefit korišćenja ML *pipeline*-ova je lakša optimizacija hiper-parametara. Cjelokupni *pipeline* se treitra kao *estimator* i prosleđuje se npr. *grid search* algoritmu radi pretrage optimalnih parametara.

C. Featurization

Za rad sa obilježjima su takođe podržani većinski popularni algoritmi. Govorili smo o *transformerima* kod *pipeline*-ova, i oni spadaju u ovaj podmodul. Pored transformacija možemo raditi i ekstrakciju obilježja, kao i selekciju.

Algoritmi koji od sirovog teksta vraćaju vektorizovani numerički reprezent su obuhvaćeni u MLlibu. Tako imamo od jednostavnijih *bag of words* modela, kao što je *TF-IDF* ekstraktor, pa do *word2vec* modela koji sadrži određenu semantiku. Nažalost nisu podržani istrenirani modeli *word2vec*, već nam je samo omogućeno da na isti način dobijemo vektore iz našeg tekstualnog korpusa. Možemo reći da je rad sa tekstom iz ovog razloga dosta ograničen sa osnovnom MLlib bibliotekom. Fale istrenirani *embedding* vektori iz *word2vec* modela, zatim *elMo*, ili čak i najpopularniji *transformer (BERT* npr.) *embedding* vektori. Za rad sa Sparkom, preporuka je koristiti drugu biblioteku koja ima više *state-of-the-art* funkcionalnosti za ekstrakciju obilježja [6].

Pored ekstrakcije, govorili smo o transformaciji obilježja. Podržani su skoro svi postupci koji se koriste u praktičnim situacijama. Od *Tokenizer*-a koji razbija rečenice na tokene (najčešće riječi), pa sve do *StopWordRemover*-a (uklanjanje riječi koje se prečesto ponavljaju i nisu nam od značaja, tzv. *stop* riječi) i *OneHotEncoder*-a (konverzija kategoričkog obilježja, npr države rođenja u numerički - vektor), svi neophodni algoritmi za transformaciju su podržani.

Za selekciju obilježja možemo koristiti *VectorSlicer*. Koristimo ga da od kolone koja sadrži atribute, proslijedimo indekse atributa koje želimo da zadržimo. To nam je najjednostavniji vid selekcije obilježja, koju možemo raditi uz oslonac na znanje domenskog eksperta. Postoje i sofisticiraniji selektori obilježja, koji koriste određeno statistčko znanje pri selekciji. Tako imamo npr. *ChiSqSelector* klasu koja pomoću hi-kvadratne distribucije određuje koja su obilježja od značaja. Imamo takođe i *VarianceThresholdSelector* koji na osnovu zadatog praga varijanse odbacuje ili zadržava pojedinačno obilježje.

D. Dodatne funkcionalnosti

Pored standardnih tehnika mašinskog učenja, u MLlib biblioteku implementirane su i tehnike statističkih obrada podataka. Implementirani je testiranje hipoteze. U statistici se često koristi za ispitivanje statističke značajnosti nekog ishoda. U ove svrhe implementiran je *ChiSquareTest,* pomoću kojeg radimo hi-kvadratni test nulte hipoteze. Takođe, postoji i *Summarizer* klasa za deskriptivnu statistiku. Nakon prosljeđivanja *dataframe*-a sa podacima, možemo pozivati *summary()* metodu i konkretne statistike od interesa, kao što su srednja vrijednost (*mean*), minimum/maksimum, varijansa, itd.

IV. Primjer korišćenja

U ovomo poglavlju biće opisan poseban slučaj korišćenja MLlib modula u domenu procesiranja prirodnog jezika [7]. Nad korisničkim recenzijama će biti odrađeno procesiranje riječi, tačnije:

- tokenizacija - razbijanje rečenica na pojedinačne riječi pomoću *RegexTokenizer*-a
- uklanjanje riječi bez semantičkog značaja pomoću *StopWordRemover*-a
- vektorizacija recenzija pomoću *Word2Vec* modela - treniranje modela nad svim riječima u korpusu, zatim pravljenje *embedding*-a recenzije kao usrednjenog prosjeka vektora pojedinačnih riječi

Učitavanje podataka u *DataFrame* možemo uraditi iz običnog CSV fajla, ili iz *Hive* tabele. U nastavku je dat primjer učitvanja iz *Hive* tabele, gdje su nam kolone od interesa su nam samo “business_id” i “text” koji predstavlja samu recenziju.

Kolona od značaja nad kojom radimo procesiranje je “business”. Pomoću tokenizatora ćemo razbiti recenziju na više riječi.

```
from pyspark.ml.feature import RegexTokenizer

regexTokenizer = RegexTokenizer(gaps = False, pattern = '\w+', inputCol = 'text', outputCol = 'text_token')

reviews_groupedby_business_token=regexTokenizer.transform(reviews_groupedby_business)

reviews_groupedby_business_token.show(3)
```

Kao rezultat ćemo pored kolone "text" sada imati još jednu kolonu "text_token" u novom *DataFrame*-u koji je kreiran pod nazivom *reviews_groupedby_business_token*.

```
+--------------------+--------------------+--------------------+
|         business_id|                text|          text_token|
+--------------------+--------------------+--------------------+
|Y8C2FuA0Aiy7uMgXX...|Bad haircut, filt...|[bad, haircut, fi...|
|FeXsYKhyJ7F8t0-kn...|I have gone to th...|[i, have, gone, t...|
|CrIWqmuO2uQWwl3z1...|I'm picky on pizz...|[i, m, picky, on,...|
+--------------------+--------------------+--------------------+
```

Slika 4.1 Novi DataFrame sa kolonom tokena

U sledećem koraku izbacujemo sve *stop* riječi iz recenzija.

```
swr = StopWordsRemover(inputCol = 'text_token', outputCol = 'text_sw_removed')

reviews_swr= swr.transform(reviews_groupedby_business_token)

reviews_swr.show(3)
```

I kao rezultat dobijamo *DataFrame* sa novom kolonom koja je rezultat *StopWordRemover.transform()* poziva.

```
+--------------------+--------------------+--------------------+--------------------+
|         business_id|                text|          text_token|     text_sw_removed|
+--------------------+--------------------+--------------------+--------------------+
|Y8C2FuA0Aiy7uMgXX...|Bad haircut, filt...|[bad, haircut, fi...|[bad, haircut, fi...|
|FeXsYKhyJ7F8t0-kn...|I have gone to th...|[i, have, gone, t...|[gone, grocery, s...|
|CrIWqmuO2uQWwl3z1...|I'm picky on pizz...|[i, m, picky, on,...|[picky, pizza, fa...|
+--------------------+--------------------+--------------------+--------------------+
```

Slika 4.2 Novi DataFrame sa kolonom koja sadrži tokene bez stop words-a

Kada imamo tokene koji su očišćeni možemo ih proslijediti *word2vec* modelu koji će napraviti vektore za svaki od pojedinačnih riječi u korpusu. Vektor jedne recenzije će biti dužine 100 i predstavljaće prosjek vektora svih riječi iz *text_sw_removed* kolone.

```
word2vec = Word2Vec(vectorSize = 100, minCount = 5, inputCol = 'text_sw_removed', outputCol = 'result')

model = word2vec.fit(reviews_swr)

result = model.transform(reviews_swr)
```

Sada u novom *DataFrame*-u imamo novu kolonu "result". U nju je smješten vektor cijele recenzije.

```
+--------------------+--------------------+--------------------+--------------------+--------------------+
|         business_id|                text|          text_token|     text_sw_removed|              result|
+--------------------+--------------------+--------------------+--------------------+--------------------+
|Y8C2FuA0Aiy7uMgXX...|Bad haircut, filt...|[bad, haircut, fi...|[bad, haircut, fi...|[-0.0540009386138...|
|FeXsYKhyJ7F8t0-kn...|I have gone to th...|[i, have, gone, t...|[gone, grocery, s...|[-0.0259041081774...|
|CrIWqmuO2uQWwl3z1...|I'm picky on pizz...|[i, m, picky, on,...|[picky, pizza, fa...|[-0.0585137852811...|
+--------------------+--------------------+--------------------+--------------------+--------------------+
```

Slika 4.3 Rezultat word2vec modela je novi DataFrame sa kolonom result u koju je smješten vektor recenzije

Dobijeni vektori recenzija bi trebali imati semantiku same recenzije. Dalje se mogu koristiti za traženje sličnosti između samih recenzija, eventualno klasterovanje kako bismo vidjeli da li postoje određene grupe sličnih recenzija i zatim ručno da utvrdimo zbog čega. Problem je što nam recenzije nisu anotirane sentimentom, pa ne možemo raditi nadgledano učenje nad njima.

Eventualno proširenje se nalazi u korišćenju Spark NLP biblitoeke [6], koja sadrži pretrenirane modele nad tekstom. Pored *word2vec* modela sadržani su i napredniji *embedding* modeli, kao i *state-of-the-art Transformer* modeli poput *BERT*-a.

V. Zaključak

Generisanje ogromnih količina podataka ne pokazuje tendenciju usporavanja. Tehnike mašinskog učenja pomoću kojih ekstrahujemo znanje iz podataka i pokušavamo uočavati obrasce iz prošlosti, na osnovu kojih predviđamo budućnost, takođe se razvijaju rapidnom brzinom. U radu je pokazano da je Spark i njegov modul MLlib jako dobar odgovor na pomenute zahtjeve. Interna arhitektura samog Sparka omogućava veliko ubrzanje i otpornost na greške. Nažalost, vidjeli smo da sama biblioteka MLlib ne drži korak sa vremenom, pa većina *state-of-the-art* modela nije implementirana, već se mora uključiti drugi modul. Dobre vijesti su da i sam Spark evoluira i izlazi u susret novim zahtjevima. Od verzije 3.0 je predstavljen nov način izvršavanja zadataka, tzv *barrier execution mode*. Potreba za tim je prvenstveno nastala iz adaptacije *deep learning* modela. Međutim implementacija i detalji samog načina rada mogu biti tema proširenja, eventualno novog rada.

Reference

[1] • Total data volume worldwide 2010-2025 | Statista

[2] Knowledge Doubling Every 12 Months, Soon to be Every 12 Hours - Industry Tap

[3] Apache Spark™ history

[4] Spark TensorFlow Distributor

[5] XGBoost4J-Spark Tutorial (version 0.9+) — xgboost 1.5.1 documentation

[6] Spark NLP

[7] Word2Vec_Content_Final - Databricks