Algoritmi, tehnike, biblioteke i savremeni alati u praksi

Mašinsko učenje sa C++

PREVOD DRUGOG IZDANJA

Kiril Kolodiažnij

kompjuter biblioteka

packt

Mašinsko učenje sa C++

PREVOD DRUGOG IZDANJA

Ova knjiga, čiji je autor iskusni softverski inženjer sa dugogodišnjim iskustvom u industriji, objašnjava osnove mašinskog učenja i prikazuje načine primena C++ biblioteka za izradu modela nadgledanog i nenadgledanog učenja.

Steći ćete praktično iskustvo u podešavanju i optimizaciji modela za različite namene, što vam omogućava da efikasno birate odgovarajuće modele i merite njihove performanse. U knjizi su detaljno objašnjene tehnike kao što su preporučivanje proizvoda, ansambl učenje, detekcija anomalija, analiza osećanja i prepoznavanje objekata, uz primenu savremenih C++ biblioteka. Takođe ćete naučiti da rešavate izazove prilikom implementacije modela na mobilnim platformama i saznate kako format modela ONNX olakšava taj proces.

Ovo izdanje je unapređeno važnim temama, među kojima su primena analize osećanja kroz učenje prenosom znanja i modeli zasnovani na transformatorima, kao i praćenje i vizuelizacija eksperimenata mašinskog učenja pomoću alata MLflow. U dodatnom poglavlju objašnjena je upotreba biblioteke Optuna za izbor hiperparametara. Deo posvećen implementaciji modela na mobilnim platformama sada sadrži detaljno objašnjenje prepoznavanja objekata u realnom vremenu na Android sistemu pomoću C++ programskog jezika.

Kada pročitate ovu knjigu o mašinskom učenju i programskom jeziku C++, steći ćete praktično znanje iz oblasti mašinskog učenja i veštinu korišćenja jezika C++ za razvoj naprednih sistema mašinskog učenja.

Šta ćete naučiti

- Primenu najvažnijih algoritama mašinskog učenja uz različite C++ biblioteke.
- Učitavanje i obradu različitih tipova podataka u odgovarajuće C++ strukture podataka.
- Prepoznavanje optimalnih parametara modela mašinskog učenja.
- Primenu otkrivanja anomalija za filtriranje korisničkih podataka.
- Upotrebu saradničke filtracije za upravljanje promenljivim korisničkim preferencijama.
- Upotrebu C++ biblioteka i API interfejsa za upravljanje strukturom i parametrima modela.
- Implementaciju C++ koda za prepoznavanje objekata pomoću savremene neuronske mreže.

Ova knjiga je namenjena početnicima koji žele da istraže algoritme i tehnike mašinskog učenja upotrebom programskog jezika C++. Takođe će biti korisna analitičarima podataka, naučnicima i programerima koji žele da primene modele mašinskog učenja u proizvodnim okruženjima. Da bi se u potpunosti iskoristio sadržaj knjige, potrebno je osnovno poznavanje jezika C++.

REČ UREDNIKA

*Nakon prevođenja, knjigu je recenzirao **Miroslav Ristić**, redovni profesor na Prirodno-matematičkom fakultetu Univerziteta u Nišu, sa preko 25 godina iskustva u razvoju softvera. Tako da je ova knjiga prošla kroz njegovo stručno vrednovanje, sa ciljem da se osigura da prevod bude ne samo jasan, precizan i prilagođen čitaocima, već i da održi visok kvalitet i stručnu relevantnost knjige.*

Skenirajte QR kod, registrujte knjigu osvojite nagradu

ISBN: 978-86-7310-610-6

Mašinsko učenje sa C++

Algoritmi, tehnike, biblioteke i savremeni alati u praksi

Kiril Kolodiažnij

kompjuter biblioteka

Packt›

Izdavač:

kompjuter biblioteka

Obalskih radnika 4a
Beograd, Srbija
Tel: 011/2520272
e-pošta: kombib@gmail.com
veb-sajt: www.kombib.rs

Za izdavača:
Mihailo J. Šolajić, urednik

Autor:
Kiril Kolodiažnij

Prevod: Nemanja Lukić
Recezent: Miroslav Ristić
Slog: Zvonko Aleksić

Znak Kompjuter biblioteke:
Miloš Milosavljević

Štampa: „Pekograf“, Zemun
Tiraž: 500
Godina izdanja: 2025.

Broj knjige: 587
Izdanje: Prvo
ISBN: 978-86-7310-610-6

Naslov originala:

Hands-On Machine Learning with C++
Second Edition
ISBN 978-1-80512-057-5

Packt Publishing Ltd.
Birmingham, UK, packt.com

Mašinsko učenje sa C++

Autorizovani prevod sa engleskog jezika.

CIP - Каталогизација у публикацији
Народна библиотека Србије, Београд

004.85
004.432.2C++

КОЛОДИАЖНИЈ, Кирил

Mašinsko učenje sa C++: algoritmi, tehnike, biblioteke i savremeni alati u praksi / Kiril Kolodiažnij; [prevod Nemanja Lukić]. - 1. izd. - Beograd: Kompjuter Biblioteka, 2025 (Zemun: Pekograf). - XXV, 479 str.: ilustr.; 24 cm. - (Kompjuter biblioteka; br. knj. 587)

Prevod dela: Hands-On Machine Learning with C++. - Tiraž 500. - O autoru: str. III. - Registar.

ISBN 978-86-7310-601-4 (broš.)

a) Машинско учење
b) Програмски језик „C++“

COBISS.SR-ID 165412873

SARADNICI

O AUTORU

Kiril Kolodiažnij je iskusan softverski inženjer i stručnjak za razvoj prilagođenih softverskih rešenja. Ima višegodišnje iskustvo u izradi modela mašinskog učenja i razvojnih rešenja zasnovanih na podacima u programskom jeziku C++. Diplomirao je računarske nauke na Nacionalnom univerzitetu za radioelektroniku u Harkovu.

O RECENZENTIMA

Haršit Džain je strastveni inženjer veštačke inteligencije sa bogatim iskustvom u razvoju najnaprednijih rešenja veštačke inteligencije. Radio je na globalnim projektima i sarađivao s međunarodnim timovima na razvoju naprednih rešenja iz oblasti veštačke inteligencije. Njegov rad obuhvata i društveno odgovorne projekte zaštite divljih životinja i očuvanja životne sredine. Zahvaljujući svojim analitičkim sposobnostima i dubokom poznavanju veštačke inteligencije, značajno je doprineo inovacijama u obradi podataka, prepoznavanju slika i razvoju naprednih jezičkih modela. Strastveni je putnik i ljubitelj automobila, voli da kuva i da upoznaje različite kulture kroz gastronomiju i književnost.

Lalitkumar Prakaščand je iskusan softverski inženjer sa decenijskim iskustvom u radu na razvoju poslovnih aplikacija visokih performansi. Stručnjak je za mikroservise, distribuirano računarstvo, mašinsko učenje i veštačku inteligenciju i značajno je doprineo uspehu vodećih tehnoloških kompanija, kao što su **Meta** i **Careem**. Imao je važnu ulogu u izgradnji robusnih mikroservisa i optimizaciji najznačajnijih platformskih funkcionalnosti, što je dovelo do značajnih poboljšanja u performansama sistema i angažovanju korisnika. Kao recenzent knjige *Mašinsko učenje sa C++*, ponosan je što svojim znanjem i iskustvom doprineo tome da se čitaocima obezbedi sadržaj najvišeg kvaliteta.

Hemanat Kumar J je stručnjak za analizu podataka sa bogatim iskustvom u razvoju i primeni modela mašinskog učenja, generativne veštačke inteligencije, vizuelizacije podataka i analitičkih rešenja. Radio je u sektorima transporta, obrazovanja, finansija i zdravstva, gde je dosledno razvijao rešenja zasnovana na podacima i unapredio procese donošenje odluka visokom preciznošću i operativnom efikasnošću. Kao tehnički recenzent knjige *Mašinsko učenje sa C++*, svojim znanjem doprineo je tačnosti i razumljivosti sadržaja. Posebnu zahvalnost izražava porodici, mentorima i prijateljima na podršci tokom ovog projekta.

Miroslav Ristić je redovni profesor na Prirodno-matematičkom fakultetu Univerziteta u Nišu, sa preko 25 godina iskustva u razvoju statističkog softvera. Posebno se ističe njegov rad na razvoju grafičkog korisničkog interfejsa R Commander za programski jezik R. Dugi niz godina recenzirao je značajan broj knjiga za izdavačku kuću Springer i časopis Journal of Applied Statistics. Od 2023. godine aktivno recenzira najaktuelnija izdanja izdavačke kuće "Kompjuter biblioteka". Nakon prevođenja, svako izdanje prolazi kroz njegovo stručno vrednovanje i recenziju prevoda, sa ciljem da se osigura da prevodi budu ne samo jasni, precizni i prilagođeni čitaocima, već i da održe visok kvalitet i stručnu relevantnost knjiga.

Kratak sadržaj

Sadržaj

POGLAVLJE 2

POGLAVLJE 3

POGLAVLJE 14

Predgovor

Programski jezik C++ može učiniti vaše modele **mašinskog učenja** (**ML**) bržim i efikasnijim**.** Ova knjiga vas uvodi u osnove mašinskog učenja i pokazuje kako da koristite C++ biblioteke za implementaciju modela mašinskog učenja. Objašnjava proces kreiranja modela nadgledanog i nenadgledanog mašinskog učenja.

Kroz praktične primere naučićete da podešavate i optimizujete modele za različite primene, uz smernice za odabir modela i merenje njegovih performansi. Knjiga pokriva tehnike kao što su preporučivanje proizvoda, ansambl učenje, detekcija anomalija, analiza osećanja i prepoznavanje objekata pomoću savremenih C++ biblioteka. Takođe, obrađuje izazove vezane za proizvodno okruženje i implementaciju modela na mobilnim platformama, kao i način na koji ONNX format modela može olakšati ove zadatke.

Najvažnije teme su ažurirane u ovom novom izdanju, uključujući implementaciju analize osećanja pomoću učenja prenosom znanja i modela zasnovanih na transformatorima, kao i praćenje i vizuelizaciju eksperimenata u oblasti mašinskog učenja pomoću alata MLflow. Dodat je i poseban deo o upotrebi biblioteke Optuna za izbor hiperparametara. Deo o implementaciji modela na mobilnim platformama proširen je detaljnim objašnjenjem detekcije objekata u realnom vremenu, na Android sistemu sa programskim jezikom C++.

Kada pročitate ovu knjigu imaćete praktično znanje iz oblasti mašinskog učenja i C++ programiranja, kao i veštine potrebne za razvoj moćnih sistema mašinskog učenja u programskom jeziku C++.

Kome je ova knjiga namenjena?

Ako želite da počnete da radite sa algoritmima i tehnikama mašinskog učenja pomoću popularnog programskog jezika C++, ova knjiga je pravi izbor za vas. Pored toga što služi kao uvodni kurs mašinskog učenja u programskom jeziku C++, knjiga će biti korisna i analitičarima podataka, naučnicima koji rade sa podacima i inženjerima mašinskog učenja koji žele da implementiraju različite modele mašinskog učenja u proizvodnji pomoću programskog jezika C++. To je posebno korisno u specifičnim

okruženjima, kao što su ugrađeni sistemi. Za rad sa ovom knjigom potrebno je predznanje iz C++ programskog jezika, linearne algebre i osnovno poznavanje više matematike.

Teme koje ova knjiga obuhvata

Poglavlje 1: Uvod u mašinsko učenje sa programskim jezikom C++ vas vodi kroz osnovne koncepte mašinskog učenja, uključujući pojmove iz linearne algebre, vrste algoritama mašinskog učenja i njihove osnovne komponente.

*Poglavlje 2: Obrada podataka o*bjašnjava kako se učitavaju podaci iz različitih formata datoteka za obučavanje modelamašinskog učenja, kao i kako se inicijalizuju objekti skupova podataka u raznim C++ bibliotekama.

Poglavlje 3: Merenje performansi i izbor modela prikazuje kako se mere performanse različitih vrsta modela mašinskog učenja, kako odabrati optimalne hiperparametre za postizanje boljih rezultata i kako koristiti metod pretraživanja unutar mreže u raznim C++ i eksternim bibliotekama za izbor modela.

Poglavlje 4: Klasterovanje obrađuje algoritme za grupisanje objekata na osnovu njihovih ključnih karakteristika. Objašnjava zašto se za ove zadatke uglavnom koriste nenadgledani algoritmi i daje pregled različitih tehnika klasterovanja, uz njihove implementacije i primenu u C++ bibliotekama.

Poglavlje 5: Detekcija anomalija razmatra osnove detekcije anomalija i noviteta, pruža uvod u različite algoritme za detekciju anomalija i njihovu implementaciju u različitim C++ bibliotekama.

Poglavlje 6: Smanjenje dimenzionalnosti objašnjava različite algoritme za smanjenje dimenzionalnosti podataka uz očuvanje njihovih ključnih karakteristika, kao i njihovu implementaciju i primenu u različitim C++ bibliotekama.

Poglavlje 7: Klasifikacija pokazuje šta je zadatak klasifikacije i po čemu se razlikuje od klasterovanja. Predstavlja vam razne klasifikacione algoritme, njihovu implementaciju i upotrebu u raznim C++ bibliotekama.

Poglavlje 8: Sistemi preporuke daje pregled osnovnih koncepata sistema preporuke. Objašnjava različite pristupe rešavanju ovih problema i pokazuje kako se implementiraju u jeziku C++.

Poglavlje 9: Ansambl učenje obrađuje metode kombinovanja više modela mašinskog učenja radi poboljšanja tačnosti i rešavanja problema učenja. Sadrži primere implementacije ansambl učenja uz upotrebu različitih C++ biblioteka.

Poglavlje 10: Neuronske mreže za klasifikaciju slika vam predstavlja osnove veštačkih neuronskih mreža, uključujući njihove osnovne komponente, potrebne matematičke koncepte i algoritme učenja. Pruža uvod u razne C++ biblioteke za implementaciju neuronskih mreža i demonstrira implementaciju duboke konvolucione neuronske mreže za klasifikaciju slika pomoću biblioteke PyTorch.

Poglavlje 11: Model BERT i učenje prenosom znanja za analizu osećanja upoznaje vas sa **velikim jezičkim modelima** (**LLM**) i ukratko objašnjava njihov način rada. Takođe,

pokazuje kako se koristi tehnika učenja prenosom znanja za implementaciju analize osećanja pomoću unapred obučenih velikih jezičkih modela i biblioteke PyTorch.

Poglavlje 12: Izvoz i uvoz modela objašnjava kako sačuvati i učitati parametre i arhitekture modela mašinskog učenja pomoću raznih C++ biblioteka. Takođe, pokazuje kako se koristi ONNX format za učitavanje i upotrebu unapred obučenog modela pomoću C++ API interfejsa biblioteke Caffe2.

Poglavlje 13: Praćenje i vizuelizacija eksperimenata mašinskog učenja pokazuje kako se alat MLflow koristi za praćenje i vizuelizaciju eksperimenata u oblasti mašinskog učenja. Vizuelizacija je ključna za razumevanje obrazaca, odnosa i trendova u eksperimentima, dok praćenje eksperimenata omogućava poređenje rezultata, identifikaciju najboljih pristupa i izbegavanje ponavljanja grešaka.

Poglavlje 14: Implementacija modela na mobilnoj platformi vodi vas kroz proces razvoja aplikacija za detekciju objekata na slikama sa kamere mobilnog uređaja, pomoću neuronske mreže na Android platformi.

Kako je najbolje koristiti ovu knjigu

Da biste mogli da kompajlirate i pokrenete primere iz ove knjige, potrebno je da konfigurišete odgovarajuće razvojno okruženje. Svi primeri koda testirani su na Ubuntu Linux 22.04 distribuciji. Sledeća lista prikazuje pakete koje je potrebno instalirati na Ubuntu platformi:

- `unzip`
- `build-essential`
- `gdb`
- `git`
- `libfmt-dev`
- `wget`
- `cmake`
- `python3`
- `python3-pip`
- `python-is-python3`
- `libblas-dev`
- `libopenblas-dev`
- `libfftw3-dev`
- `libatlas-base-dev`
- `liblapacke-dev`
- `liblapack-dev`
- `libboost-all-dev`

- libopencv-core4.5d
- libopencv-imgproc4.5d
- libopencv-dev
- libopencv-highgui4.5d
- libopencv-highgui-dev
- libhdf5-dev
- libjson-c-dev
- libx11-dev
- openjdk-8-jdk
- openjdk-17-jdk
- ninja-build
- gnuplot
- vim
- python3-venv
- libcpuinfo-dev
- libspdlog-dev

Biće vam potreban paket `cmake` sa verzijom koja nije manja od 2.27. Da biste ga imali na Ubuntu 22.04 platformi, morate ga preuzeti ručno i instalirati. Na primer, to se može uraditi na sledeći način:

```
wget https://github.com/Kitware/CMake/releases/download/v3.27.5/cmake-
3.27.5-Linux-x86_64.sh \
      -q -O /tmp/cmake-install.sh \
      && chmod u+x /tmp/cmake-install.sh \
      && mkdir /usr/bin/cmake \
      && /tmp/cmake-install.sh --skip-license --prefix=/usr/bin/cmake \
      && rm /tmp/cmake-install.sh

export PATH="/usr/bin/cmake/bin:${PATH}"
```

Takođe, potrebno je da instalirate dodatne pakete za jezik Python, što se može uraditi sledećim komandama:

```
pip install pyyaml
pip install typing
pip install typing_extensions
pip install optuna
pip install torch==2.3.1 \
  --index-url https://download.pytorch.org/whl/cpu
pip install transformers
pip install mlflow==2.15.0
```

Pored razvojnog okruženja, moraćete da proverite izvorne kodove neophodnih biblioteka trećih strana i da ih izgradite. Većina ovih biblioteka se aktivno razvija, te ćete

morati da obezbedite određene verzije (Git oznake) koje treba da proverite kako bi bile kompatibilne sa našim primerima koda. Sledeća tabela prikazuje biblioteke koje morate da proverite, njihove URL adrese spremišta, kao i oznake i heš brojeve potvrda.

SPREMIŠTE BIBLIOTEKE	IME GRANE/ OZNAKA	POTVRDA
`https://bitbucket.org/blaze-lib/blaze.git`	v3.8.2	
`https://github.com/arrayfire/arrayfire`	v3.8.3	
`https://github.com/flashlight/flashlight.git`	v0.4.0	
`https://github.com/davisking/dlib`	v19.24.6	
`https://gitlab.com/conradsnicta/armadillo-code`	14.0.x	
`https://github.com/xtensor-stack/xtl`	0.7.7	
`https://github.com/xtensor-stack/xtensor`	0.25.0	
`https://github.com/xtensor-stack/xtensor-blas`	0.21.0	
`https://github.com/nlohmann/json.git`	v3.11.3	
`https://github.com/mlpack/mlpack`	4.5.0	
`https://gitlab.com/libeigen/eigen.git`	3.4.0	
`https://github.com/BlueBrain/HighFive`	v2.10.0	
`https://github.com/yhirose/cpp-httplib`	v0.18.1	
`https://github.com/Kolkir/plotcpp`		c86bd4f5d9029986f0d5f368450d79f0dd32c7e4
`https://github.com/ben-strasser/fast-cpp-csv- parser`		4ade42d5f8c454c6c57b3dce9c51c6dd02182a66
`https://github.com/lisitsyn/tapkee`		Ba5f052d2548ec03dcc6a4ac0ed8deeb79f1d43a
`https://github.com/Microsoft/onnxruntime.git`	v1.19.2	

SPREMIŠTE BIBLIOTEKE	IME GRANE/ OZNAKA	POTVRDA
https://github.com/pytorch/pytorch	v2.3.1	

Imajte na umu da je najbolje kompajlirati i instalirati PyTorch na kraju, zbog mogućih konflikata sa verzijom `protobuf` biblioteke koju koristi `onnxruntime`.

Takođe, za poslednje poglavlje, možda ćete želeti da instalirate Android Studio IDE. Možete ga preuzeti sa zvanične stranice: `https://developer.android.com/studio`. Pored IDE okruženja, biće potrebno da instalirate i podesite Android SDK, NDK i Android verziju OpenCV biblioteke. Potrebne su vam sledeće verzije alata:

NAZIV	VERZIJA
OpenCV	4.10.0
Android command-line tools for Linux	9477386
Android NDK	26.1.10909125
Android platform	35

Možete konfigurisati ove alate pomoću Android IDE okruženja ili alata komandne linije, na sledeći način:

```
wget https://github.com/opencv/opencv/releases/download/4.10.0/opencv-
4.10.0-android-sdk.zip
unzip opencv-4.10.0-android-sdk.zip

wget https://dl.google.com/android/repository/commandlinetools-
linux-9477386_latest.zip
unzip commandlinetools-linux-9477386_latest.zip

./cmdline-tools/bin/sdkmanager --sdk_root=$ANDROID_SDK_ROOT „cmdline-
tools;latest”
./cmdline-tools/latest/bin/sdkmanager --licenses
./cmdline-tools/latest/bin/sdkmanager „platform-tools” „tools”
./cmdline-tools/latest/bin/sdkmanager „platforms;android-35”
./cmdline-tools/latest/bin/sdkmanager „build-tools;35.0.0”
./cmdline-tools/latest/bin/sdkmanager „system-images;android-
35;google_apis;arm64-v8a”
./cmdline-tools/latest/bin/sdkmanager --install „ndk;26.1.10909125”
```

Drugi način za konfigurisanje razvojnog okruženja je upotreba Docker platforme. Docker vam omogućava da podesite laganu virtuelnu mašinu sa određenim komponentama. Možete instalirati Docker iz zvaničnog Ubuntu spremišta paketa. Nakon toga, koristite skriptove koji dolaze uz ovu knjigu za automatsku konfiguraciju okruženja. U spremištu sa primerima naći ćete direktorijum `build-env`. Sledeći koraci pokazuju kako se koriste Docker konfiguracioni skriptovi:

1. Prvo konfigurišite svoj GitHub nalog. Zatim ćete moći da konfigurišete GitHub autentifikaciju putem SSH mrežnog protokola, kao što je opisano u članku *Connecting to GitHub with SSH* (`https://docs.github.com/en/authentication/connecting-to-github-with-ssh`); ovo je preporučeni način. Takođe, možete koristiti HTTPS i unositi korisničko ime i lozinku svaki put kada klonirate novo spremište. Ako koristite dvofaktorsku autentifikaciju za zaštitu svog GitHub naloga, umesto lozinke moraćete da koristite lični pristupni token, kao što je objašnjeno u članku *Creating a personal access token* (`https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token`).
2. Pokrenite sledeće komande da biste kreirali Docker sliku, pokrenuli je i konfigurisali okruženje:

   ```
   cd docker
   docker build -t buildenv:1.0 .
   ```

3. Koristite sledeću komandu da biste pokrenuli novi Docker kontejner i sa njim podelili izvorni kod primera iz knjige:

   ```
   docker run -it -v [host_examples_path]:[container_examples_path]
   [tag name] bash
   ```

Ovde, `host_examples_path` predstavlja putanju do preuzetih primera iz spremišta `https://github.com/PacktPublishing/Hands-on-Machine-learning-with-C-Second-Edition.git`, dok `container_examples_path` predstavlja putanju unutar kontejnera gde će se primeri priključiti, na primer `/samples`.

Nakon pokretanja prethodne komande, naći ćete se u okruženju komandne linije sa svim potrebnim konfigurisanim paketima, kompajliranim bibliotekama trećih strana i pristupom paketu programskih primera. Ovo okruženje možete koristiti za kompajliranje i pokretanje primera kodova iz ove knjige. Svaki primer je podešen da koristi CMake sistem za izgradnju, tako da ih sve možete kompajlirati na isti način. Sledeći skript prikazuje mogući scenario kompajliranja primera koda:

```
cd Chapter01
mkdir build
cd build
cmake ..
cmake --build . --target all
```

Ovo je ručni pristup. Takođe, obezbedili smo unapred pripremljene skriptove za izgradnju svakog primera. Ovi skriptovi se nalaze u direktorijumu `build_scripts`, u

spremištu. Na primer, skript za kompajliranje primera iz prvog poglavlja ima naziv `build_ch1.sh` i može se direktno pokrenuti iz tog direktorijuma.

Ako ćete svoje razvojno okruženje konfigurisati ručno, vodite računa o promenljivoj `LIBS_DIR`, koja treba da pokazuje na direktorijum gde su instalirane sve biblioteke trećih strana. Kada koristite unapred pripremljene skriptove za Docker okruženje, ona će automatski biti postavljena na `$HOME/development/libs`.

Takođe, možete podesiti svoje lokalno okruženje tako da deli X Server sa Docker kontejnerom, što omogućava pokretanje aplikacija koje koriste grafički korisnički interfejs iz kontejnera. To će vam omogućiti da koristite, na primer, Android Studio IDE ili C++ razvojno okruženje kao što je Qt Creator, direktno iz Docker kontejnera, bez potrebe za lokalnom instalacijom. Sledeći skript pokazuje kako se to postiže:

```
xhost +local:root
docker run --net=host -e DISPLAY=$DISPLAY -v /tmp/.X11-unix:/tmp/.X11-
unix -it -v [host_examples_path]:[container_examples_path] [tag name]
bash
```

Da biste bolje razumeli i efikasno koristili primere kodova, preporučujemo da pažljivo proučite dokumentaciju za svaku biblioteku treće strane i posvetite neko vreme učenju osnova Docker sistema i razvoja za Android platformu. Takođe, pretpostavljamo da imate dovoljno radnog znanja o C++ jeziku i kompajlerima, kao i da ste upoznati sa CMake sistemom za izgradnju.

Ako koristite digitalnu verziju ove knjige, savetujemo vam da sami otkucate kod ili mu pristupite putem GitHub spremišta (adresa se nalazi u sledećem odeljku). Na taj način ćete izbeći moguće greške koje mogu nastati prilikom kopiranja i lepljenja koda.

Preuzimanje datoteka sa primerima koda

Možete preuzeti datoteke sa primerima koda za ovu knjigu sa GitHub spremišta na adresi `https://github.com/PacktPublishing/Hands-on-Machine-learning-with-C-Second-Edition`. Ako dođe do ažuriranja koda, ono će biti ažurirano u GitHub spremištu.

Takođe, imamo i druge pakete koda iz našeg bogatog kataloga knjiga i video sadržaja dostupne na `https://github.com/PacktPublishing/`. Pogledajte ih!

Primenjene konvencije

U ovoj knjizi koristimo razne tekstualne konvencije.

`Kod u tekstu`: označava kod u okviru teksta, nazive tabela u bazama podataka, nazive direktorijuma, imena datoteka, ekstenzije datoteka, putanje, lažne URL adrese, korisnički unos i Twitter/X naloge. Na primer: „Biblioteka `Dlib` ne sadrži mnogo klasifikacionih algoritama”.

Blok koda je prikazan na sledeći način:

```
std::vector<fl::Tensor> fields{train_x, train_y};
auto dataset = std::make_shared<fl::TensorDataset>(fields);
int batch_size = 8;
auto batch_dataset = std::make_shared<fl::BatchDataset>(dataset,
                                                        batch_size);
```

Kada želimo da skrenemo pažnju na određeni deo bloka koda, relevantni redovi ili stavke su podebljani:

```
[default]
exten => s,1,Dial(Zap/1|30)
exten => s,2,Voicemail(u100)
exten => s,102,Voicemail(b100)
exten => i,1,Voicemail(s0)
```

Bilo koji unos ili izlaz iz komandne linije prikazuje se ovako:

```
$ mkdir css
$ cd css
```

Podebljan tekst: označava novi termin, važnu reč ili pojmove koji se pojavljuju na ekranu. Na primer, opcije u menijima ili dijalog prozorima prikazane su **podebljano**. Na primer: „U strategiji **jedan-protiv-svih** za N klasa, obučavaju se N klasifikatora, od kojih svaki razdvaja svoju klasu od svih ostalih.”

> **Saveti** ili **važne napomene**
>
> Prikazujemo na sledeći način

DEO 1

Pregled mašinskog učenja

U ovom delu ćemo se posvetiti osnovama mašinskog učenja uz pomoć primera u jeziku C++ i raznih okvira mašinskog učenja. Pokazaćemo kako se učitavaju podaci iz različitih formata datoteka, opisaćemo tehnike merenja performansi modela i najbolje pristupe za izbor modela.

Ovaj deo obuhvata sledeća poglavlja:

- *Poglavlje 1, Uvod u mašinsko učenje sa programskim jezikom C++*
- *Poglavlje 2, Obrada podataka*
- *Poglavlje 3, Merenje performansi i izbor modela*

1

Uvod u mašinsko učenje sa programskim jezikom C++

Postoje različiti pristupi rešavanju zadataka pomoću računara. Jedan od njih je definisanje eksplicitnog algoritma, dok je drugi korišćenje implicitnih strategija zasnovanih na matematičkim i statističkim metodima. **Mašinsko učenje** (**ML**) jedan je od implicitnih metoda koji koristi matematičke i statističke pristupe za rešavanje zadataka. To je oblast koja se aktivno razvija, a mnogi naučnici i istraživači smatraju je jednim od najboljih puteva ka sistemima koji funkcionišu kao **veštačka inteligencija** (**AI**) na ljudskom nivou.

Uopšteno, pristupi mašinskog učenja se zasnivaju na ideji pretraživanja obrazaca u datom skupu podataka. Razmotrimo sistem preporuke za prikaz vesti, koji korisniku pruža personalizovani sadržaj na osnovu njegove prethodne aktivnosti ili interesovanja. Softver prikuplja podatke o vrstama članaka koje korisnik čita i izračunava određene statistike. Na primer, to može biti učestanost pojavljivanja određenih tema u skupu vesti. Zatim, sprovodi predikcionu analizu, identifikuje opšte obrasce i koristi ih za kreiranje personalizovanog toka vesti korisnika. Takvi sistemi periodično prate aktivnost korisnika, ažuriraju bazu podataka i izračunavaju nove trendove za preporuke.

Mašinsko učenje je oblast koja se brzo razvija i nalazi široku primenu u različitim industrijama. U zdravstvu, analizira medicinske podatke radi otkrivanja obrazaca i predviđanja bolesti i ishoda lečenja. U finansijama, pomaže u proceni kreditne sposobnosti, detekciji prevara, analizi rizika, optimizaciji portfolija i algoritamskom trgovanju, poboljšavajući donošenje odluka i operacije. E-trgovina koristi sisteme preporuke koji predlažu proizvode na osnovu ponašanja kupaca, povećavajući prodaju i zadovoljstvo korisnika. Autonomna vozila koriste mašinsko učenje za percepciju okruženja, donošenje odluka i bezbednu navigaciju.

Korisnička podrška je poboljšana uz četbotove i virtuelne asistente koji obrađuju upite i zadatke. Sajber bezbednost koristi mašinsko učenje za detekciju i sprečavanje sajber

napada analizom mrežnog saobraćaja i prepoznavanjem pretnji. Alati za prevođenje jezika koriste mašinsko učenje za precizan i efikasan prevod teksta. Prepoznavanje slika, zasnovano na algoritmima računarskog vida, identifikuje objekte, lica i scene na slikama i video zapisima, podržavajući aplikacije poput prepoznavanja lica i moderacije sadržaja. Prepoznavanje govora u glasovnim asistentima kao što su Siri, Google Assistant i Alexa oslanja se na mašinsko učenje za razumevanje i odgovaranje na korisničke komande. Ovi primeri ilustruju ogroman potencijal mašinskog učenja u oblikovanju naših života.

Ovo poglavlje objašnjava šta je mašinsko učenje, koje zadatke može rešavati i razmatra različite pristupe koji se koriste u mašinskom učenju. Cilj je da se predstave matematičke osnove neophodne za početak primene algoritama mašinskog učenja. Takođe, objašnjavaju se osnovne operacije **linearne algebre** u bibliotekama kao što su `Eigen`, `xtensor`, `ArrayFire`, `Blaze` i `Dlib`, a takođe se objašnjava zadatak linearne regresije kao primer.

U ovom poglavlju su obrađene sledeće teme:

- Osnove mašinskog učenja
- Uvod u linearnu algebru
- Primer linearne regresije

Osnove mašinskog učenja

Postoje različiti pristupi za kreiranje i obučavanje modela mašinskog učenja. U ovom odeljku ćemo prikazati koji su to pristupi i po čemu se međusobno razlikuju. Pored pristupa koji koristimo za izradu modela mašinskog učenja, postoje i parametri koji određuju njegovo ponašanje tokom procesa obučavanja i ocenjivanja. Parametri modela mogu se podeliti u dve grupe, pri čemu se svaka konfiguriše na drugačiji način. Prva grupa su težine modela, koje algoritmi mašinskog učenja koriste za prilagođavanje predikcija modela. Ove numeričke vrednosti se dodeljuju tokom procesa obučavanja i određuju način na koji model donosi odluke ili predviđa rezultate na novim podacima. Druga grupa su hiperparametri modela, koji kontrolišu ponašanje modela mašinskog učenja tokom obučavanja. Za razliku od težina modela, hiperparametri nisu naučeni iz podataka, nego ih korisnik, ili algoritam, podešava pre početka obučavanja. Poslednji ključni deo procesa mašinskog učenja je tehnika koju koristimo za obučavanje modela. Obično se tehnika obučavanja oslanja na neki numerički optimizacioni algoritam koji pronalazi minimalnu vrednost ciljne funkcije. U mašinskom učenju, ciljnu funkciju obično nazivamo funkcija gubitka i služi za kažnjavanje algoritma obučavanja kada napravi grešku. Ove koncepte ćemo preciznije obraditi u narednim odeljcima.

Uvod u tehnike mašinskog učenja

Pristupi u mašinskom učenju mogu se podeliti na dve glavne tehnike:

- **Nadgledano učenje** je pristup koji se zasniva na upotrebi označenih podataka. Označeni podaci su skup poznatih uzoraka sa odgovarajućim poznatim ciljnim izlazima. Ovi podaci služe za izgradnju modela koji može da predviđa buduće izlazne vrednosti.

- **Nenadgledano učenje** je pristup koji ne zahteva označene podatke i može da pronalazi skrivene obrasce i strukture u bilo kojoj vrsti podataka.

Hajde da detaljnije razmotrimo ove tehnike.

Nadgledano učenje

Nadgledani algoritmi mašinskog učenja obično uzimaju ograničen skup označenih podataka i grade modele koji mogu davati razumne predikcije za nove podatke. Algoritme nadgledanog učenja možemo podeliti u dve glavne kategorije, klasifikacione i regresione tehnike, koje su opisane na sledeći način:

- Klasifikacioni modeli predviđaju konačne i jasno definisane kategorije—na primer, oznaku koja identifikuje da li je elektronska poruka spam ili ne, ili da li slika sadrži ljudsko lice. Klasifikacioni modeli se primenjuju za prepoznavanje govora i teksta, identifikovanje objekata na slikama, procenu kreditne sposobnosti, kao i u drugim oblastima. Tipični algoritmi za kreiranje klasifikacionih modela su **mašine sa potpornim vektorima** (**SVM**), metodi stabla odlučivanja, metod **k-najbližih suseda** (**KNN**), logistička regresija, naivni Bajesov metod i neuronske mreže. Sledeća poglavlja opisuju detalje nekih od ovih algoritama.
- Regresioni modeli predviđaju vrednosti neprekidnih promenljivih, kao što su promene temperature ili vrednosti deviznih kurseva. Regresioni modeli se koriste za algoritamsko trgovanje, predviđanje opterećenja električne mreže, prognoziranje prihoda, kao i u drugim oblastima. Kreiranje regresionog modela, obično, ima smisla ako su izlazne vrednosti označenih podataka realni brojevi. Tipični algoritmi za kreiranje regresionih modela su linearna i višestruka regresija, polinomijalni regresioni modeli i postepena regresija. Takođe, tehnike stabla odlučivanja i neuronske mreže mogu se koristiti za kreiranje regresionih modela.

Sledeća poglavlja opisuju detalje nekih od ovih algoritama.

Nenadgledano učenje

Algoritmi nenadgledanog učenja ne koriste označene skupove podataka. Oni kreiraju modele koji koriste unutrašnje odnose u podacima kako bi pronašli skrivene obrasce koje mogu koristiti za predikciju. Najpoznatija tehnika nenadgledanog učenja je **klasterovanje**. Klasterovanje podrazumeva podelu datog skupa podataka na ograničen broj grupa, prema određenim unutrašnjim svojstvima podataka. Klasterovanje se koristi za istraživanje tržišta, različite vrste istraživačkih analiza, analizu **dezoksiribonukleinske kiseline** (**DNK**), segmentaciju slika i detekciju objekata. Tipični algoritmi za kreiranje modela za klasterovanje su *k-sredine*, *k-medoidi*, modeli mešavina Gausovih raspodela, hijerarhijsko klasterovanje i skriveni modeli Markova. Neki od ovih algoritama objašnjeni su u narednim poglavljima ove knjige.

Rad sa modelima mašinskog učenja

Modele mašinskog učenja možemo tumačiti kao funkcije koje primaju različite tipove parametara. Takve funkcije daju izlazne vrednosti za zadate ulaze na osnovu vrednosti tih parametara. Programeri mogu konfigurisati ponašanje modela mašinskog učenja za rešavanje problema podešavanjem parametara modela. Obučavanje modela mašinskog

učenja se obično može posmatrati kao proces traženja najbolje kombinacije njegovih parametara. Parametre modela mašinskog učenja možemo podeliti u dve kategorije. Prva kategorija obuhvata parametre koji su interni za model i čije vrednosti možemo oceniti na osnovu podataka za obuku (ulaznih podataka). Druga kategorija obuhvata parametre koji su eksterni za model i čije vrednosti ne možemo oceniti na osnovu podataka za obuku. Parametre koji su eksterni za model obično nazivamo **hiperparametri**.

Interni parametri imaju sledeće karakteristike:

- Neophodni su za donošenje predikcija
- Definišu kvalitet modela za dati problem
- Mogu se naučiti iz podataka za obuku
- Obično su deo modela

Ako model sadrži fiksiran broj internih parametara, nazivamo ga **parametarski**. U suprotnom, može se klasifikovati kao **neparametarski**.

Primeri internih parametara su sledeći:

- Težine **veštačkih neuronskih mreža** (**ANN**)
- Vrednosti potpornih vektora za SVM modele
- Koeficijenti polinoma za linearnu regresiju ili logističku regresiju

Veštačke neuronske mreže su računarski sistemi inspirisani strukturom i funkcijom bioloških neuronskih mreža ljudskog mozga. One se sastoje od međusobno povezanih čvorova, odnosno neurona, koji obrađuju i prenose informacije. ANN modeli su dizajnirani da uče obrasce i odnose iz podataka, što im omogućava da prave predikcije ili donose odluke na osnovu novih ulaza. Proces učenja obuhvata prilagođavanje težina i pristrasnosti veza između neurona, kako bi se poboljšala tačnost modela.

S druge strane, hiperparametri imaju sledeće karakteristike:

- Koriste se za konfigurisanje algoritama koji ocenjuju parametre modela
- Obično ih određuje korisnik modela
- Njihova ocena se često zasniva na heurističkim metodima
- Specifični su za konkretan problem modeliranja

Teško je unapred znati najbolje vrednosti hiperparametara za određeni problem. Takođe, korisnici modela obično moraju sprovesti dodatna istraživanja o tome kako podesiti potrebne hiperparametre kako bi model ili algoritam obuke radio na najbolji način. U praksi se koriste pravila zasnovana na iskustvu, preuzimanje vrednosti iz sličnih projekata, kao i specijalizovane tehnike, poput pretrage po mreži za ocenjivanje hiperparametara.

Primeri hiperparametara su:

- Parametri C i sigma upotrebljeni u SVM algoritmu za podešavanje kvaliteta klasifikacije
- Parametar brzine učenja, koji se koristi u procesu obuke neuronskih mreža za podešavanje konvergencije algoritma

- Vrednost *k* koja se koristi u KNN algoritmu za određivanje broja suseda

Ocenjivanje parametara modela

Ocenjivanje parametara modela obično koristi neki optimizacioni algoritam. Brzina i kvalitet rezultujućeg modela značajno zavise od izabranog optimizacionog algoritma. Istraživanje optimizacionih algoritama je popularna tema u industriji i u akademskoj zajednici. Mašinsko učenje često koristi tehnike i algoritme optimizacije zasnovane na optimizaciji funkcije gubitka. Funkciju koja ocenjuje koliko dobro model predviđa podatke nazivamo **funkcija gubitka**. Ako su predikcije veoma različite od ciljanih izlaza, funkcija gubitka će vratiti vrednost koja se može protumačiti kao loša, što je obično veliki broj. Na taj način funkcija gubitka kažnjava optimizacioni algoritam kada se kreće u pogrešnom pravcu. Dakle, opšta ideja je minimizacija vrednosti funkcije gubitka, kako bi se smanjile kazne. Ne postoji univerzalna funkcija gubitka za optimizacione algoritme. Različiti faktori određuju izbor funkcije gubitka. Primeri takvih faktora su:

- Specifičnosti datog problema—na primer, da li je u pitanju regresioni ili klasifikacioni model
- Jednostavnost računanja izvoda
- Procenat netipičnih vrednosti u skupu podataka

U mašinskom učenju, termin **optimizator** se koristi za definisanje algoritma koji povezuje funkciju gubitka i tehniku ažuriranja parametara modela, u zavisnosti od vrednosti funkcije gubitka. Dakle, optimizatori prilagođavaju modele mašinskog učenja kako bi što tačnije predviđali ciljne vrednosti za nove podatke, podešavanjem parametara modela. Optimizatori, odnosno optimizacioni algoritmi, igraju ključnu ulogu u obučavanju modela mašinskog učenja. Oni pomažu u pronalaženju najboljih parametara za model, što može poboljšati njegovu tačnost i performanse. Imaju široku primenu u različitim oblastima, kao što su prepoznavanje slika, obrada prirodnog jezika i detekcija prevara. Na primer, u zadacima klasifikacije slika, optimizacijski algoritmi se mogu koristiti za obučavanje dubokih neuronskih mreža, kako bi tačno identifikovale objekte na slikama. Postoji mnogo optimizatora: gradijentni spust, Adagrad, RMSProp, Adam i drugi. Pored toga, razvoj novih optimizatora je aktivno područje istraživanja. Na primer, kompanija Microsoft u Redmondu ima istraživačku grupu za mašinsko učenje i optimizaciju, čije oblasti istraživanja obuhvataju kombinatornu optimizaciju, konveksnu i nekonveksnu optimizaciju i njihovu primenu u mašinskom učenju i veštačkoj inteligenciji. Druge kompanije u industriji, takođe, imaju slične istraživačke grupe; objavljeni su brojni naučni radovi istraživačkih timova kompanija kao što su Facebook Research, Amazon Research i OpenAI.

Sada ćemo naučiti šta je mašinsko učenje i koji su njegovi glavni konceptualni delovi. Dakle, preći ćemo na najvažniji deo njegovih matematičkih osnova: linearnu algebru.

Uvod u linearnu algebru

Koncepti linearne algebre su ključ za razumevanje teorije mašinskog učenja, jer nam pomažu da shvatimo kako algoritmi mašinskog učenja funkcionišu u pozadini. Takođe, većina definicija algoritama mašinskog učenja koristi pojmove linearne algebre.

Linearna algebra nije samo korisna matematička oblast, već se njeni koncepti mogu veoma efikasno primeniti na savremene arhitekture računara. Uspon mašinskog učenja, a posebno dubokog učenja, počeo je nakon značajnog poboljšanja performansi modernih **grafičkih procesorskih jedinica (GPU)**. One su prvobitno dizajnirane za rad sa konceptima linearne algebre i masovna paralelna izračunavanja koja se koriste za računarske igre. Nakon toga, razvijene su posebne biblioteke za rad sa opštim konceptima linearne algebre. Primeri biblioteka koje implementiraju osnovne rutine linearne algebre su `Cuda` i `OpenCL`, dok je primer specijalizovane biblioteke linearne algebre `cuBLAS`. Pored toga, ustalila se upotreba **grafičkih procesorskih jedinica opšte namene (GPGPU)**, jer oni pretvaraju računske mogućnosti savremenog procesora u moćan resurs za opštu namenu.

Takođe, **centralna procesorska jedinica (CPU)** poseduje skupove instrukcija posebno dizajniranih za istovremena numerička izračunavanja. Ta izračunavanja nazivamo **vektorizovana**, a uobičajeni skupovi vektorizovanih instrukcija su `AVX`, `SSE` i `MMX`. Ovi skupovi instrukcija su poznati i pod terminom **jedna instrukcija, više podataka (SIMD)**. Mnoge numeričke biblioteke linearne algebre, kao što su `Eigen`, `xtensor`, `ViennaCL` i druge, koriste ove instrukcije radi poboljšanja računskih performansi.

Koncepti linearne algebre

Linearna algebra je široko područje. To je oblast algebre koja proučava objekte linearne prirode: vektorske (ili linearne) prostore, linearne reprezentacije i sisteme linearnih jednačina. Glavni alati koji se koriste u linearnoj algebri su determinante, matrice, konjugacija i tenzorski račun.

Da bismo razumeli algoritme mašinskog učenja, potrebno je da poznajemo samo mali skup koncepata linearne algebre. Međutim, za istraživanje novih algoritama mašinskog učenja, potrebno je temeljno razumevanje linearne algebre i više matematike.

Sledeća lista sadrži najvažnije koncepte linearne algebre koji su ključni za razumevanje algoritama mašinskog učenja:

- **Skalar:** Jedan broj.
- **Vektor:** Niz uređenih brojeva. Svaki element ima jedinstveni indeks. Notacija za vektore podrazumeva podebljana mala slova za nazive i kurzivna slova sa donjim indeksom za elemente, kao što je prikazano u sledećem primeru:

$$\vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{bmatrix}$$

- **Matrica:** Ovo je dvodimenzionalni niz brojeva. Svaki element ima jedinstven par indeksa. Notacija za matrice podrazumeva podebljana velika slova za nazive i kurzivna, ali ne podebljana, slova sa donjim indeksima odvojenim zarezom za elemente, kao što je prikazano u sledećem primeru:

$$\boldsymbol{A} = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \\ A_{3,1} & A_{3,2} \end{bmatrix}$$

- **Tenzor:** Ovo je niz brojeva raspoređen u višedimenzionalnu pravilnu mrežu i predstavlja generalizaciju matrica. Može se posmatrati kao višedimenzionalna matrica. Na primer, tenzor *A* dimenzija 2 × 2 × 2 može izgledati ovako:

$$\boldsymbol{A} = \begin{bmatrix} \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \\ \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} \end{bmatrix}$$

Biblioteke linearne algebre i okviri mašinskog učenja obično koriste koncept tenzora umesto matrica, jer implementiraju opšte algoritme, dok je matrica samo specijalan slučaj tenzora sa dve dimenzije. Takođe, vektor se može posmatrati kao matrica veličine *n* × *1*.

Osnovne operacije linearne algebre

Najčešće operacije koje se koriste za programiranje algoritama linearne algebre su:

- **Operacije po elementima:** ove operacije se izvode element po element na vektorima, matricama ili tenzorima iste veličine. Rezultujući elementi biće rezultat operacija nad odgovarajućim ulaznim elementima, kao što je prikazano ovde:

$$\boldsymbol{A} + \boldsymbol{B} = \boldsymbol{C}, C_{i,j} = A_{i,j} + B_{i,j}$$
$$\boldsymbol{A} - \boldsymbol{B} = \boldsymbol{C}, C_{i,j} = A_{i,j} - B_{i,j}$$
$$\boldsymbol{A} * \boldsymbol{B} = \boldsymbol{C}, C_{i,j} = A_{i,j} * B_{i,j}$$
$$\boldsymbol{A} / \boldsymbol{B} = \boldsymbol{C}, C_{i,j} = A_{i,j} / B_{i,j}$$

Sledeći primer prikazuje sabiranje po elementima:

$$\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix}+\begin{bmatrix}5 & 6\\7 & 8\end{bmatrix}=\begin{bmatrix}1+5 & 2+6\\3+7 & 4+8\end{bmatrix}$$

- **Skalarni proizvod:** u linearnoj algebri postoje dve vrste množenja za tenzore i matrice—jedna je množenje po elementima, a druga je skalarni proizvod. Skalarni proizvod uzima dva niza brojeva iste dužine i vraća jedan broj. Ova operacija, kada se primeni na matrice ili tenzore, zahteva da matrica ili tenzor *A* ima isti broj kolona kao što matrica ili tenzor *B* ima vrsta. Sledeći primer prikazuje skalarni proizvod kada je *A* matrica dimenzija $n \times m$, a *B* matrica dimenzija $m \times p$:

$$\boldsymbol{A}\cdot\boldsymbol{B}=\boldsymbol{C}, C_{i,j}=\sum_{k=1}^{m}A_{i,k}B_{k,j}, i=1,\ldots,n, j=1,\ldots,p$$

- **Transponovanje:** transponovanje matrice je operacija koja obrće matricu oko njene dijagonale, što dovodi do zamene vrsta i kolona, čime se stvara nova matrica. Uopšteno, ova operacija podrazumeva zamenu vrsta sa kolonama. Sledeći primer prikazuje kako transponovanje funkcioniše:

$$(\boldsymbol{A})^{T}{}_{i,j}=\boldsymbol{A}_{j,i}\begin{bmatrix}1 & 3 & 5\\2 & 4 & 6\end{bmatrix}^{T}=\begin{bmatrix}1 & 2\\3 & 4\\5 & 6\end{bmatrix}$$

- **Norma:** ova operacija izračunava veličinu vektora, a rezultat je nenegativan realan broj. Formula norme je sledeća:

$$\|\boldsymbol{x}\|_{p}=\left(\sum_{i}|\boldsymbol{x}_{i}|^{p}\right)^{\frac{1}{p}}$$

Opšti naziv ove vrste norme je L^p norma za $p \in R, p \geq 1$. Obično koristimo konkretne norme, kao što je L^2 norma sa $p = 2$, koja je poznata kao Euklidova norma, a može se tumačiti kao Euklidovo rastojanje između tačaka. Druga često korišćena norma je kvadratna L^2 norma, čija formula za izračunavanje je $x^T x$. Kvadratna L^2 norma je pogodnija za matematičke i računarske operacije od obične L^2 norme. Svaki parcijalni izvod kvadratne L^2 norme zavisi samo od odgovarajućeg elementa vektora *x*, dok parcijalni izvodi L^2 norme zavise od celog vektora; ova osobina igra ključnu ulogu u algoritmima optimizacije. Još jedna često korišćena norma je L^1 norma sa $p = 1$, koja se često koristi u mašinskom učenju, kada je važno razlikovati nulte i nenulte elemente. L^1 norma je poznata i kao **Menhetn rastojanje**.

- **Invertovanje:** inverzna matrica je matrica za koju važi $\mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$, gde je I jedinična matrica. Jedinična matrica je matrica koja ne menja nijedan vektor kada taj vektor pomnožimo njome.

Razmotrili smo glavne koncepte linearne algebre, kao i operacije nad njima. Pomoću ovog matematičkog aparata, možemo definisati i programirati mnoge algoritme mašinskog učenja. Na primer, tenzori i matrice mogu se koristiti za definisanje skupova podataka za obučavanje, dok se skalari mogu koristiti kao različiti tipovi koeficijenata. Operacije po elementima omogućavaju izvođenje aritmetičkih operacija nad celim skupom podataka (matricom ili tenzorom). Na primer, množenje po elementima može se koristiti za skaliranje skupa podataka. Obično koristimo transponovanje da bismo promenili prikaz vektora ili matrice i učinili ih pogodnim za skalarni proizvod. Skalarni proizvod se obično koristi za primenu linearne funkcije, gde su težine izražene kao koeficijenti matrice na vektor; na primer, taj vektor može biti uzorak za obučavanje. Takođe, skalarni proizvod se koristi za ažuriranje parametara modela, koji su predstavljeni kao koeficijenti matrice ili tenzora, u skladu sa određenim algoritmom.

Norma se često koristi u formulama za funkcije gubitka jer prirodno izražava koncept rastojanja i može meriti razliku između ciljnih i predviđenih vrednosti. Inverzna matrica je ključan koncept za analitičko rešavanje sistema linearnih jednačina. Takvi sistemi se često javljaju u različitim problemima optimizacije. Međutim, izračunavanje inverzne matrice je računski veoma zahtevno.

Predstavljanje tenzora u računarstvu

Tenzorski objekti u memoriji računara mogu biti predstavljeni na različite načine. Najjednostavniji metod je linearni niz u memoriji računara (**memorija sa slučajnim pristupom** ili **RAM**). Međutim, linearni niz je i najefikasnija računarska struktura podataka za moderne procesore. Postoje dva standardna pristupa za organizaciju tenzora kao linearnog niza u memoriji: **rasporedu po vrstama** i **raspored po kolonama**.

U **rasporedu po vrstama**, uzastopni elementi jedne vrste smeštaju se linearno, jedan za drugim, dok se svaka sledeća vrsta postavlja odmah nakon završetka prethodne. U **rasporedu po kolonama**, primenjuje se isti princip, ali na elemente kolona. Organizacija podataka u memoriji značajno utiče na računarske performanse, jer brzina prolaska kroz niz zavisi od arhitekture savremenih procesora, koji mnogo efikasnije obrađuju **sekvencijalne podatke** nego one koji nisu sekvencijalno raspoređeni. Ovakvo ponašanje proističe iz efekata **keširanja** u procesorima. Takođe, **neprekidni raspored podataka u memoriji** omogućava upotrebu **SIMD vektorizovanih instrukcija**, koje optimizuju rad sa sekvencijalnim podacima i mogu se koristiti kao vid paralelne obrade.

Različite biblioteke, čak i u okviru istog programskog jezika, mogu koristiti različite metode rasporeda podataka. Na primer, biblioteka `Eigen` koristi raspored po kolonama, dok biblioteka `PyTorch` koristi raspored po vrstama. Zato programeri moraju biti svesni unutrašnje reprezentacije tenzora u bibliotekama koje koriste i voditi računa o tome prilikom učitavanja podataka, ili implementacije algoritama od početka.

Posmatrajmo sledeću matricu:

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{bmatrix}$$

U rasporedu podataka po vrstama, elementi matrice će biti organizovani u memoriji na sledeći način:

0	1	2	3	4	5
a_{11}	a_{12}	a_{13}	a_{21}	a_{22}	a_{23}

Tabela 1.1 – Primer rasporeda podataka po vrstama

U rasporedu podataka po kolonama, elementi matrice su poređani ovako:

0	1	2	3	4	5
a_{11}	a_{21}	a_{12}	a_{21}	a_{13}	a_{23}

Tabela 1.2 – Primer rasporeda podataka po kolonama

Primeri upotrebe API interfejsa linearne algebre

Razmotrimo neke C++ **programske interfejse aplikacija** (**API**) za linearnu algebru i pogledajmo kako ih možemo koristiti za kreiranje osnovnih struktura linearne algebre i izvođenje algebarskih operacija.

Upotreba biblioteke Eigen

`Eigen` je C++ biblioteka za linearnu algebru opšte namene. U biblioteci Eigen, sve matrice i vektori predstavljeni su objektima klase `Matrix`, a vektori su specijalizovani oblici matrica sa jednom vrstom ili jednom kolonom. Tenzorski objekti nisu zvanično deo glavnog API interfejsa, ali postoje kao podmoduli.

Tip matrice sa poznatim dimenzijama 3 × 3 i podacima u formatu sa pokretnim zarezom možemo definisati na sledeći način:

```
typedef Eigen::Matrix<float, 3, 3> MyMatrix33f;
```

Vektor kolonu možemo definisati ovako:

```
typedef Eigen::Matrix<float, 3, 1> MyVector3f;
```

Biblioteka Eigen već sadrži mnogo unapred definisanih tipova za objekte vektora i matrica—na primer, `Eigen::Matrix3f` (tip matrice dimenzija 3 × 3 sa elementima u formatu sa pokretnim zarezom) ili `Eigen::RowVector2f` (tip vektora dimenzija 1 × 2 sa elementima u formatu floating-point). Takođe, biblioteka Eigen nije ograničena samo na matrice čije su dimenzije poznate u vreme kompajliranja. Možemo definisati tipove matrica kod kojih se broj vrsta i kolona određuje prilikom inicijalizacije tokom izvršavanja. Za definisanje tih tipova, možemo koristiti specijalni tip promenljive kao argument generičke klase `Matrix`, nazvan `Eigen::Dynamic`. Na primer, za definisanje matrice sa elementima tipa `double` i dinamičkim dimenzijama, koristimo sledeću definiciju:

```
typedef Eigen::
    Matrix<double, Eigen::Dynamic, Eigen::Dynamic>
        MyMatrix;
```

Objekti inicijalizovani iz definisanih tipova izgledaju ovako:

```
MyMatrix33f a;
MyVector3f v;
MyMatrix m(10,15);
```

Za dodelu vrednosti ovim objektima možemo koristiti više pristupa. Možemo koristiti posebne unapred definisane funkcije za inicijalizaciju, kao što je prikazano:

```
a = MyMatrix33f::Zero(); // popunjava sve elemente matrice nulama
a = MyMatrix33f::Identity(); // popunjava matricu kao jedinicnu matricu
v = MyVector3f::Random(); // popunjava elemente matrice slučajnim vrednostima
```

Možemo koristiti i *sintaksu inicijalizacije zarezom*, kao što je prikazano:

```
a << 1,2,3,
     4,5,6,
     7,8,9;
```

Ova konstrukcija koda inicijalizuje matricu sledećim vrednostima:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Možemo koristiti direktan pristup elementima da bismo postavili ili promenili vrednosti pojedinačnih koeficijenata matrice. Sledeći primer koda pokazuje kako se za tu operaciju koristi operator `()`:

```
a(0,0) = 3;
```

Možemo koristiti objekat tipa `Map` da bismo obuhvatili postojeći C++ niz ili vektor unutar objekta tipa `Matrix`. Ovaj tip mapiranja koristi memoriju i vrednosti iz postojećeg objekta, bez dodatnog alociranja memorije i kopiranja vrednosti. Sledeći kod pokazuje kako se koristi tip `Map`:

```
int data[] = {1,2,3,4};
Eigen::Map<Eigen::RowVectorXi> v(data,4);

std::vector<float> data = {1,2,3,4,5,6,7,8,9};
Eigen::Map<MyMatrix33f> a(data.data());
```

Možemo koristiti inicijalizovane objekte matrica u matematičkim operacijama. Aritmetičke operacije sa matricama i vektorima u biblioteci `Eigen` realizuju se ili kroz preklopljene C++ aritmetičke operatore poput `+`, `-` ili `*`, ili pomoću metoda kao što su `dot()` i `cross()`. Sledeći primer koda prikazuje kako se mogu izraziti opšte matematičke operacije u `Eigen` biblioteci:

```
using namespace Eigen;
auto a = Matrix2d::Random();
auto b = Matrix2d::Random();
auto result = a + b;
result = a.array() * b.array(); // množenje po elementima
result = a.array() / b.array();
a += b;
result = a * b; // matrično množenje

// Takođe je moguće koristiti skalare:
a = b.array() * 4;
```

Važno je napomenuti da u `Eigen` biblioteci aritmetički operatori, kao što je `+`, ne izvršavaju odmah računsku operaciju. Ovi operatori vraćaju *objekat izraza*, koji opisuje koju operaciju treba izvršiti. Računanja se odvijaju kasnije, najčešće kada se ceo izraz evaluira, obično u operatoru `=`. Ovo može dovesti do nepredviđenog ponašanja, naročito ako se ključna reč `auto` prečesto koristi.

Ponekad je potrebno izvršiti operacije samo na određenom delu matrice. Za tu svrhu, biblioteka `Eigen` nudi metod `block`, koji prima četiri parametra: `i, j, p, q`. Ovi parametri određuju početnu tačku `(i, j)` i veličinu bloka `(p, q)`. Sledeći primer koda prikazuje kako se koristi ovaj metod:

```
Eigen::MatrixXf m(4,4);
Eigen::Matrix2f b = m.block(1,1,2,2); // kopira središnji deo matrice
m.block(1,1,2,2) *= 4; // menja vrednosti u originalnoj matrici
```

Postoje još dva metoda za pristup vrstama i kolonama pomoću indeksa, što su takođe specijalni oblici `block` operacija. Sledeći primer koda prikazuje kako se koriste metodi `col` i `row`:

```
m.row(1).array() += 3;
m.col(2).array() /= 4;
```

Još jedna važna osobina biblioteka za linearnu algebru je proširivanje, a biblioteka Eigen je podržava pomoću metoda `colwise` i `rowwise`. Proširivanje se može interpretirati kao kopiranje matrice u jednom pravcu. Pogledajmo sledeći primer koji pokazuje kako se vektor dodaje svakoj koloni matrice:

```
Eigen::MatrixXf mat(2,4);
Eigen::VectorXf v(2); // vektor kolona
mat.colwise() += v;
```

Ova operacija daje sledeći rezultat:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}.colwise() + \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ \mathbf{5} & \mathbf{6} & \mathbf{7} \end{bmatrix}$$

Upotreba biblioteke xtensor

Biblioteka `xtensor` je C++ biblioteka za numeričku analizu sa višedimenzionalnim nizovima. Kontejneri `xtensor` biblioteke inspirisani su bibliotekom NumPy –Python bibliotekom za rad sa nizovima. Pošto se mašinsko učenje najčešće opisuje u Python jeziku uz NumPy, ova biblioteka može olakšati njihovo prevođenje u C++. Sledeće klase kontejnera omogućavaju rad sa višedimenzionalnim nizovima u `xtensor` biblioteci.

Tip `xarray` predstavlja višedimenzionalni niz sa dinamičkom veličinom, što se može videti u sledećem primeru koda:

```
std::vector<size_t> shape = { 3, 2, 4 };
xt::xarray<double, xt::layout_type::row_major> a(shape);
```

Dinamička veličina za tip `xarray` znači da se oblik može promeniti tokom kompajliranja.

Tip `xtensor` predstavlja višedimenzionalni niz čiji je opseg fiksiran u vreme kompajliranja. Tačne vrednosti dimenzija mogu se podesiti u inicijalizaciji, kao što pokazuje sledeći primer koda:

```
std::array<size_t, 3> shape = { 3, 2, 4 };
xt::xtensor<double, 3> a(shape);
```

Tip `xtensor_fixed` predstavlja višedimenzionalni niz sa fiksiranom dimenzijom u vreme kompajliranja, što se vidi u sledećem primeru:

```
xt::xtensor_fixed<double, xt::xshape<3, 2, 4>> a;
```

Biblioteka `xtensor` takođe implementira aritmetičke operatore tehnikom šablona izraza, slično biblioteci `Eigen` (što je čest pristup u C++ matematičkim bibliotekama). Zbog toga se računanja izvršavaju lenjo, a pravi rezultat se računa tek kada se ceo izraz evaluira.

Lenjo izračunavanje, poznato i kao **lenja evaluacija** ili **evaluacija pozivom po potrebi** je strategija u programiranju, gde se evaluacija izraza odlaže dok njegova vrednost ne bude potrebna.

Ovo je suprotno pohlepnoj evaluaciji, gde se izrazi odmah evaluiraju kada program naiđe na njih. Kontejneri u `xtensor` biblioteci su takođe izrazi. Postoji i funkcija `xt::eval`, koja može da nametne evaluaciju izraza.

Postoje različiti načini inicijalizacije kontejnera u `xtensor` biblioteci. Inicijalizacija `xtensor` nizova može se izvršiti pomoću C++ liste inicijalizatora, kao u sledećem primeru:

```
xt::xarray<double> arr1{{1.0, 2.0, 3.0},
                        {2.0, 5.0, 7.0},
                        {2.0, 5.0, 7.0}}; // inicijalizuje 3x3 niz
```

Biblioteka `xtensor` takođe sadrži funkcije za kreiranje specijalnih tipova tenzora. Sledeći primer koda prikazuje neke od njih:

```
std::vector<uint64_t> shape = {2, 2};
auto x = xt::ones(shape); // kreira 2x2 matricu ispunjenu jedinicama
auto y = xt::zero(shape); // kreira 2x2 matricu ispunjenu nulama
auto z = xt::eye(shape);  // kreira 2x2 jediničnu matricu
```

Takođe, možemo mapirati postojeće C++ nizove u `xtensor` kontejnere pomoću funkcije `xt::adapt`. Ova funkcija vraća objekat koji koristi memoriju i vrednosti iz postojećeg objekta, kao što pokazuje sledeći primer koda:

```
std::vector<float> data{1,2,3,4};
std::vector<size_t> shape{2,2};
auto data_x = xt::adapt(data, shape);
```

Možemo koristiti direktan pristup elementima kontejnera pomoću `()` operatora za postavljanje ili promenu vrednosti tenzora, kao što pokazuje sledeći primer koda:

```
std::vector<size_t> shape = {3, 2, 4};
xt::xarray<float> a = xt::ones<float>(shape);
a(2,1,3) = 3.14f;
```

Biblioteka `xtensor` implementira aritmetičke operacije linearne algebre kroz preklopljene standardne C++ aritmetičke operatore poput `+`, `-` i `*`. Za korišćenje drugih operacija, poput skalarnih proizvoda, potrebno je povezati aplikaciju sa bibliotekom `xtensor-blas`. Ove operacije su deklarisane u prostoru imena `xt::linalg`.

Sledeći kod prikazuje upotrebu aritmetičkih operacija sa `xtensor` bibliotekom:

```
auto a = xt::random::rand<double>({2,2});
auto b = xt::random::rand<double>({2,2});
auto c = a + b;
a -= b;
c = xt::linalg::dot(a, b);
c = a + 5;
```

Da bismo dobili delimičan pristup `xtensor` kontejnerima, možemo koristiti funkciju `xt::view`. Funkcija `view` vraća novi objekat tenzora, koji deli iste osnovne podatke sa originalnim tenzorom, ali sa drugačijim oblikom ili koracima. Ovo omogućava pristup podacima na drugačiji način, bez stvarne izmene osnovnih podataka. Sledeći primer prikazuje kako ova funkcija radi:

```
xt::xarray<int> a{
  {1, 2, 3, 4},
  {5, 6, 7, 8},
  {9, 10, 11, 12},
  {13, 14, 15, 16}
};
auto b = xt::view(a, xt::range(1, 3), xt::range(1, 3));
```

Ova operacija uzima pravougaoni blok iz tenzora, koji izgleda ovako:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \rightarrow \begin{bmatrix} 6 & 7 \\ 10 & 11 \end{bmatrix}$$

Biblioteka `xtensor` implementira automatski proširivanje u većini slučajeva. Kada operacija uključuje dva niza različitih dimenzija, manja dimenzija se proširuje duž vodeće dimenzije drugog niza, tako da možemo direktno sabirati vektor i matricu. Sledeći primer koda prikazuje koliko je to jednostavno:

```
auto m = xt::random::rand<double>({2,2});
auto v = xt::random::rand<double>({2,1});
auto c = m + v;
```

Upotreba biblioteke Blaze

`Blaze` je opšta C++ biblioteka visokih performansi za guste i retke objekte linearne algebre. Postoji više klasa za predstavljanje matrica i vektora u `Blaze` biblioteci.

Možemo definisati tip matrice sa poznatim dimenzijama i tipom podataka sa pokretnim zarezom, ovako:

```
typedef blaze::
    StaticMatrix<float, 3UL, 3UL, blaze::columnMajor>
        MyMatrix33f;
```

Možemo definisati vektor na sledeći način:

```
typedef blaze::StaticVector<float, 3UL> MyVector3f;
```

Takođe, biblioteka Blaze nije ograničena samo na matrice čije su dimenzije poznate u vreme kompajliranja. Možemo definisati tipove matrica kod kojih će se broj vrsta i kolona odrediti prilikom inicijalizacije tokom izvršavanja programa. Za definisanje takvih tipova, možemo koristiti klase `blaze::DynamicMatrix` ili `blaze::DynamicVector`. Na primer, da bismo definisali matricu sa dinamičkim dimenzijama i elementima tipa double, koristimo sledeću definiciju:

```
typedef blaze::DynamicMatrix<double> MyMatrix;
```

Objekti inicijalizovani iz ovako definisanih tipova izgledaju ovako:

```
MyMatrix33f a;
MyVector3f v;
MyMatrix m(10, 15);
```

Za dodelu vrednosti ovim objektima možemo koristiti više pristupa. Možemo koristiti posebne unapred definisane funkcije za inicijalizaciju, kao što je prikazano:

```
a = blaze::zero<float>(3UL, 3UL); //  Matrica ispunjena nulama
a = blaze::IdentityMatrix<float>(3UL); //  Jedinična matrica
blaze::Rand<float> rnd;
v = blaze::generate(3UL, [&](size_t) { return rnd.generate(); });
// Slučajno generisan vektor

// Matrica popunjena pomoću liste inicijalizatora
a = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};

// Matrica popunjena jednom vrednošću
a = blaze::uniform(3UL, 3UL, 3.f);
```

Ova konstrukcija koda inicijalizuje matricu sledećim vrednostima:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Možemo koristiti direktan pristup elementima matrice pomoću `()` operatora za postavljanje ili promenu koeficijenata matrice, kao što pokazuje sledeći primer koda:

```
a(0,0) = 3;
```

Možemo koristiti objekat tipa `blaze::CustomVector` da bismo obuhvatili postojeći C++ niz ili vektor unutar objekta tipa `Matrix` ili `Vector`. Ovaj tip mapiranja koristi memoriju i vrednosti iz postojećeg objekta, bez dodatnog alociranja memorije i kopiranja vrednosti. Sledeći primer koda prikazuje kako možemo koristiti ovaj pristup:

```
std::array<int, 4> data = {1, 2, 3, 4};
blaze::CustomVector<int,
                    blaze::unaligned,
                    blaze::unpadded,
                    blaze::rowMajor>
    v2(data.data(), data.size());

std::vector<float> mdata = {1, 2, 3, 4, 5, 6, 7, 8, 9};
blaze::CustomMatrix<float,
                    blaze::unaligned,
                    blaze::unpadded,
                    blaze::rowMajor>
    a2(mdata.data(), 3UL, 3UL);
```

Možemo koristiti inicijalizovane objekte matrica u matematičkim operacijama. Kao što vidite koristili smo dva parametra: `blaze::unaligned` i `blaze::unpadded`. Parametar `unpadded` može se koristiti u nekim funkcijama ili metodima za kontrolu ponašanja popunjavanja ili skraćivanja nizova. Ovaj parametar može biti važan u situacijama kada želimo da izbegnemo nepotrebno popunjavanje ili skraćivanje podataka tokom operacija poput promene oblika, sečenja ili spajanja nizova. Parametar `blaze::unaligned` omogućava korisnicima da izvode operacije nad neporavnatim podacima, što može biti korisno u situacijama kada podaci nisu usklađeni sa specifičnim granicama memorije.

Aritmetičke operacije sa matricama i vektorima u `Blaze` biblioteci mogu se izvršavati preko preklopljenih C++ aritmetičkih operatora, poput `+`, `-` ili `*`, ili pomoću metoda kao što su `dot()` i `cross()`. Sledeći primer koda prikazuje kako možemo izraziti opšte matematičke operacije u `Blaze` biblioteci:

```
blaze::StaticMatrix<float, 2UL, 2UL> a = {{1, 2}, {3, 4}};
auto b = a;

// operacije po elementima
blaze::StaticMatrix<float, 2UL, 2UL> result = a % b;
a = b * 4;

// matrične operacije
result = a + b;
a += b;
result = a * b;
```

Važno je napomenuti da u biblioteci Blaze, aritmetički operatori poput + ne izvršavaju nikakva izračunavanja sama po sebi. Ovi operatori vraćaju *objekat izraza*, koji opisuje koju operaciju treba izvršiti. Stvarna izračunavanja se izvršavaju kasnije, kada se ceo izraz evaluira, obično u = aritmetičkom operatoru ili u konstruktoru konkretnog objekta. To može dovesti do nekih nepredviđenih ponašanja, posebno ako programer prečesto koristi `auto` ključnu reč. Biblioteka obezbeđuje dve funkcije, `eval()` i `evaluate()`, za evaluaciju izraza. Funkcija `evaluate()` pomaže u određivanju tačnog tipa rezultata operacije korišćenjem `auto` ključne reči, dok `eval()` treba koristiti za eksplicitnu evaluaciju podizraza unutar većeg izraza.

Ponekad je potrebno izvršiti operacije samo nad delom matrice. Za ovu svrhu, biblioteka Blaze obezbeđuje klase `blaze::submatrix` i `blaze::subvector`, koje mogu biti parametrizovane pomoću generičkih parametara. Ovi parametri predstavljaju početnu tačku u gornjem levom uglu i širinu i visinu oblasti. Takođe, postoje funkcije sa istim nazivima koje prihvataju iste argumente i mogu se koristiti u vreme izvršavanja. Sledeći kod prikazuje kako se koristi ova klasa:

```
blaze::StaticMatrix<float, 4UL, 4UL> m = {{1, 2, 3, 4},
                                          {5, 6, 7, 8},
                                          {9, 10, 11, 12},
                                          {13, 14, 15, 16}};

// kreiranje prikaza srednjeg dela matrice
auto b = blaze::submatrix<1UL, 1UL, 2UL, 2UL>(m);

// promena vrednosti u originalnoj matrici
blaze::submatrix<1UL, 1UL, 2UL, 2UL>(m) *= 0;
```

Postoje još dve funkcije za pristup vrstama i kolonama pomoću indeksa, što su takođe specijalni oblici `block` operacija. Sledeći primer koda prikazuje kako možemo koristiti funkcije `col` i `row`:

```
blaze::row<1UL>(m) += 3;
blaze::column<2UL>(m) /= 4;
```

Za razliku od biblioteke Eigen, biblioteka Blaze ne podržava implicitno proširivanje. Međutim, postoji funkcija `blaze::expand()`, koja može virtuelno proširiti matricu ili vektor bez stvarnog alociranja memorije. Sledeći kod prikazuje kako se koristi ova funkcija:

```
blaze::DynamicMatrix<float, blaze::rowVector> mat =
  blaze::uniform(4UL, 4UL, 2);

blaze::DynamicVector<float, blaze::rowVector> vec = {1, 2, 3, 4};
auto ex_vec = blaze::expand(vec, 4UL);

mat += ex_vec;
```

Rezultat ove operacije biće sledeći:

```
( 3 4 5 6 )
( 3 4 5 6 )
( 3 4 5 6 )
( 3 4 5 6 )
```

Upoteba biblioteke ArrayFire

`ArrayFire` je C++ biblioteka opšte namene i visokih performansi za paralelno izračunavanje.

Paralelno izračunavanje je metod rešavanja složenih problema tako što se oni dele na manje zadatke i izvršavaju istovremeno na više procesora ili jezgara. Ovaj pristup može značajno ubrzati obradu podataka u poređenju sa sekvencijalnim izvršavanjem, što ga čini ključnim alatom za aplikacije koje obrađuju velike količine podataka, poput mašinskog učenja.

Biblioteka `ArrayFire` koristi jedan tip `array` za predstavljanje matrica, trodimenzionalnih nizova i vektora. Ova notacija zasnovana na nizovima omogućava izražavanje računskih algoritama na čitljiv način, tako da korisnici ne moraju eksplicitno izražavati paralelne operacije. Podržava opsežnu vektorizaciju i paralelne operacije u grupama. Ova biblioteka omogućava ubrzano izvršavanje na CUDA i OpenCL uređajima. Još jedna zanimljiva funkcionalnost `ArrayFire` biblioteke je optimizacija korišćenja memorije i aritmetičkih operacija kroz analizu koda u vreme izvršavanja. Ovo se postiže izbegavanjem mnogih privremenih alociranja memorije.

Možemo definisati tip matrice sa poznatim dimenzijama i tipom podataka sa pokretnim zarezom ovako:

```
af::array a(3, 3, af::dtype::f32);
```

Na sledeći način možemo definisati 64-bitni vektor formata u pokretnom zarezu:

```
af::array v(3, af::dtype::f64);
```

Za dodelu vrednosti ovim objektima možemo koristiti više pristupa. Možemo koristiti posebne unapred definisane funkcije za inicijalizaciju, kao što je prikazano:

```
a = af::constant(0, 3, 3);  //  Matrica popunjena nulama
a = af::identity(3, 3);     //  Jedinična matrica
v = af::randu(3);           //  Slučajno generisan vektor

// Matrica popunjena jednom vrednošću
a = af::constant(3, 3, 3);

// Matrica popunjena pomoću liste inicijalizatora
a = af::array(
af::dim4(3, 3),
{1.f, 2.f, 3.f, 4.f, 5.f, 6.f, 7.f, 8.f, 9.f});
```

Ova konstrukcija koda inicijalizuje matricu sledećim vrednostima:

```
1.0  4.0  7.0
2.0  5.0  8.0
3.0  6.0  9.0
```

Važno je napomenuti da je matrica inicijalizovana u formatu rasporeda po kolonama. Biblioteka `ArrayFire` ne podržava inicijalizaciju rasporeda po vrstama.

Možemo koristiti direktan pristup elementima matrice pomoću `()` operatora za postavljanje ili promenu koeficijenata matrice, kao što pokazuje sledeći primer:

```
a(0,0) = 3;
```

Jedna važna razlika u odnosu na druge biblioteke koje smo pomenuli je da nije moguće mapirati postojeće C/C++ nizove na `ArrayFire` objekat `array`, jer će se podaci kopirati. Sledeći primer koda prikazuje ovu situaciju:

```
std::vector<float> mdata = {1, 2, 3, 4, 5, 6, 7, 8, 9};
a = af::array(3, 3, mdata.data());
```

Samo memorija alocirana na CUDA ili OpenCL uređajima neće biti kopirana, ali će `ArrayFire` preuzeti vlasništvo nad pokazivačem.

Možemo koristiti inicijalizovane nizove u matematičkim operacijama. Aritmetičke operacije u `ArrayFire` biblioteci mogu se izvoditi preko preklopljenih standardnih C++ aritmetičkih operatora poput `+`, `-` ili `*`, ili pomoću metoda poput `af::matmul`. Sledeći kod prikazuje kako možemo izraziti opšte matematičke operacije u `ArrayFire` biblioteci:

```
auto a = af::array(af::dim4(2, 2), {1, 2, 3, 4});
a = a.as(af::dtype::f32);
auto b = a.copy();

// operacije po elementima
auto result = a * b;
a = b * 4;

// matrične operacije
result = a + b;
a += b;
result = af::matmul(a, b);
```

Za razliku od drugih biblioteka koje smo već pomenuli, biblioteka `ArrayFire` ne koristi intenzivno generičke izraze za spajanje aritmetičkih operacija. Ova biblioteka koristi kompajliranje **po potrebi (JIT)** mehanizam, koji konvertuje matematičke izraze u računarska jezgra za CUDA, OpenCL ili CPU uređaje. Takođe, ovaj mehanizam spaja različite operacije radi postizanja najboljih performansi. Ova tehnologija fuzije operacija smanjuje broj poziva jezgra i smanjuje operacije sa globalnom memorijom.

Ponekad je potrebno izvršiti operacije samo na delu niza. Za ovu svrhu, biblioteka ArrayFire obezbeđuje specijalnu tehniku indeksiranja. Postoje posebne klase koje se mogu koristiti za izražavanje pod-opsega indeksa za određene dimenzije. One su sledeće:

```
seq – predstavlja linearnu sekvencu
end – predstavlja poslednji element dimenzije
span – predstavlja celu dimenziju
```

Sledeći kod prikazuje kako možemo pristupiti samo centralnom delu matrice:

```
auto m = af::iota(af::dim4(4, 4));
auto center = m(af::seq(1, 2), af::seq(1, 2));

// promena dela matrice
center *= 2;
```

Za pristup i ažuriranje određene vrste ili kolone u nizu, postoje metodi `row(i)` i `col(i)`, koji specificiraju jednu vrstu ili jednu kolonu. Mogu se koristiti na sledeći način:

```
m.row(1) += 3;
m.col(2) /= 4;
```

Takođe, za rad sa više vrsta ili kolona, postoje metodi `rows(first, last)` i `cols(first, last)`, koji specificiraju opseg vrsta ili kolona.

Biblioteka `ArrayFire` ne podržava implicitno proširivanja, ali postoji funkcija `af::batchFunc`, koja može simulirati i paralelizovati takvu funkcionalnost. Uopšteno, ova funkcija pronalazi dimenziju podataka grupe i primenjuje zadatu funkciju paralelno na više delova podataka. Sledeći kod prikazuje kako se koristi ova funkcija:

```
auto mat = af::constant(2, 4, 4);
auto vec = af::array(4, {1, 2, 3, 4});
mat = af::batchFunc(
    vec,
    mat,
    [](const auto& a, const auto& b) { return a + b; });
```

Rezultat ove operacije biće sledeći:

```
3.0  3.0  3.0  3.0
4.0  4.0  4.0  4.0
5.0  5.0  5.0  5.0
6.0  6.0  6.0  6.0
```

Važno je napomenuti da je vektor bio u formatu rasporeda po kolonama.

Korišćenje Dlib biblioteke

`Dlib` je savremena C++ biblioteka koja sadrži algoritme mašinskog učenja i alate za kreiranje softvera za računarski vid u jeziku C++. Većina algebarskih alata u Dlib biblioteci bavi se gustim matricama. Međutim, postoji i ograničena podrška za rad sa retkim matricama i vektorima. Konkretno, alati Dlib biblioteke predstavljaju retke vektore pomoću kontejnera iz C++ **standardne biblioteke šablona** (**STL**).

U biblioteci Dlib postoje dva glavna tipa kontejnera za rad sa linearnom algebrom: klase `matrix` i `vector`. Matrične operacije u Dlib biblioteci implementirane su pomoću tehnike generičkog izraza, što omogućava eliminaciju privremenih objekata matrica, koji bi inače bili kreirani u izrazima poput: `M = A+B+C+D`.

Možemo kreirati matricu određene veličine u vreme kompajliranja na sledeći način, tako što navodimo dimenzije kao generičke argumente:

```
Dlib::matrix<double,3,1> y;
```

Alternativno, veličinu matrice možemo kreirati dinamički. U tom slučaju, dimenzije matrice prosleđujemo konstruktoru, kao u sledećem primeru koda:

```
Dlib::matrix<double> m(3,3);
```

Kasnije možemo promeniti veličinu ove matrice pomoću sledećeg metoda:

```
m.set_size(6,6);
```

Možemo inicijalizovati vrednosti matrice pomoću operatora zarez (,), kao što pokazuje sledeći primer koda:

```
m = 54.2, 7.4, 12.1,
  1, 2, 3,
  5.9, 0.05, 1;
```

Kao i u prethodnim bibliotekama, možemo obuhvatiti postojeći C++ niz u objekat matrice, kao u sledećem primeru koda:

```
double data[] = {1,2,3,4,5,6};
auto a = Dlib::mat(data, 2,3); // kreira matricu dimenzija 2x3
```

Takođe, možemo pristupiti pojedinačnim elementima matrice pomoću operatora `()`, da bismo promenili ili dohvatili određenu vrednost, kao u sledećem primeru koda:

```
m(1,2) = 3;
```

Dlib biblioteka sadrži skup unapred definisanih funkcija za inicijalizaciju matrice vrednostima, kao što su jedinična matrica, matrica ispunjena jedinicama ili matrica sa slučajnim vrednostima, što je prikazano u sledećem primeru:

```
auto a = Dlib::identity_matrix<double>(3);
auto b = Dlib::ones_matrix<double>(3,4);
auto c = Dlib::randm(3,4); // matrica sa slučajnim vrednostima
                           // dimenzija 3x4
```

Većina aritmetičkih operacija linearne algebre u Dlib biblioteci implementirana je preko preklopljenih standardnih C++ aritmetičkih operatora, poput +, - ili *. Ostale složenije operacije dostupne su u biblioteci kao samostalne funkcije.

Sledeći primer prikazuje upotrebu aritmetičkih operacija u Dlib biblioteci:

```
auto c = a + b;
auto e = a * b; // pravo množenje matrica
auto d  = Dlib::pointwise_multiply(a, b); // množenje po elementima

a += 5;
auto t = Dlib::trans(a); // transponovana matrica
```

Za rad sa delimičnim pristupom matricama, Dlib ima skup posebnih funkcija. Sledeći primer koda prikazuje kako se koriste neke od njih:

```
a = Dlib::rowm(b,0); // uzima prvu vrstu matrice
a = Dlib::rowm(b,Dlib::range(0,1)); // uzima prve dve vrste
a = Dlib::colm(b,0); // uzima prvu kolonu

// uzima pravougaoni deo iz centra matrice:
a = Dlib::subm(b, range(1,2), range(1,2));

// inicijalizuje deo matrice:
Dlib::set_subm(b,range(0,1), range(0,1)) = 7;

// dodaje vrednost delu matrice:
Dlib::set_subm(b,range(0,1), range(0,1)) += 7;
```

Proširivanje u Dlib biblioteci može se modelovati pomoću funkcija `set_rowm()`, `set_colm()` i `set_subm()`, koje vraćaju objekte modifikatora za određenu vrstu, kolonu ili pravougaoni deo originalne matrice. Objekti koje vraćaju ove funkcije podržavaju sve operacije postavljanja vrednosti ili aritmetičke operacije. Sledeći primer koda prikazuje kako možemo dodati vektor kolonama matrice:

```
Dlib::matrix<float, 2,1> x;
Dlib::matrix<float, 2,3> m;
Dlib::set_colm(b,Dlib::range(0,1)) += x;
```

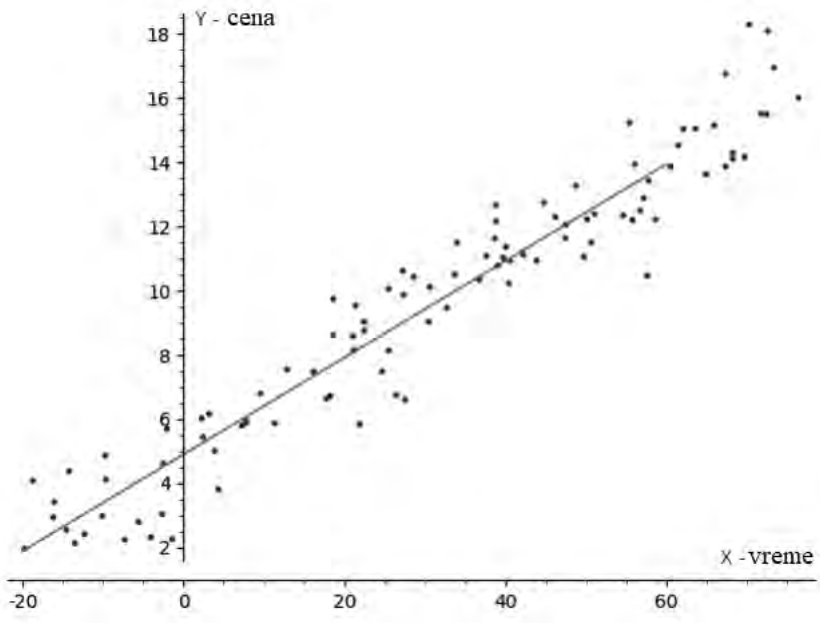
U ovom odeljku smo učili o glavnim konceptima linearne algebre i njihovoj implementaciji u raznim C++ bibliotekama. Videli smo kako se kreiraju matrice i tenzori i kako se nad njima izvode razne matematičke operacije. U narednom odeljku videćemo naš prvi potpuni primer mašinskog učenja—rešavanje regresionog problema putem linearne regresije.

Pregled linearne regresije

Posmatrajmo primer stvarnog nadgledanog algoritma mašinskog učenja koji nazivamo linearna regresija. Uopšteno, **linearna regresija** je pristup za modelovanje ciljne vrednosti (zavisne vrednosti) na osnovu prediktorske vrednosti (nezavisne vrednosti). Ovaj metod se koristi za predviđanje i pronalaženje odnosa između vrednosti. Regresione metode možemo klasifikovati prema broju ulaza (nezavisnih promenljivih) i vrsti odnosa između ulaza i izlaza (zavisnih promenljivih).

Prosta linearna regresija je slučaj kada je broj nezavisnih promenljivih 1, a odnos između nezavisne (x) i zavisne (y) promenljive je linearan.

Linearna regresija ima široku upotrebu u različitim oblastima, kao što su naučna istraživanja, gde može opisati odnose između promenljivih, kao i u industrijskim primenama, poput predviđanja prihoda. Na primer, može odrediti liniju trenda za prikaz dugoročne promene cena akcija u određenom vremenskom periodu. Ona pokazuje da li je vrednost interesa određenog skupa podataka porasla ili opala tokom određenog vremenskog perioda, kao što je ilustrovano na sledećem prikazu snimka ekrana:



Slika 1.1 – Prikaz linearne regresije

Ako imamo jednu ulaznu promenljivu (nezavisnu promenljivu) i jednu izlaznu promenljivu (zavisnu promenljivu), onda je to prosta regresija i za nju koristimo termin **prosta linearna regresija**. Kada postoji više nezavisnih promenljivih, taj tip regresije nazivamo **višestruka linearna regresija** ili **linearna regresija sa više promenljivih**. U realnim problemima obično imamo veliki broj nezavisnih promenljivih, pa se takvi problemi modeluju pomoću višestrukih regresionih modela. Višestruki regresioni modeli imaju opštu definiciju koja obuhvata i druge tipove, pa se čak i prosta linearna regresija često definiše višestrukom regresijom.

Razne biblioteke za rešavanje zadataka linearne regresije

Pretpostavimo da imamo skup podataka, $\{y_i, x_{i1}, \ldots, x_{ip}\}_{i=1}^{n}$, tako da možemo izraziti linearnu zavisnost između y i x pomoću sledeće matematičke formule:

$$y_i = \beta_0 1 + \beta_1 * x_{i1} + \ldots + \beta_{p+1} x_{ip} + \epsilon_i = x_i^T \beta + \epsilon_i, i = 1, \ldots, n$$

Ovde je p dimenzija nezavisne promenljive, a T označava transponovanje, tako da skalarni proizvod između vektora x_i i β možemo zapisati kao ${x_i}^T \beta$. Takođe, prethodni izraz možemo preformulisati u matričnoj notaciji, na sledeći način:

$$y = X\beta + \epsilon$$

$$y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

$$X = \begin{pmatrix} x_1^T \\ x_2^T \\ \vdots \\ x_n^T \end{pmatrix} = \begin{pmatrix} 1 & x_{11} & \cdots & x_{1p} \\ 1 & x_{21} & \cdots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & \cdots & x_{np} \end{pmatrix}$$

$$\beta = \begin{pmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_p \end{pmatrix}$$

$$\epsilon = \begin{pmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_n \end{pmatrix}$$

Prethodna matrična notacija može se objasniti na sledeći način:

- y: vektor opserviranih ciljnih vrednosti.
- *x*: matrica vektora vrsta, x_i koji su poznati kao prediktori ili nezavisne promenljive.
- β: vektor parametara dimenzije *p+1*.
- ε: greška ili šum. Ova promenljiva obuhvata sve druge faktore koji utiču na zavisnu promenljivu *y*, osim regresora.

Kada posmatramo prost linearni model, `p` je jednako 1,pa će jednačina imati sledeći oblik:

$$y_i = \beta_0 1 + \beta_1 x_i + \epsilon_i$$

Cilj zadatka linearne regresije jeste pronalaženje vektora parametara koji zadovoljava prethodnu jednačinu. Obično ne postoji tačno rešenje za ovaj sistem linearnih jednačina, pa je cilj ocenjivanje parametara koji uz određene pretpostavke približno zadovoljavaju ove jednačine. Jedan od najpoznatijih pristupa ocenjivanja parametara zasniva se na principu najmanjih kvadrata: minimizaciji zbira kvadrata razlika između opserviranih vrednosti zavisne promenljive u datom skupu podataka i vrednosti predviđenih linearnom funkcijom. Ovu ocenu nazivamo **obična ocena najmanjih kvadrata (OLS)**. Zadatak može biti formulisan pomoću sledeće formule:

$$\hat{\beta} = argmin_{\beta} S(\beta)$$

U prethodnoj formuli, ciljna funkcija S data je sledećom matričnom notacijom:

$$S(\beta) = \sum_{i=1}^{n} \left| y_i - \sum_{j=1}^{p} X_i j \beta_j^2 \right| = \| y - X\beta^2 \|$$

Ovaj problem minimizacije ima jedinstveno rešenje ako su `p` kolona matrice `x` linearno nezavisne. Možemo ga dobiti rešavanjem *normalne jednačine*, na sledeći način:

$$\beta = (X^T X)^{-1} X^T y$$

Biblioteke linearne algebre mogu rešiti ovakve jednačine direktno, analitičkim pristupom, ali on ima jedan značajan nedostatak—računsku složenost. Kod velikih dimenzija matrica `y` i `x`, potrebna količina memorije i vreme računanja postaju preveliki za rešavanje realnih problema.

Zbog toga se zadaci minimizacije obično rešavaju iterativnim pristupima. Jedan od takvih algoritama je **gradijentni spust (GD)**. Gradijentni spust je tehnika zasnovana na posmatranju da, ako je funkcija definisana i diferencijabilna u nekoj okolini tačke β, tada vrednost `S(β)` opada najbrže kada se pomera u pravcu negativnog gradijenta funkcije `S` u tački β.

Možemo promeniti ciljnu funkciju `S(β)` u oblik pogodniji za iterativni pristup. Za to možemo koristiti **srednje kvadratnu grešku (MSE)**, koja meri razliku između ocenjene i stvarne vrednosti, što je ilustrovano ovde:

$$S(\beta) = \frac{1}{n}\sum_{i=1}^{n}(y_i - X_i\beta)^2$$

U slučaju višestruke regresije, računamo parcijalne izvode ove funkcije za svaku komponentu `X`, na sledeći način:

$$\frac{\partial S}{\partial \beta_j}$$

Tako, u slučaju linearne regresije, dobijamo sledeće izvode:

$$\frac{\partial S}{\partial \beta_0} = \frac{2}{n}\sum_{i=1}^{n}(X_i\beta - y_i)$$

$$\frac{\partial S}{\partial \beta_1} = \frac{2}{n}\sum_{i=1}^{n}(X_i\beta - y_i)X_i$$

Ceo algoritam može se opisati na sledeći način:

1. Inicijalizovati elemente vektora parametara `β` na nule.
2. Definisati vrednost parametra brzine učenja, koji određuje koliko prilagođavamo parametre tokom procesa učenja.
3. Računati sledeće vrednosti `β`:

$$\beta_0 = \beta_0 - \gamma\frac{2}{n}\sum_{i=1}^{n}(X_i\beta - y_i)$$

$$\beta_1 = \beta_1 - \gamma\frac{2}{n}\sum_{i=1}^{n}(X_i\beta - y_i)X_i$$

4. Ponavljati korake 1–3 određeni broj puta ili dok vrednost srednje kvadratne greške ne dostigne prihvatljiv nivo.

Prethodno opisani algoritam jedan je od najjednostavnijih nadgledanih algoritama mašinskog učenja. Opisali smo ga pomoću koncepata linearne algebre koje smo objasnili ranije u ovom poglavlju. Kasnije postaje očigledno da gotovo svi algoritmi mašinskog učenja koriste linearnu algebru u pozadini. Linearna regresija se široko koristi u različitim industrijama za prediktivnu analizu, prognoziranje i donošenje odluka. Evo nekoliko stvarnih primera primene linearne regresije u finansijama, marketingu i zdravstvu. Linearna regresija može se koristiti za predviđanje cena akcija na osnovu istorijskih podataka kao što su zarada kompanije, kamatne stope i ekonomski pokazatelji. Ovo pomaže investitorima da donesu informisane odluke o tome kada kupiti ili prodati akcije. Modeli linearne regresije mogu se kreirati za predviđanje ponašanja kupaca na osnovu demografskih informacija, istorije kupovine i drugih relevantnih podataka. Ovo omogućava marketinškim stručnjacima da preciznije ciljaju kampanje i optimizuju potrošnju na marketing. Linearna regresija se koristi za analizu medicinskih podataka kako bi se identifikovali obrasci i veze koji mogu poboljšati ishode lečenja pacijenta. Na primer, može se koristiti za proučavanje uticaja određenih tretmana na zdravlje pacijenata.

Sledeći primeri prikazuju API interfejs višeg nivoa u različitim bibliotekama linearne algebre za rešavanje zadatka linearne regresije, a dajemo ih da bismo pokazali kako biblioteke mogu pojednostaviti složenu matematiku koja se koristi u pozadini. Detalje API inteferjsa korišćenih u ovim primerima objasnićemo u narednim poglavljima.

Biblioteka Eigen za rešavanje zadataka linearne regresije

Postoji nekoliko iterativnih metoda za rešavanje problema tipa $Ax = b$ u biblioteci Eigen. Klasa `LeastSquaresConjugateGradient` je jedna od njih i omogućava nam da rešimo probleme linearne regresije pomoću **algoritma konjugovanog gradijenta**. Algoritam `ConjugateGradient` može konvergirati ka minimumu funkcije brže od običnog **gradijentnog spusta**, ali zahteva da matrica `A` bude **pozitivno definitna** da bi se garantovala **numerička stabilnost.** Klasa `LeastSquaresConjugateGradient` ima **dva glavna podešavanja**: **maksimalan broj iteracija i prag tolerancije**, koji se koristi kao kriterijum zaustavljanja, odnosno gornja granica relativne rezidualne greške. Sledeći blok koda to ilustruje:

```
typedef float DType;
using Matrix = Eigen::Matrix<DType, Eigen::Dynamic, Eigen::Dynamic>;

int n = 10000;
Matrix x(n,1);
Matrix y(n,1);

Eigen::LeastSquaresConjugateGradient<Matrix> gd;
gd.setMaxIterations(1000);
gd.setTolerance(0.001);
gd.compute(x);
auto b = gd.solve(y);
```

Za nove x ulazne vrednosti, možemo predvideti nove y vrednosti pomoću matričnih operacija, kao što je prikazano u sledećem primeru:

```
Eigen::MatrixXf new_x(5, 2);
new_x << 1, 1, 2, 1, 3, 1, 4, 1, 5;
auto new_y = new_x.array().rowwise() * b.transpose().array();
```

Takođe, možemo izračunati vektor parametara b (rešenje zadatka linearne regresije) rešavanjem *normalne jednačine* direktno, na sledeći način:

```
auto b = (x.transpose() * x).ldlt().solve(x.transpose() * y);
```

Biblioteka Blaze za rešavanje zadataka linearne regresije

Pošto je Blaze samo matematička biblioteka, ne postoje posebne klase ili funkcije za rešavanje zadataka linearne regresije. Međutim, pristup preko normalne jednačine može se lako implementirati. Pogledajmo kako možemo definisati i rešiti zadatke linearne regresije pomoću Blaze biblioteke.

Pretpostavimo da imamo skup podataka za obuku:

```
typedef blaze::DynamicMatrix<float, blaze::columnMajor> Matrix;
typedef blaze::DynamicVector<float, blaze::columnVector> Vector;

// prva kolona matrice X je ispunjena jedinicama zbog pristrasnog člana
Matrix x(n, 2UL);
Matrix y(n, 1UL);
```

Zatim, koeficijente linearne regresije možemo izračunati na sledeći način:

```
// računanje X^T * X
auto xtx = blaze::trans(x) * x;

// računanje inverzne matrice (X^T * X)^(-1)
auto inv_xtx = blaze::inv(xtx);

// računanje X^T * y
auto xty = blaze::trans(x) * y;

// računanje koeficijenata linearne regresije
Matrix beta = inv_xtx * xty;
```

Zatim možemo koristiti ocenjene koeficijente za pravljenje predikcija na novim podacima. Sledeći deo koda prikazuje kako se to radi:

```
auto line_coeffs = blaze::expand(
  blaze::row<0UL>(blaze::trans(beta)), new_x.rows());
auto new_y = new_x % line_coeffs;
```

Važno je napomenuti da virtuelno proširujemo vektor koeficijenata kako bismo mogli da izvršimo množenje po elementima sa podacima x.

Biblioteka ArrayFire za rešavanje zadataka linearne regresije

Niti biblioteka ArrayFire nema posebne funkcije i klase za rešavanje ove vrste problema. Međutim, pošto sadrži sve potrebne matematičke apstrakcije, može se primeniti pristup normalne jednačine. Drugi pristup je iterativni metod koji koristi gradijentni spust. Taj algoritam je opisan u prvom delu ovog poglavlja. Ovaj pristup eliminiše potrebu za računanjem inverznih matrica, što ga čini pogodnim za veće skupove podataka za obuku. Računanje inverzne matrice za veliku matricu je računski veoma zahtevna operacija.

Definisaćemo lambda funkciju koja računa vrednosti predikcije na osnovu podataka i koeficijenata:

```
auto predict = [](auto& v, auto& w) {
  return af::batchFunc(v, w, [](const auto& a, const auto& b) {
    return af::sum(a * b, /*dim*/ 1);
  });
};
```

Pretpostavimo da imamo skup podataka za obuku definisan kroz promenljive `x` i `y`. Definišemo promenljivu `train_weights` koja će čuvati i ažurirati koeficijente koje model uči iz podataka za obuku:

```
// prva kolona je za pristrasni član
af::dim4 weights_dim(1, 2);
auto train_weights = af::constant(0.f, weights_dim, af::dtype::f32);
```

Zatim možemo definisati petlju gradijentnog spusta u kojoj ćemo iterativno ažurirati koeficijente. Sledeći kod prikazuje kako se to implementira:

```
af::array j, dj;  // vrednost gubitka i njegov gradijent
float lr = 0.1f;  // brzina učenja
int n_iter = 300;
for (int i = 0; i < n_iter; ++i) {
  std::cout << "Iteracija " << i << ":\n";
  // računanje gubitka
  auto h = predict(x, train_weights);
  auto diff = (y - h);
```

```
  auto j = af::sum(diff * diff) / n;
  af_print(j);

  // pronalaženje gradijenta gubitka
  auto dm = (-2.f / n) * af::sum(x.col(1) * diff);
  auto dc = (-2.f / n) * af::sum(diff);
  auto dj = af::join(1, dc, dm);

  // ažuriranje parametara pomoću gradijentnog spusta
  train_weights = train_weights - lr * dj;
}
```

Najvažniji deo ove petlje je računanje greške predikcije:

```
auto h = predict(x, train_weights);
auto diff = (y - h);
```

Drugi važan deo je izračunavanje vrednosti gradijenta na osnovu parcijalnih izvoda u odnosu na svaki od koeficijenata:

```
auto dm = (-2.f / n) * af::sum(x.col(1) * diff);
auto dc = (-2.f / n) * af::sum(diff);
```

Spajamo ove vrednosti u jedan vektor kako bismo dobili jedan izraz za ažuriranje parametara za obuku modela:

```
auto dj = af::join(1, dc, dm);
train_weights = train_weights - lr * dj;
```

Ovu petlju za obuku možemo zaustaviti nakon određenog broja iteracija ili kada vrednost funkcije gubitka dostigne odgovarajuću vrednost konvergencije. Vrednost funkcije gubitka može se izračunati na sledeći način:

```
auto j = af::sum(diff * diff) / n;
```

Ovo je jednostavno suma kvadrata grešaka za sve uzorke obučavanja.

Biblioteka Dlib za rešavanje zadataka linearne regresije

Biblioteka `Dlib` obezbeđuje klasu `krr_trainer`, koja može koristiti `linear_kernel` kao generički argument za rešavanje zadataka linearne regresije. Ova klasa implementira direktno analitičko rešavanje ovog tipa problema pomoću kernel grebenog regresionog algoritma, kao što je prikazano u sledećem kodu:

```
std::vector<matrix<double>> x;
std::vector<float> y;
krr_trainer<KernelType> trainer;
trainer.set_kernel(KernelType());
decision_function<KernelType> df = trainer.train(x, y);
```

Za nove `x` ulazne vrednosti možemo predvideti nove `y` vrednosti na sledeći način:

```
std::vector<matrix<double>> new_x;
for (auto& v : x) {
    auto prediction = df(v);
    std::cout << prediction << std::endl;
}
```

U ovom odeljku naučili smo da rešavamo probleme linearne regresije pomoću raznih C++ biblioteka. Videli smo da neke od njih sadrže kompletnu implementaciju algoritma, koju možemo lako primeniti i videli smo kako da implementiramo ovaj pristup od početka, samo pomoću osnovnih operacija linearne algebre.

Rezime

U ovom poglavlju naučili smo šta je mašinsko učenje, kako se razlikuje od drugih računarskih algoritama i kako je postalo toliko popularno. Takođe smo upoznali matematičke osnove neophodne za rad sa algoritmima mašinskog učenja. Pregledali smo softverske biblioteke koje pružaju API interfejse za linearnu algebru i implementirali naš prvi algoritam mašinskog učenja – linearnu regresiju.

Postoje i druge biblioteke linearne algebre za jezik C++. Osim toga, popularni okviri za duboko učenje koriste sopstvene implementacije biblioteka linearne algebre. Na primer, MXNet okvir je zasnovan na biblioteci `mshadow`, dok PyTorch koristi biblioteku ATen. Neke od ovih biblioteka mogu koristiti GPU ili specijalne CPU instrukcije za ubrzavanje izračunavanja. Ove funkcionalnosti obično ne menjaju API interfejs, ali mogu zahtevati dodatna podešavanja prilikom inicijalizacije biblioteke ili eksplicitnu konverziju objekata između različitih sistema, kao što su procesori ili grafički procesori.

Projekti mašinskog učenja mogu biti složeni i zahtevni. Česte prepreke predstavljaju problemi sa kvalitetom podataka, prekomerno ili nedovoljno prilagođavanje modela, pogrešan izbor modela i ograničeni računarski resursi. Loš kvalitet podataka može značajno uticati na performanse modela, zbog čega je važno izvršiti pravilnu pripremu podataka, ukloniti netipične vrednosti, obraditi podatke koji nedostaju i transformisati karakteristike za bolju reprezentaciju. Model se bira u zavisnosti od prirode problema i dostupnih podataka. Prekomerno prilagođavanje se dešava kada model nauči podatke

napamet, umesto da prepoznaje opšte obrasce, dok se nedovoljno prilagođavanje javlja kada model ne uspeva da prepozna osnovnu strukturu podataka. Da bi se izbegle ove greške, važno je jasno definisati problem, primeniti odgovarajuće tehnike obrade podataka, izabrati odgovarajući model i koristiti relevantne metrike za procenu performansi. Dobre prakse u mašinskom učenju su neprekidno praćenje performansi modela u proizvodnji i prilagođavanje modela po potrebi. Ove tehnike ćemo detaljno obraditi u nastavku knjige. U naredna dva poglavlja upoznaćemo softverske alate potrebne za implementaciju složenijih algoritama i istražiti teorijske osnove upravljanja algoritmima mašinskog učenja.

Dodatna literatura

- *Osnovni koncepti linearne algebre za duboko učenje*: `https://towardsdatascience.com/linear-algebra-for-deep-learning-f21d7e7d7f23`
- *Duboko učenje*, izdavač MIT Press: `https://www.deeplearningbook.org/contents/linear_algebra.html`
- Šta je mašinsko učenje?: `https://www.mathworks.com/discovery/machine-learning.html`
- Dokumentacija biblioteke `Eigen`: `https://gitlab.com/libeigen/eigen`
- Dokumentacija biblioteke `xtensor`: `https://xtensor.readthedocs.io/en/latest/`
- Dokumentacija biblioteke `Dlib`: `http://dlib.net/`
- Dokumentacija biblioteke `blaze`: `https://bitbucket.org/blaze-lib/blaze/wiki/Home`
- Dokumentacija biblioteke `ArrayFire`: `https://arrayfire.org/docs/index.htm`