Chapter 7 - Machine learning domains

Joydeep Bhattacharjee

June 8, 2019

In the previous two chapters, we took a look at how machine learning is approached in the two major domains of NLP and computer vision. These domains cover the major breakthroughs of machine learning and the state of the art is continually being pushed forward in these domains. But a lot of machine learning and high performance computing come outside these domains as well. In this chapter we will take a look at some of these domains and how Rust can help in creating applications in these domains. We shall start with Statistical Analysis where we will compute z-scores for different diseases in a genetic data set. Then we will move on to understanding how high performance code can be written using SIMD and BLAS libraries in Rust. Finally we will build a good books recommender in Rust.

By the end of this chapter, we should have a fair understanding of how machine learning applications can be built in different domains.

0.1 Statistical Analysis

As data scientists and machine learning engineers, we will need to perform a lot of statistical analysis on different types of data. The underlying assumption of this section is that if we are able to parse the data and store different measurements of a variable in a ndarray matrix, we should be able to perform statistical analysis on them.

As an example on how Rust enables us to perform statistical analysis on raw data, we can perform a simple differential expression analysis on a gene expression dataset. We will use false discovery rates to provide interpretable results when conducting an analysis that involves large scale multiple hypothesis testing. Note the format of the data set and how we will be reading the dataset to create the vectors of our choice.

We can download the raw data from this website
https://www.ncbi.nlm.nih.gov/sites/GDSbrowser?acc=GDS1615.

```
$ wget ftp://ftp.ncbi.nlm.nih.gov/geo/datasets/GDS1nnn/GDS1615/soft/GDS1615_full.s
$ gunzip GDS1615_full.soft.gz
```

Once we have downloaded the data we will see that the file is in this format.

```
DATABASE = Geo
### DATASET HEADERS Go HERE ...
!dataset_table_begin
## DATASET HEADERS HERE ...
1007_s_at          MIR4640 80.7287 ...
// THE INDIVIDUAL RECORDS ...
AFFX-TrpnX-M_at --Control          1.62238 ...
!dataset_table_end
```

The first 193 lines are the headers of the whole request and other metadata of the data records in the soft file. The records are kept between the lines !dataset_table_begin ... !dataset_table_end. And the first record after dataset_table_begin contains the following datastructures

- GID : A list of gene identifiers of length d
- SID : A list of sample identifiers of length n
- STP : A list of sample descriptions of length d
- X : A dxn array of gene expression values

This dataset contains analysis of peripheral blood mononuclear cells (PBMCs) from 59 Crohn's disease patients and 26 ulcerative colitis (UC) patients. There are 22283 samples. In this section, we will consider the mean expression levels in the two groups.

Moving on to the code part. We will first need to parse through the whole file. So we create a process_file function and pass the path to the file. In side the function we can implement the loop to read the lines.

Listing 1: chapter7/statistics/src/main.rs

```
use std::path::Path;
use std::fs::File;
use std::io::{BufRead, BufReader};

fn process_file(filename: &Path) {
  let file = File::open(filename).unwrap();
  for line in BufReader::new(file).lines() {
    let thisline = line?;
    // business logic on the lines implemented.
  }
}

fn main() {
  let filename = Path::new("GDS1615_full.soft");
  process_file(&filename).unwrap();
}
```

Now before moving ahead, let us talk about the dependencies that we would need. Since we will be converting our vectors to ndarray matrices we will need ndarray. As an option, if you have higher statistics to be measured, we can also bring in ndarray-stats which gives us a couple of more options when dealing with ndarray vectors.

Now we should be able to read in the headers. The subset_description comes first and then in the next line subset_sample_id are given as a list. An example of this is shown below.

```
// prev data ..
!subset_description = normal
!subset_sample_id = GSM76115,...
!subset_type = disease state
// subsequent data ...
```

Keeping that in mind we can write the below code.

Listing 2: chapter7/statistics/src/main.rs

```
fn process_file(filename: &Path) {
  let mut SIF = HashMap::new();
  let mut subset_description = String::new();
  let mut within_headers = true;
  for line in BufReader::new(file).lines() {
    if within_headers {
      if thisline.starts_with("!subset_description") {
        subset_description = line_split[1].trim().to_owned();
      };
      let subset_ids = if thisline.starts_with("!subset_sample_id") {
        let subset_ids = line_split[1].split(",");
        let subset_ids = subset_ids.map(|s| s.trim().to_owned());
        subset_ids.collect()
      } else {
        Vec::new()
      };
      for k in subset_ids {
        SIF.insert(k, subset_description.to_owned());
      }
    }
  }
  // rest of the code ...
}
```

In the above code SIF is a mapping from the different subset_ids in the line with subset_sample_id to the subset_description mentioned above them. This needs to go on till dataset_table_begin is reached.

Listing 3: chapter7/statistics/src/main.rs

```
'linereading: for line in BufReader::new(file).lines() {
  let thisline = line?;
  let line_split: Vec<String> = thisline.split("=").map(|s| s.to_owned()).collect(
  if thisline.starts_with("!dataset_table_begin") {
      within_dataset_table = true;
      within_headers = false;
      continue 'linereading;
  }
// rest of the code ...
}
```

Once the dataset_table_begin is reached, we turn the flag within_headers to false and then jump to the next line. That way in the remaining line reads the if within_headers block is not executed anymore.

Once the mapping between subset sample id to subset description (SIF) is done, we should be able to parse through the headers which is the first line after dataset_table_begin. Notice that in the thisline .starts_with ("!dataset_table_begin") block the within_dataset_table is assigned as true suggesting that we are within the table. We will also have another variable gene_expression_headers to sug-

gest that the header line as passed. We will create a function the header line as passed. We will create a function process_gene_expresssion_data_headers to process the headers for the header line. This function will take the SIF mapping created earlier and will create the column indices for the columns to which the sample descriptions are mapped.

Listing 4: chapter7/statistics/src/main.rs

```
fn process_gene_expresssion_data_headers(
    thisline: &String,
    SIF: &HashMap<String, String>)
    -> (Vec<String>, Vec<String>, Vec<usize>) {
  let SID: Vec<String> = thisline.split("\t").map(|s| s.to_owned()).collect();
  let indices: Vec<usize> = SID.iter()
              .enumerate()
              .filter(|&(_, x)| x.starts_with("GSM") )
              .map(|(i, _)| i).collect();
  let SID: Vec<String> = indices.iter().map(|&i| SID[i].clone()).collect();
  let STP: Vec<String> = SID.iter().map(
      |k| SIF.get(&k.to_string()).unwrap())
      .cloned().collect();
  (SID, STP, indices)
}

fn process_file(filename: &Path) -> io::Result<HashMap<String, String>> {
  //prev code ..
  let mut gene_expression_headers = true;
  let mut indices: Vec<usize> = Vec::new();
  let mut gene_expression_measures_vec = Vec::new();
  let mut gene_identifiers_vec = Vec::new();
  let mut SID = Vec::new();
  let mut STP = Vec::new();
  'linereading: for line in BufReader::new(file).lines() {
    let thisline = line?;
    let line_split: Vec<String> = thisline.split("=").map(|s| s.to_owned()).collect();
    if within_dataset_table && gene_expression_headers {
      let sid_stp_indices = process_gene_expresssion_data_headers(&thisline, &SIF);
      indices = sid_stp_indices.2.clone();
      SID = sid_stp_indices.0.clone();
      STP = sid_stp_indices.1.clone();
      gene_expression_headers = false;
      continue 'linereading;
    };
  // remaining code...
}
```

In the above code the indices are created when the individual identifiers start with "GSM". Once the identifiers are there the mapping to the description are understood from SIF and then the values STD, STP and indices are returned to be used later. Once this header line is parsed we do not want this

line to be evaluated anymore and hence we continue the loop after assigning gene_expression_headers to be false so that this block is not executed in any more of the consecutive line reads.

Now that all the header information have been parsed we run through the gene data which runs from line 196 to the end in the soft file. This comes under the unique condition of being in the dataset table and not being the gene_expression_headers. Hence we can resort to the following code.

Listing 5: chapter7/statistics/src/main.rs

```
if within_dataset_table && !gene_expression_headers {
  if thisline.starts_with("!dataset_table_end") {
    break 'linereading;
  }
  let (gene_expression_measures, gene_identifiers) = process_gene_expresssion_data
  gene_expression_measures_vec.extend(gene_expression_measures);
  gene_identifiers_vec.push(gene_identifiers);
}
```

process_gene_expresssion_data is the function that will handle this parsing of the lines and putting it in gene_expression_measures_vec and gene_identifiers_vec . The gene_expression_measures_vec will have all the vectors for the different headers and the gene_identifiers_vec will have the concatenated of id references and identifiers.

Once the gene_expression_measures_vec has been created we can convert them to ndaray matrices.

Listing 6: chapter7/statistics/src/main.rs

```
use ndarray;
use ndarray::{Array, Array2, Array1, Axis, stack};

fn convert_to_log_scale(X: &Array2<f64>) -> Array2<f64> {
  let two = 2.0f64;
  let two_log = two.ln();
  X.mapv(|x| x.ln()/two_log)
}

let gene_expression_measures_matrix = Array::from_shape_vec(
  (22283, 127), gene_expression_measures_vec).unwrap();
let gene_expression_measures_matrix = convert_to_log_scale(
  &gene_expression_measures_matrix);
```

From this matrix we would need to separate out the ulcerative colitis and Crohn's disease sample indices.

Listing 7: chapter7/statistics/src/main.rs

```
fn filter_specific_samples(
    STP: &Vec<String>,
    group_type: &str) -> Vec<usize> {
  STP.iter().enumerate()
```

```
    .filter(|&(_, x)| x == group_type)
    .map(|(i, _)| i).collect()
}

fn different_samples(
    STP: &Vec<String>) -> (Vec<usize>, Vec<usize>) {
  let UC = filter_specific_samples(&STP, "ulcerative colitis");
  let CD = filter_specific_samples(&STP, "Crohn's disease");
  (UC, CD)
}

let (UC, CD) = different_samples(&STP);
```

Now that we have the data, we can calculate the mean and variance of each group which will be used to calculated the z-scores to summarize the evidence of differential expression. To construct the z-scores, we will need to construct some functions. The first function is to filter out the columns from gene_expression_measures_matrix if we pass the samples indices. The idea is that we should be able to pass the relevant UC and CD indices and get the sub matrices.

Listing 8: chapter7/statistics/src/main.rs

```
fn filter_out_relevant_columns(samples: &Vec<usize>,
    gene_expression_measures_matrix: &Array2<f64>) -> Array2<f64> {
  let shape1 = samples.len();
  let shape0 = gene_expression_measures_matrix.shape()[0];
  let mut cols = Vec::new();
  for &msamples_columns in samples {
    let col = gene_expression_measures_matrix.column(
      msamples_columns);
    cols.push(col);
  }
  let Msamples = stack(Axis(0), &cols[..]).unwrap();
  let Msamples = Array::from_iter(Msamples.iter());
  let Msamples = Msamples.into_shape((shape0, shape1)).unwrap();
  Msamples.mapv(|&x| x)
}
```

Now that we have the sub-matrices we can either construct the mean of the samples or the variance of the samples which is a simple function in ndarray.

Listing 9: chapter7/statistics/src/main.rs

```
fn mean_of_samples(samples: &Vec<usize>,
    gene_expression_measures_matrix: &Array2<f64>) -> Array1<f64> {
  let Msamples = filter_out_relevant_columns(
    samples, gene_expression_measures_matrix);
  Msamples.mean_axis(Axis(1))
}

fn variance_of_samples(
```

```
    samples: &Vec<usize>,
    gene_expression_measures_matrix: &Array2<f64>)
    -> Array1<f64> {
  let Msamples = filter_out_relevant_columns(samples, gene_expression_measures_ma
  Msamples.var_axis(Axis(1), 1.)
}
```

Thus we are finally able to compute the zscores for the two samples together.

Listing 10: chapter7/statistics/src/main.rs

```
fn generate_zscores(UC: &Vec<usize>,
                    CD: &Vec<usize>,
                    gene_expression_measures_matrix: &Array2<f64>)
                    -> Array1<f64> {
    let MUC = mean_of_samples(&UC, &gene_expression_measures_matrix);
    let MCD = mean_of_samples(&CD, &gene_expression_measures_matrix);
    let VUC = variance_of_samples(&UC, &gene_expression_measures_matrix);
    let VCD = variance_of_samples(&CD, &gene_expression_measures_matrix);
    let nUC = UC.len();
    let nCD = CD.len();
    let z_scores_num = MUC - MCD;
    let z_scores_den = (VUC/nUC as f64 + VCD/nCD as f64).mapv(f64::sqrt);
    let z_scores = z_scores_num / z_scores_den;
    z_scores
}

let z_scores = generate_zscores(&UC, &CD, &gene_expression_measures_matrix);
let z_scores_mean = z_scores.sum() / z_scores.len() as f64;
let z_scores_std = z_scores.std_axis(Axis(0), 1.);
println!("z scores mean {:?}", z_scores_mean);
println!("z scores mean {:?}", z_scores_std);
```

If a gene is not differentially expressed, it has the same expected values in the two groups of samples. In this case the Z-scores will be standardised or will have zero mean and unit standard deviation. Printing out z_scores_mean and z_scores_std we find that the values are 0.042918212228268235 and 3.5369539066322027 respectively. Notice that since the vectors are in ndarray format, we can easily compute the stats using the ndarray api.

Since the standard deviation is much greater than 1, there appear to be multiple genes for which the mean expression levels in the ulcerative colitis and Crohn's disease samples differ. Further, since the mean Z-score is positive, it appears that the dominant pattern is for genes to be expressed at a higher level in the ulcerative colitis compared to the Crohn's disease samples.

Similarly we can find the p-values. The exact results are inconsequential for this section, the aim of which is to show that once values are converted to ndarray matrices in Rust it is fairly trivial to perform statistical analysis on them.

0.2 Writing high performance code

A lot of the machine learning code that have the ability to use GPU is through Rust bindings to libraries that are essentially created in C/C++. To be able to use the GPU or write high performance code we can use crates such as blas-src or lapack-src which provide access to low-level mathematical operations.

In the majority of the cases the high level crates and libraries that we have discussed in the previous chapters would be enough, but sometimes we would need to go low-level and perform direct vector operations. The Single Instruction Multiple Data operation library, or more popularly known as SIMD, is a popular library for performing vector operations and the faster[1] crate gives us good abstractions for running SIMD instructions in Rust.

SIMD and faster are great if we are trying to get parallel instructions being run on a vector and to return the result. The transformation runs on each element of the vector. In the below code we will take a vector and return the cube of all the elements in the vector.

Now before running the below code you should probably have blas and lapack libraries installed in the system. Installing in an ubuntu system might need a command such as below.

Listing 11: chapter7/high-performance-computing/src/main.rs

```
$ sudo apt-get install libblas-dev liblapack-dev
$ sudo apt-get install libopenblas-dev
$ sudo apt-get install gfortran
```

Listing 12: chapter7/high-performance-computing/src/main.rs

```
use faster::*;

fn main() {
    let my_vector: Vec<f32> = (0..10).map(
        |v| v as f32).collect();
    let power_of_3 = (&my_vector[..]).simd_iter(f32s(0.0))
        .simd_map(|v| v * v * v)
        .scalar_collect();
    println!("{:?}", power_of_3);
}
```

Similarly when we are trying to perform a reduction on the vector we can use the simd_reduce method. For example in the below code we will try to find the sum of the elements in a vector.

Listing 13: chapter7/high-performance-computing/src/main.rs

```
let reduced = (&power_of_3[..])
        .simd_iter(f32s(0.0))
        .simd_reduce(f32s(0.0), |a, v| a + v ).sum();
```

[1] https://github.com/AdamNiederer/faster

Another common operation that is performed on vectors is the dot product of two vectors and in that case we will use the rblas crate which is a wrapper over blas and openblas libraries. BLAS which stands for Basic Linear Algebra Specification are a set of low level routines for performing common linear operation. The rblas crate has the Dot product implemented over blas[**OT:4**].

In the below example we take the dot product of two vectors initialised.

Listing 14: chapter7/high-performance-computing/src/main.rs

```
use rblas::Dot;

fn main() {
    let x = vec![1.0, -2.0, 3.0, 4.0];
    let y = [1.0, 1.0, 1.0, 1.0, 7.0];

    let d = Dot::dot(&x, &y[..x.len()]);
    println!("dot product {:?}", d);
}
```

Thus using these wrappers over high performance libraries, we should be able to write high performance code in our Rust applications.

0.3 Recomender systems

Recommender systems are one of the most successful and widespread applications of machine learning technologies in business. Machine learning algorithms in recommender systems are typically classified into two categories - content based filtering and collaborative filtering although modern architectures mainly employ a combination of both. Content based methods are based on similarity of item attributes and collaborative methods calculate similarity from interactions.

In Rust, we are able to create recommendation engines mostly due to the great sbr-rs crate[2]. In this crate there are two models that are implemented. This crate will need some additional OS level dependencies installed. Ensure that you have libssl installed in your system. In an ubuntu system running the following command should be fine.

```
$ sudo apt install libssl-dev
```

- LSTM: an LSTM network is used to model the sequence of a user's interaction to predict their next action;
- EWWA: This model uses an exponentially-weighted average of past actions to predict future interactions.

To start the project we can create a project using cargo

[2]https://github.com/maciejkula/sbr-rs/

```
$ cargo new --bin goodbooks-recommender
```

This will setup the project with the src/main.rs file and the Cargo.toml files as we have seen in the previous chapters. We can add the dependencies in the cargo file. Some of the dependencies, we have already seen such as reqwest to serve as a web client. Failure is an excellent crate for better handling of errors. The crates serde, serde_json and serde_json are there to be able to serialize and deserialize the data. We will need the csv crate as we will be working with csv files. The crate rand will help us in randomizing the data and structopt to crate good command line interfaces. The most important crate as per this section is the sbr crate which has the recommendation modules implemented that we will call here.

Listing 15: chapter7/goodbooks-recommender/Cargo.toml

```
[package]
name = "goodbooks-recommender"
version = "0.1.0"
authors = ["author names"]
edition = "2018"

[dependencies]
reqwest = "0.9.17"
failure = "0.1.5"
serde = "1"
serde_derive = "1"
serde_json = "1"
csv = "1"
sbr = "0.4.0"
rand = "0.6.5"
structopt = "0.2.15"
```

0.3.1 Command line

We will now start with the creation of the command line. For readers who are familiar with git, you might have noticed that git has first level commands such as git clone or git pull and then dependending on the first command we will have a second level of commands such as origin master like a subcommand for pull which enables us to write git pull origin master. Along similar lines we will try to build a command line parser.

Listing 16: goodbooks-recommender/src/main.rs

```
use structopt::StructOpt;

#[derive(Debug, StructOpt)]
#[structopt(name = "goodbooks-recommender", about = "Books Recommendation")]
enum Opt {
  #[structopt(name = "fit")]
  /// Will fit the model.
```

```
  Fit,
  #[structopt(name = "predict")]
  /// Will predict the model based on the string after this. Please run
  /// this only after fit has been run and the model has been saved.
  Predict(BookName),
}

// Subcommand can also be externalized by using a 1-uple enum variant
#[derive(Debug, StructOpt)]
struct BookName {
  #[structopt(short = "t", long = "text")]
  /// Write the text for the book that you want to predict
  /// Multiple books can be passed in a comma separated manner
  text: String,
}

fn main() {
  let opt = Opt::from_args();
}
```

In the above code, the goodbooks-recommender takes in a first command which is essentially an enum of either Fit or Predict. Fit is fine by itself but Predict takes in a separate subcommand to it which is a text. The help function also gives documentation on this.

```
$ cargo run -- --help
    Finished dev [unoptimized + debuginfo] target(s) in 0.23s
     Running `target/debug/goodbooks-recommender --help`
goodbooks-recommender 0.1.0
Joydeep Bhattacharjee <joydeepubuntu@gmail.com>
Books Recommendation

USAGE:
    goodbooks-recommender <SUBCOMMAND>

FLAGS:
    -h, --help       Prints help information
    -V, --version    Prints version information

SUBCOMMANDS:
    fit        Will fit the model.
    help       Prints this message or the help of the given subcommand(s)
    predict    Will predict the model based on the string after this. Please run this only after fit has been r
               and the model has been saved.
```

We are also able to see the help text for predict

```
$ cargo run -- predict --help
    Finished dev [unoptimized + debuginfo] target(s) in 0.23s
     Running `target/debug/goodbooks-recommender predict --help`
goodbooks-recommender-predict 0.1.0
Joydeep Bhattacharjee <joydeepubuntu@gmail.com>
Will predict the model based on the string after this. Please run this only after fit has been run and the mode
been saved.

USAGE:
    goodbooks-recommender predict --text <text>

FLAGS:
    -h, --help       Prints help information
    -V, --version    Prints version information

OPTIONS:
    -t, --text <text>    Write the text for the book that you want to predict Multiple books can be passed in a
                         separated manner
```

0.3.2 Dowloading data

We should now be able to create functions for downloading the data and creating saving the files in the destination folders.

Listing 17: chapter7/goodbooks-recommender/Cargo.toml

```
use std::fs::File;
use std::io::BufWriter;
use std::path::Path;

fn download(url: &str, destination: &Path)
            -> Result<(), failure::Error> {

  if destination.exists() {
    return Ok(())
  }

  let mut writer = BufWriter::new(file);
  let mut response = reqwest::get(url)?;
  response.copy_to(&mut writer)?;

  Ok(())
}
```

In the above download function, we take the url as a str and destination as a path. If the path exists then nothing needs to be done. We will leave to the caller of this function to ensure that the url and destination combinations are correct. Then we will get the response from the url and write it to the path using BufWriter module. This should download any url that we pass to the function.

Using the above function we can download the good books data set from this github repo https://github.com/zygmuntz/goodbooks-10k/. This data set contains six million ratings for ten thousand most popular books. The other kinds of information that is provided in the data sets are. We will be downloading the ratings.csv and the books.csv

Listing 18: chapter7/goodbooks-recommender/cargo.toml

```
/// download ratings and metadata both.
fn download_data(ratings_path: &path,
                 books_path: &path) {
  let ratings_url = "https://github.com/zygmuntz/\
                     goodbooks-10k/raw/master/ratings.csv";
  let books_url = "https://github.com/zygmuntz/\
                   goodbooks-10k/raw/master/books.csv";

  download(&ratings_url, ratings_path)
    .expect("could not download ratings");
  download(&books_url, books_path)
    .expect("could not download metadata");
```

```
}
```

0.3.3 Data

Now we can run this function and we will see the files downloaded in the root folder. Take a look at these two csv's. The columns in books.csv file are book_id, goodreads_book_id, best_book_id, work_id, books_count, isbn, isbn13, authors, original_publication_year , original_title , title, language_code, average_rating, ratings_count, work_ratings_count, work_text_reviews_count, ratings_1, ratings_2, ratings_3, ratings_4, ratings_5, image_url, small_image_url thus having 23 columns. Similarly the ratings file has user_id, book_id and rating.

From this ratings file we will create the mapping between user_id and book_id and from the book file we will create the mapping between the book_id and the title . This can be encoded in a struct. We are only taking a small number of features for the sake of simplicity but you should try out with more features.

Listing 19: chapter7/goodbooks-recommender/src/main.rs

```
#[derive(Debug, Serialize, Deserialize)]
struct UserReadsBook {
  user_id: usize,
  book_id: usize,
}

#[derive(Debug, Deserialize, Serialize)]
struct Book {
  book_id: usize,
  title: String
}
```

We can now write two functions, one for deserializing the ratings and one for deserializing the books.

Listing 20: chapter7/goodbooks-recommender/src/main.rs

```
use csv;

fn deserialize_ratings(path: &Path)
        -> Result<Vec<UserReadsBook>, failure::Error> {
  let mut reader = csv::Reader::from_path(path)?;
  let entries = reader.deserialize()
    .collect::<Result<Vec<_>, _>>()?;

  Ok(entries)
}

fn deserialize_books(path: &Path)
   -> Result<(HashMap<usize, String>,
              HashMap<String, usize>), failure::Error> {
  let mut reader = csv::Reader::from_path(path)?;
```

```
    let entries: Vec<Book> = reader.deserialize::<Book>()
      .collect::<Result<Vec<_>, _>>()?;

    let id_to_title: HashMap<usize, String> = entries
      .iter()
      .map(|book| (book.book_id, book.title.clone()))
      .collect();
    let title_to_id: HashMap<String, usize> = entries
      .iter()
      .map(|book| (book.title.clone(), book.book_id))
      .collect();

    Ok((id_to_title, title_to_id))
}
```

In both the above functions we will read the files using csv and then collect the vectors. In the deserialize_ratings instead of directly returning the vectors, we will return the result of the vectors given by Result<Vec<UserReadsBook>>. In the deserialize_books function, additionally we create id_to_title and title_to_id mappings so that we are able to get one from the other through a simple lookup on the relevant mapping.

0.3.4 Model Building

The data part being done we can compose functions for building the models. As stated above the sbr implements two models that we can use - an LSTM based model and a moving average based model (EWMA). We will go ahead and use the EWMA model. The EWMA model is simpler in terms of computational weight and we will use this here. An exponentially weighted moving average is a type of control used to monitor small shifts in the process mean. It weights observations in the geometrically decreasing order so that the most recent observations carry the most weight and the older observations contribute very little to the model. In many cases this is enough.

First we write a function that sets up the hyperparameters of the model.

Listing 21: chapter7/goodbooks-recommender/src/main.rs

```
use sbr::models::ewma::{Hyperparameters, ImplicitEWMAModel};
use sbr::models::{Loss, Optimizer};

fn build_model(num_items: usize) -> ImplicitEWMAModel {
  let hyperparameters = Hyperparameters::new(num_items, 128)
    .embedding_dim(32)
    .learning_rate(0.16)
    .l2_penalty(0.0004)
    .loss(Loss::WARP)
    .optimizer(Optimizer::Adagrad)
    .num_epochs(10)
    .num_threads(1);
```

```
  hyperparameters.build()
}
```

For the model to work, we need the data converted to sbr Interactions objects. Interactions take in number of users and number of items and the timestamp. Since the ids are ordinal, hence we can take the greatest ids to be the number of users and items and have the id as the timestamp assuming that the data is ordered chronologically. In other situations we will need to take care of these in a different manner.

Listing 22: chapter7/goodbooks-recommender/src/main.rs

```
fn build_interactions(data: &[UserReadsBook]) -> Interactions {
  let num_users = data
    .iter()
    .map(|x| x.user_id)
    .max()
    .unwrap() + 1;
  let num_items = data
    .iter()
    .map(|x| x.book_id)
    .max()
    .unwrap() + 1;
  let mut interactions = Interactions::new(num_users,
                                           num_items);
  for (idx, datum) in data.iter().enumerate() {
    interactions.push(
      Interaction::new(datum.user_id,
                       datum.book_id,
                       idx)
    );
  }
  interactions
}
```

Now that the model has been built we will need to train on the data. We will now go ahead and create a fit function. As with other training done before the data needs to be split into test and train so that the fitness of the model can be determined and we can understand that the model is actually learning.

Listing 23: chapter7/goodbooks-recommender/src/main.rs

```
fn fit(model: &mut ImplicitEWMAModel,
       data: &Interactions)
       -> Result<f32, failure::Error> {

  let (train, test) = user_based_split(
        data, &mut rng, 0.2);
  model.fit(&train.to_compressed())?;
```

```
  let mrr = mrr_score(model, &test.to_compressed())?;
  Ok(mrr)
}
```

In the above fit function the result is the mrr score. The MRR score or the Mean reciprocal score is the score when the validity of the single highest ranking item is measured. Unfortunately the other popular means of scoring which is the mean average precision is not implemented in sbr and would need to be implemented by the user.

Once the model training is done, we will need to serialize the model, so that we can save the model in a file. We can use the serde library to do this.

Listing 24: chapter7/goodbooks-recommender/src/main.rs

```
fn serialize_model(model: &ImplicitEWMAModel,
                   path: &Path) -> Result<(), failure::Error> {
  let file = File::create(path)?;
  let mut writer = BufWriter::new(file);

  Ok(serde_json::to_writer(&mut writer, model)?)
}
```

We will now need a function execute the above functions one after the other.

Listing 25: chapter7/goodbooks-recommender/src/main.rs

```
fn main_build() {
  let ratings_path = Path::new("ratings.csv");
  let books_path = Path::new("books.csv");
  let model_path = Path::new("model.json");

  // Exit early if we already have a model.
  if model_path.exists() {
      println!("Model already fitted.");
      return ();
  }

  download_data(ratings_path, books_path);

  let ratings = deserialize_ratings(ratings_path).unwrap();
  let (id_to_title,
       title_to_id) = deserialize_books(books_path).unwrap();

  println!("Deserialized {} ratings.", ratings.len());
  println!("Deserialized {} books.", id_to_title.len());

  let interactions = build_interactions(&ratings);
  let mut model = build_model(interactions.num_items());

  println!("Fitting...");
  let mrr = fit(&mut model, &interactions)
      .expect("Unable to fit model.");
```

```
  println!("Fit model with MRR of {:.2}", mrr);

  serialize_model(&model, &model_path)
      .expect("Unable to serialize model.");
}
```

We should now plugin this main_build function in the main function.

Listing 26: chapter7/goodbooks-recommender/src/main.rs

```
fn main() {
  let opt = Opt::from_args();
  match opt {
    Opt::Fit => main_build(),
    _ => {
      unimplemented!();
    },
  }
}
```

Running it now with cargo run -- fit should save the model in the model.json file.

```
$ ls
books.csv  Cargo.lock  Cargo.toml  fit  model.json  ratings.csv
src  target
```

0.3.5 Model Prediction

To be able to do predictions we need to be able to do two things. First is model deserialization. After model deserialization, the model will be in ImplicitEWMAModel struct.

Listing 27: chapter7/goodbooks-recommender/src/main.rs

```
use std::io::BufReader;

fn deserialize_model() -> Result<ImplicitEWMAModel,
                                 failure::Error> {
  let file = File::open("model.json")?;
  let reader = BufReader::new(file);

  let model = serde_json::from_reader(reader)?;

  Ok(model)
}
```

After model deserialization is done, we will use the model to make predictions. So the predict function need to be passed the model for inference. Also the target tokens needs to be passed.

Listing 28: chapter7/goodbooks-recommender/src/main.rs

```
fn predict(input_titles: &[String],
           model: &ImplicitEWMAModel)
           -> Result<Vec<String>, failure::Error> {
  let (id_to_title,
       title_to_id) = deserialize_books(
      &Path::new("books.csv")
  ).unwrap();

  // Let's first check if the inputs are valid.
  for title in input_titles {
    if !title_to_id.contains_key(title) {
      println!("No such title, ignoring: {}", title);
    }
  }
  // rem code..

}
```

For this we will need to create the user representation and the possible indices that can be predicted.

Since the model is trained on the title ids we will provide the ids as input for prediction. We will also need the possible ids that can be predicted. In this case the ids are ordered and hence we can just take from 0 to the length. In other cases we would have needed to collect the actual vector.

Listing 29: chapter7/goodbooks-recommender/src/main.rs

```
fn predict(input_titles: &[String],
           model: &ImplicitEWMAModel)
           -> Result<Vec<String>, failure::Error> {
  // prev code..

  // Map the titles to indices.
  let input_indices: Vec<_> = input_titles
    .iter()
    .filter_map(|title| title_to_id.get(title))
    .cloned()
    .collect();
  let indices_to_score: Vec<usize> =
     (0..id_to_title.len()).collect();

  // rem code..
}
```

Based on the input_indices vector we can get the user representation which will be passed to the model for predictions.

Listing 30: chapter7/goodbooks-recommender/src/main.rs

```
fn predict(input_titles: &[String],
```

```
            model: &ImplicitEWMAModel)
            -> Result<Vec<String>, failure::Error> {
  // prev code ..

  // Get the user representation.
  let user = model.user_representation(&input_indices)?;
  // Get the actual predictions.
  let predictions = model.predict(&user, &indices_to_score)?;

  // rem code ..
}
```

Once the predictions are generated, we will need to sort based on the score. Here we will show only the top 10 results.

Listing 31: chapter7/goodbooks-recommender/src/main.rs

```
fn predict(input_titles: &[String],
            model: &ImplicitEWMAModel)
            -> Result<Vec<String>, failure::Error> {
    // prev code ...

    let mut predictions: Vec<_>
        = indices_to_score.iter()
        .zip(predictions)
        .map(|(idx, score)| (idx, score))
        .collect();

    predictions
        .sort_by(|(_, score_a), (_, score_b)|
                 score_b.partial_cmp(score_a)
                 .unwrap());

    Ok((&predictions[..10])
       .iter()
       .map(|(idx, _)| id_to_title.get(idx).unwrap())
       .cloned()
       .collect())
}
```

Plugging it in the main method should yield the results.

Listing 32: chapter7/goodbooks-recommender/src/main.rs

```
fn main() {
  let opt = Opt::from_args();
  match opt {
    Opt::Fit => main_build(),
    Predict(book) => {
      let model = deserialize_model()
        .expect("Unable to deserialize model.");
      let tokens: Vec<String> = book.text.split(",").map(
```

```
            |s| s.to_string()).collect();
        let predictions = predict(&tokens, &model)
            .expect("Unable to get predictions");
        println!("Predictions:");
        for prediction in predictions {
            println!("{}", prediction);
        }
    },
  }
}
```

Using the below command we can predict on user input titles.

Listing 33: chapter7/goodbooks-recommender/src/main.rs

```
$ cargo run -- predict --text "The Alchemist"
    Finished dev [unoptimized + debuginfo] target(s) in 0.22s
     Running `target/debug/goodbooks-recommender predict --text 'The Alchemist'`
Predictions:
The Alchemist
The Kite Runner
One Hundred Years of Solitude
The Da Vinci Code (Robert Langdon, #2)
A Thousand Splendid Suns
Life of Pi
Eat, Pray, Love
Memoirs of a Geisha
Angels & Demons  (Robert Langdon, #1)
The Five People You Meet in Heaven
```

0.4 Conclusion

This chapter introduced you to different interesting domains in high performance computing with Rust that are generally non-traditional applications of machine learning. The chapter starts with statistical analysis and how statistical analysis becomes trivial once we are able to convert data sets to ndarray matrices. In the next section we will wrappers of SIMD and BLAS to perform high performance and parallel computation on vectors. Finally we built a recommendation system using a popular recommendation crate in Rust.

In the last chapter of the book, you will learn about how we can create and deploy Rust ML applications for production using cloud and using Rust applications for the mobile and other applications.