Draft. Please do not include this page.

Single chapter compiled via LaTeX.

Shing Lyu, July 30, 2022. Git commit: e433481

Notes for the production team

Whitespace marker

You will see pink underline markers like these: `a␣b␣c`. One marker indicates one whitespace. So there is exactly one whitespace between "a" and "b" and one whitespace between "b" and "c". You'll also see them in code blocks like this:

```
#[derive(StructOpt)]
struct␣Options␣{
␣␣␣␣//␣...
␣␣␣␣#[structopt(short␣=␣"f",␣long␣=␣"file",␣parse(from_os_str))]
␣␣␣␣///␣Load␣the␣cat␣picture␣from␣the␣specified␣file
␣␣␣␣catfile:␣Option<std::path::PathBuf>,
}
```

These markers are only for you to see how many whitespaces there are. Please keep the exact number of whitespaces for inline code and code blocks. **Please do NOT include the markers during layout**.

The layout output should look like this: `a b c`

```
#[derive(StructOpt)]
struct Options {
    // ...
    #[structopt(short = "f", long = "file", parse(from_os_str))]
    /// Load the cat picture from the specified file
    catfile: Option<std::path::PathBuf>,
}
```

Auto line wrapping marker

You'll also see arrows near the end of the line and beginning of the second line:

```
This is a very very very very very very very very very very ↵
    ↳very very very long line that was wrapped automatically.
```

This indicates the original line was too long and it was automatically wrapped. **Please include these arrows.**

CHAPTER 9

Artificial intelligence and machine learning

9.1 What is machine learning?

Artificial Intelligence and machine learning have always been a hot topic in news and science fiction. Recently it got more media attention because of technology breakthroughs in deep learning and more consumer-facing applications on the market. The term *Machine learning* and *Artificial intelligence* are sometimes used interchangeably, but there is a subtle difference. Artificial intelligence focuses on "intelligence". An AI system tries to behave as if it possesses human intelligence, no matter what the underlying method or algorithm is. But in machine learning, the focus is on "learning", where the system is trying to learn something from the data without a human explicitly program the knowledge. For example, one of the early successes in AI is the expert system. In an expert system, the knowledge of a particular field is written down as rules and programmed directly into the code, so the system can answer questions or perform tasks as if it's a domain expert. This kind of system might appear to have some level of human intelligence, but underneath it's not actually "learning" from data. So this system can be called an AI system but not a machine learning system.

Researchers tried many different strategies to build AI systems, not necessarily machine learning. But machine learning gains its popularity due to a few technical advancements. First, the computing power of modern CPUs and GPUs grows exponentially because of innovations in hardware technology. This means that machine learning models can finally be trained in a reasonable time. The rise of the internet also means more and more data can be collected at very low cost, so you finally have enough data to power machine learning algorithms that require a large amount of data, like deep neural networks. All these factors contribute to the boom of machine learning in the recent decade.

Supervised vs. Unsupervised learning

There are two main branches of machine learning: supervised versus unsupervised learning. In a supervised learning setting, you give the algorithm a fully-labeled training dataset. For example, if you are trying to distinguish cat pictures from dog pictures, you need to prepare a large number of photos with the label "cat" or "dog". Because the algorithm can check its prediction with the label (or sometimes referred to as "ground-truth"), the algorithm can learn from its error and improve its predictions.

But a fully-labeled dataset is not always available. Unless there is an automated way of collecting the label with 100% accuracy, you have to fall back to label them manually. This takes tremendous time and money. Even if the data can be collected automatically, the data quality might not be ideal because of noise. Therefore, when getting a high-quality fully-labeled dataset is not possible, you can only rely on an unsupervised learning algorithm to do the job. An unsupervised algorithm takes a training dataset without labels, and try to learn the patterns from the data itself. For example, if you wish to distinguish between flower species, you can let the algorithm group the flowers by their color, shape, leaf shape, etc. But without ground-truth labels to check with, one algorithm might focus on the color and put a white rose into the same category as a white lily. Another algorithm might focus on the shape and categorize roses of all colors into one group. So unsupervised learning usually performs poorer or less predictable than a supervised model, but it's still very useful when labeled training data is hard to come by.

There are other categories of machine learning, like *semi-supervised learning*, which use a partially-labeled dataset to get high accuracy and low dataset preparation cost. There is also *reinforced learning*, which takes feedback from the environment to correct future behavior. For example, a maze-navigating robot can get a reward every time it successfully reached the end of a maze. It can learn the way to navigate a maze by seeking maximum reward and avoid potential penalties. In this chapter, you'll be focusing on supervised learning and unsupervised learning.

9.2 What are you building?

You'll use two examples to illustrate supervised and unsupervised learning. First, you start with unsupervised learning. The code for unsupervised learning is simpler then supervised learning so that you can get a taste of a machine learning program without being overwhelmed by details. You'll be building a model that can distinguish different cat breeds. You collected body size measurements from three cat breeds: Persian, British Shorthair, Ragdoll (Figure 9.1-9.3). Since these three breeds have a slightly different body height and length, you'll run a K-means clustering model (explained in Section 9.4) on these two features. The model, once trained, can be used to cluster cats into different breeds automatically. Because K-means can only see the similarity between the data points, it can group the cats into groups. It can't tell exactly which group is which cat breed.

Figure 9.1: Persian cat

Figure 9.2: British shorthair cat

Figure 9.3: Ragdoll cat

The second example is for supervised learning. Similar to the previous model, you have a handful of body measurements, but this time from both cats and dogs. You have labels that indicate if each body measurement is taken from a cat or a dog. Using this dataset, you'll use a neural network model to learn how to tell a cat from a dog. In machine learning, this kind of job is called a classification problem. When the model is trained, it can predict if a given body measurement belongs to a cat or dog, even if the data is not part of the training set. For simplicity's sake, you'll use only the height and body length as the input.

These machine learning models don't exist in a vacuum. Besides training the model and predicting using the model, there are many tasks involved, like data preparation, cleaning, and visualization. You'll also be showing how to use Rust to generate artificial training data, writing and reading the data as CSV (comma-separated values) files, and visualizing the data. The demo programs will consist of a few loosely-related binary executables. They will not communicate directly but through passing CSV files. This way, you can minimize the dependency between the steps.

9.3 Introducing `rusty-machine`

The machine learning ecosystem in any programming language relies on a strong foundation. Building machine learning libraries involves not only the machine learning algorithm itself, but also many fundamental operations like numerical computing, linear algebra, statistics, and data manipulation.

In this chapter, you are going to use the `rusty-machine` crate. The `rusty-machine` crate contains many traditional machine learning algorithms implemented in Rust. Although deep learning is the hottest topic in machine learning now, there are no mature Rust-based libraries yet. Most of the deep learning libraries in Rust now are bindings to libraries in other languages, so the API design is not very Rusty. Deep learning models are also harder to understand by intuition because they involve more advanced mathematical theories, which might distract us from the code architecture and Rust API themselves.

Some of the machine learning algorithms `rusty-machine` contains are:

- Linear Regression
- Logistic Regression
- Generalized Linear Models
- K-Means Clustering*
- Neural Networks*
- Gaussian Process Regression
- Support Vector Machines

- Gaussian Mixture Models
- Naive Bayes Classifiers
- DBSCAN
- K-Nearest Neighbor Classifiers
- Principal Component Analysis

It uses the `rulinalg` crate for linear algebra. Part of the `rulinalg` are re-exported in the `rusty_machine::linalg` namespace, so you don't need to import `rulinalg` manually. It also contains useful data transformation tools for data pre-processing, which you'll use later for normalization.

9.4 Clustering cat breeds with K-means

Introduction to the K-means algorithm

You don't need to be a cat expert to identify different cat breeds. A Persian cat looks different from a British Shorthair in many aspects: their coat has different length, their face looks different, their average size is also different. Categorizing things is a human nature that helps us make sense of this world. But machines don't have such instinct, so you need to program the power of mathematics into them. This kind of problem is called the clustering problem, and a popular algorithm to solve this problem is *K-means*.

Since this is a book for general Rust enthusiasts, not mathematicians, you are going to explain the concept in plain English. You can easily find the formal mathematical definitions by searching "K-means" online.

The concept of K-means is simple. To cluster the points into k groups, you want to split them in a way that the points in a group are close to each other, but far from points in other groups. The exact steps to do this are:

1. Randomly assigns k points as the "centroids". The centroids are the center point of each cluster.
2. Assignment: For all the other points, you assign them to the group of the nearest centroids.
3. Update the centroids: For each group, you find the center point (i.e., the mean) of all the points in the group and use this center point as the new centroid.
4. Repeat steps 2 to 3 until the centroids no longer move.

As you can imagine, during each update, the centroids will move towards the center of the points "cloud", and during the next assignment, some points might be assigned to a new centroid because the centroid position changed. You continue this process until the centroids no longer move; this is when you say the model converges. You can see a graphical example in Figure 9.4.

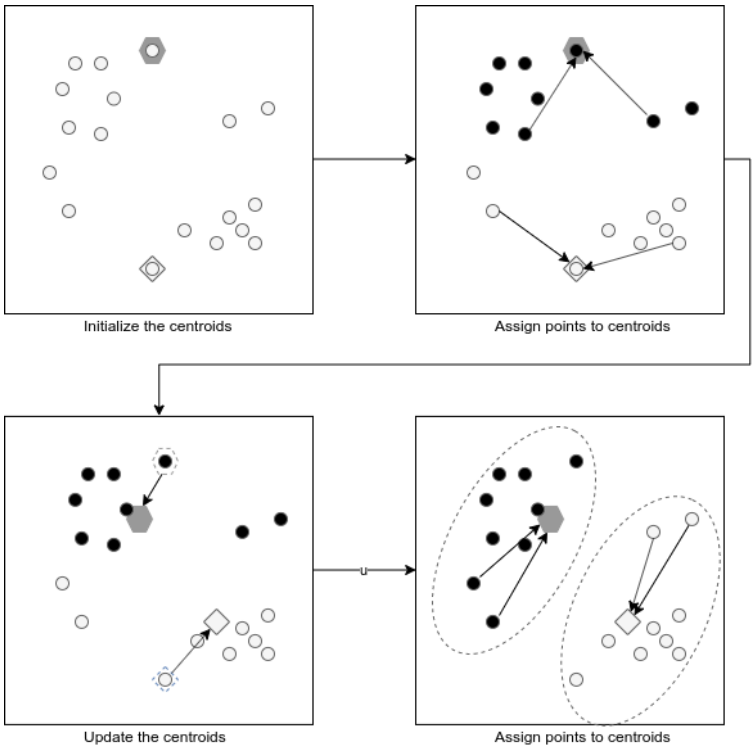

Figure 9.4: An example of the K-means algorithm

In practice, the initial location of the centroids matters a lot to the final result. If you assign the initial centroids badly, the algorithm might converge to a result that is not ideal[1]. It might also take longer for the model to converge. You can use an algorithm called K-means++ to initialize the initial centroids better. The intuition behind it is that you want to spread out the initial centroids as far as possible. The exact step to do this is as follows:

1. Choose the first centroid randomly from all the points.
2. For each point x, calculate the distance to its nearest existing centroid as $D(x)$
3. To find the next centroid, pick a point with a probability proportional to $D(x)^2$. What this means is that if a point x is further away from any existing centroids, it's $D(x)$ is larger, and it has a higher probability to be chosen as a new centroid
4. Repeat step 2 to 3 until all the centroids are picked

By using the K-means++ algorithm, the initial centroids are spread out as far away from each other as possible. This will usually result in a better result. This is the default initialization method used by `rusty-machine`.

The training data

To prepare the training data, you need to collect body measurements from many cats. Since this is time-consuming, and you don't really want to get scratches and bites from grumpy cats, you are going to create a fake data set using the average. You can generate some artificial cat body measurements with a normal distribution centered around the average. The body measurements you are using are:

- height: the height from the ground to a cat's shoulder
- length: the length from a cat's head to its bottom, excluding the tail.

The average measurements of the three breeds are:

- Persian: height 22.5 cm, length 40.5 cm
- British Shorthair: height 38.0 cm, length 50.0 cm
- Ragdoll: height 25.5 cm, length 48.0 cm

To have a fair amount of data points for demonstration, you generate 2000 samples per cat breed. These data points are generated using a normal distribution around the average with an arbitrarily chosen standard deviation of 1.8 cm. This standard deviation creates a

[1] In mathematics, you say it converges to a local optimum, rather than the global optimum.

nice cloud of data points with a little overlap, which will help us illustrate how K-means works.

To generate this training data, you are going to set up `rusty-machine` so that you can use the linear algebra part. First, let's create a Rust project with `cargo new cat-breeds-k-means`. In the `Cargo.toml` file, you need to add `rusty-machine` and `rand` (for random number generation) and `rand_distr` (for common probability distributions like normal distribution).

TODO: Add filename

```
[dependencies]
rusty-machine = "0.5.4"
rand = "0.7.0"
rand_distr = "0.2.1"
```

Because you are going to have multiple binaries in this project, you can't simply have a `src/main.rs` and run it with `cargo run`. Cargo has support for multiple binaries in a project. Simply add the files to `src/bin/`, for example `src/bin/generate.rs`. Then you can run it with `cargo run --bin generate`.

In the `src/bin/generate.rs` file, you can start by declaring the crates you are using:

```
extern crate rusty_machine;
extern crate rand;
```

You want to build a function that can generate training data around the given average with the normal distribution. Let's first define its function signature in Listing 9.4.

TODO: Add filename

```
fn generate_data(centroids: &Matrix<f64>,
                 points_per_centroid: usize,
                 noise: f64)
                 -> Matrix<f64> { ... }
```

.

As you can see in Listing 9.4, the function takes three parameters:

- A `Matrix`[2] of `f64` values. Each row is the average height and length of a cat breed. It should have a shape of (3×2) (3 cat breeds × 2 dimensions (height and length) per breed)
- The number of data points you want to generate for each cat breed.
- The standard deviation used for the normal distribution, a.k.a. the noise.

The return value will be a `Matrix` containing all the generated data points. It should have the shape (total number of samples × 2).

[2] `Matrix` is provided by `rulinalg` crate wrapped in `rusty-machine`.

The logic is pretty simple; you have nested `for` loops that iterate through the three centroids and generate the required number of samples for each centroid. This is shown in Listing 9.4.

TODO: Add filename

```
extern crate rusty_machine;
extern crate rand;

use rusty_machine::linalg::{Matrix, BaseMatrix};

use rand::thread_rng;
use rand::distributions::Distribution; // for using .sample()
use rand_distr::Normal; // split from rand since 0.7
use std::io;
use std::vec::Vec;

fn generate_data(centroids: &Matrix<f64>,
                 points_per_centroid: usize,
                 noise: f64)
                 -> Matrix<f64> {
    assert!(centroids.cols() > 0, "centroids cannot be empty.");
    assert!(centroids.rows() > 0, "centroids cannot be empty.");
    assert!(noise >= 0f64, "noise must be non-negative.");
    let mut raw_cluster_data =
        Vec::with_capacity(centroids.rows() *
    ↪ points_per_centroid * centroids.cols());

    let mut rng = thread_rng(); // [1]
    let normal_rv = Normal::new(0f64, noise).unwrap(); // [2]

    for _ in 0..points_per_centroid { // [3]
        // generate points from each centroid
        for centroid in centroids.iter_rows() {
            // generate a point randomly around the centroid
            let mut point = Vec::with_capacity(centroids.cols())
    ↪ ;
            for feature in centroid.iter() {
                point.push(feature + normal_rv.sample(&mut rng))
    ↪ ;
            }

            // push point to raw_cluster_data
            raw_cluster_data.extend(point);
        }
    }
```

```
    Matrix::new(centroids.rows() * points_per_centroid,
                centroids.cols(),
                raw_cluster_data)
}
```

When you first enter the function, you validate that the centroids matrix is not empty, and the standard deviation is non-negative. Then you initialize a `raw_cluster_data` vector with the expected capacity. You allocate the memory for the `raw_cluster_data` in advance, so the vector does not need to resize when it grows. In [1], you create the random number generator from the `rand` crate. Since you want to generate random numbers with a normal distribution, you need to initialize a normal distribution from the `rand_distr::Normal` ([2]). This normal distribution has a mean (average) of `0` and standard deviation of `noise`. You can then add this random deviation to the average cat body measurements.

Then comes the actual generation of samples. The outer `for` loop ([3]) makes sure you repeat the generation n times, where n is the number of desired samples per centroid. In the inner loop, you iterate through the centroids, so you generate a sample for each. This ensures that you generate a total number of `(number of centroids × number of sample per centroid)` samples. In the case, you have 3 × 2000 = 6000 samples.

In the body of the loop, you create a temporary vector of size two to hold the height and length. For each dimension, you get a random number from the normal distribution you just initialized. This random number is generated around 0. Then you add the random number to the average height or length, so you'll have samples that follow a normal distribution around the average height or length of the cat breed. Finally, this `point` is added to the `raw_cluster_data` vector. The vector will then become a large 1-D array. If you use the symbol A_h to denote the height of cat A, and A_l for the its length, the `raw_cluster_data` vector for cat A, B, and C will look like:

$$[A_h, A_l, B_h, B_l, C_h, C_l]$$

But what you actually want is a matrix of samples, one sample per row:

$$\begin{bmatrix} A_h & A_l \\ B_h & B_l \\ C_h & C_l \end{bmatrix}$$

To convert this, you pass the 1-D array to the `matrix::new()` function. You can provide the desired shape (number of rows and columns), and `matrix::new()` will reshape the 1-D array into the matrix you want. Finally, you return this matrix as the training data.

Exporting as CSV

In the `main()` function of the `src/bin/generate.rs`, you are going to call the `generate_data()` function and output the data to STDOUT in the CSV format. CSV(comma-separated value) is a simple format for tabular data: rows are separated into lines, and the columns are separated by commas. This format is supported in most programming languages and spreadsheet software (e.g., LibreOffice Calc or Microsoft Excel).

Although CSV format is quite simple, it's still too error-prone to format it without the help of a library. You choose to use the `csv` crate to save us all the trouble. Simply add the line `csv = "1.1"` to the `dependencies` section of the `Cargo.toml`. Then in the `main()` function, you can put everything together as in Listing 9.4.

TODO: Add filename

```
// settings
const CENTROIDS:[f64;6] = [
    //Height, length
    22.5, 40.5, // persian
    38.0, 50.0, // British shorthair
    25.5, 48.0, // Ragdoll
];
const NOISE:f64 = 1.8;
const SAMPLES_PER_CENTROID: usize = 2000;

fn generate_data(...) { ... }

fn main() -> Result<(), std::io::Error> {
    let centroids = Matrix::new(3, 2, CENTROIDS.to_vec());

    let samples = generate_data(&centroids, SAMPLES_PER_CENTROID
    ↪ , NOISE);

    let mut writer = csv::Writer::from_writer(io::stdout());
    writer.write_record(&["height", "length"])?;
    for sample in samples.iter_rows() {
        writer.serialize(sample)?;
    }
    Ok(())
}
```

First, you convert the centroids from the `const CENTROIDS` vector into an array. Ideally, all these parameters should be configurable from command-line arguments, but you'll leave that for the next section. For now, you'll just hard-code the configurations as `const`s at the beginning of the file.

Then you call the `generate_data()` function and assign the generated training data to the variable `samples`. To write these samples into CSV format, you need to

initialize a `csv::Writer`. For simplicity, you are going to write the CSV directly to STDOUT. You'll see more advanced usage of writing directly to files in Section 9.5. You initialize the `Writer` by:

```
let mut writer = csv::Writer::from_writer(io::stdout());
```

To write a simple plain-text line to the CSV output, you can use the `writer.write_record()`. It takes (a reference to) an array of strings. So you write the header "height" and "length" to it. You can also provide a serializable (i.e., implements `serde::Serialize`) struct and let it be automatically converted to a valid CSV line. This can be done by `writer.serialize()`. So you iterate through the rows of the sample array and write each line as CSV by:

```
for sample in samples.iter_rows() {
    writer.serialize(sample)?;
}
```

Now, if you run `cargo run --bin generate`, you'll see 6001 lines (including a heading line) being printed to the screen. You can easily pipe it to a file by `cargo run --bin generate > training_data.csv`.

Moving the configuration into a file

In the previous section, you hard-coded all the configurations as `const`s in the source code. This becomes an impendence when you want to experiment with many different configurations. Building a machine learning application involves a lot of experimentation. You usually need to try many different parameters and settings the get the best result. If you hard-code the parameters in the code, you'll need to change then and re-compile the program every time. It's easier to put those configurations in a configuration file, then specify the configuration file using command-line arguments. This way, you can easily choose a configuration at runtime. Another benefit is that you can keep all the configuration files in the source code repository so that you can reproduce a specific experiment quickly.

There are many machine-readable configuration file formats to choose from. For example, TOML, JSON, YAML, XML, and RON[3]. You choose TOML because Cargo uses it, so it's widely accepted by the Rust community. Also, it has excellent parser and deserialization support in Rust.

You can move the `const`s into a file named `config/generate.toml` (Listing 9.4). You'll notice that the syntax is slightly different from Rust, but it's still straightforward.

TODO: Add filename

```
centroids = [ # Height, length
    22.5, 40.5, # persian
    38.0, 50.0, # British short hair
```

[3]The Amethyst framework you introduced in Chapter 7 uses RON as its configuration format.

```
    25.5, 48.0, # Ragdoll
]
noise = 1.8
samples_per_centroid = 2000
```

But the value types in TOML does not map one-to-one to Rust types. How do you make sure the values are parsed into Rust as a `[f64; 6]`, `f64` and `usize`? The `toml` crate, which is the TOML parser you are going to use, can work with `serde` to deserialize the TOML into a pre-defined Rust struct. You need to add the `toml` and `serde` crates into the `dependencies` section of the `Cargo.toml`:

```
[dependencies]
# ...
toml = "0.5.5"
serde = { version = "1.0.103", features = ["derive"] }
```

Then you can define the struct format for the TOML file in the `src/bin/generate.rs` file (Listing 9.4)

TODO: Add filename

```
use std::fs::read_to_string;
use serde::Deserialize;
// ...

#[derive(Deserialize)]
struct Config { // [1]
    centroids: [f64; 6],
    noise: f64,
    samples_per_centroid: usize,
}

fn main() -> Result<(), std::io::Error> {
    let toml_config_str = read_to_string("config/generate.toml")
    ↪ ?; // [2]
    let config: Config = toml::from_str(&toml_config_str)?;
    // ...
}
```

In [1], you derived the `serde::Deserialize` trait on the `Config` struct, this will give the `toml` parser enough information on how to parse the TOML file into the `Config` struct. Then in [2], you read the `config/generate.toml` file into a `String` and pass its reference to `toml::from_str`. Because you set the type of the variable `config` to be `Config`, `toml` will parse it into a `Config` struct using the `Deserialize` implementation. Once it's parsed, you can access the individual fields using the dot notation, e.g., `config.centroids`, `config.noise`.

You can then remove all the `const`s and use the `config` instead. For example,

```
let centroids = Matrix::new(3, 2, CENTROIDS.to_vec());
```

```
let samples = generate_data(&centroids, SAMPLES_PER_CENTROID,
    NOISE);
```

becomes

```
let centroids = Matrix::new(3, 2, config.centroids.to_vec());
let samples = generate_data(&centroids, config.
    samples_per_centroid, config.noise);
```

In Listing 9.4, you still hard-codes the path to the TOML file. You can use `StructOpt`, which you learned in Chapter 2, to make it a command-line parameter. You need to add the `structopt` dependency in `Cargo.toml`:

```
[dependencies]
// ...
structopt = "0.3.5"
```

Then in `src/bin/generate.rs`, you can add an argument called `--config-file` (Listing 9.4).

TODO: Add filename

```
// ...
use structopt::StructOpt;

// ...

#[derive(StructOpt)]
struct Options {
    #[structopt(short = "c", long = "config-file", parse(
    ↪ from_os_str))]
    /// Configuration file TOML
    config_file_path: std::path::PathBuf,
}

fn main() -> Result<(), std::io::Error> {
    let options = Options::from_args();
    let toml_config_str = read_to_string(options.
    ↪ config_file_path)?;
    // ...
}
```

In the `main()` function, you can read the configuration file path dynamically from `Options::from_args()` and pass it to `read_to_string()`. Then to run the generate script, you can use the following command in the shell:

```
cargo run --bin generate -- --config-file config/generate.toml
```

If you want to try different configurations, simply copy-paste the `config/generate.toml`, change some parameters in it and specify the new file name in

the `--config-file` argument. You no longer need to re-compile the `src/bin/generate.rs` script every time you change the configuration. This pattern is also beneficial when you do model training in the following section. One of the key processes in machine learning is parameter selection (or parameter tuning). This involves testing various parameters for the machine learning model to find the best setup. You can test with many different configuration files using this pattern. You can also easily recreate the model with the binary, configuration file, and training data.

Visualizing the data

Before you jump into learning, it would be nice to see how the data looks like. To visualize the training data, you can use some plotting library. Since there is no mature enough Rust plotting library at the moment, you'll fall back to the popular gnuplot library. Gnuplot is a command-line-driven plotting tool. It's mostly written in C but has a Rust binding.

You need to first install the `gnuplot` binary by running `sudo apt-get install gnuplot` in the terminal. Then, you need to add the `gnuplot` binding crate to the dependencies section of the `Cargo.toml`:

```
[dependencies]
gnuplot = "0.0.31"
```

Then let's create a new script in `src/bin` named `plot.rs`. The bare minimum code to draw something onto the screen is as simple as Listing 9.4.

TODO: Add filename

```
use std::error::Error;
use gnuplot::{AxesCommon, Caption, Figure};

fn main() -> Result<(), Box<dyn Error>>{
    let mut x: Vec<f64> = Vec::new();
    let mut y: Vec<f64> = Vec::new();

    //TODO: read the CSV data into x and y

    let mut fg = Figure::new();
    fg.axes2d().points(x, y, &[Caption("Cat")]);
    fg.show();
    Ok(())
}
```

In the `main()` function of Listing 9.4, you create a gnuplot `Figure`. You render a 2D graph using `fg.axes2d()`, and you draw all the data points onto the 2D canvas using `points()`. The `x` and `y` parameters are vectors holding the x-axis and y-axis coordinates of all the points. You also add a caption "Cat" for the points. To see the figure, you run `fg.show()`, which will open a gnuplot window containing the graph. The graph will look like Figure 9.5.

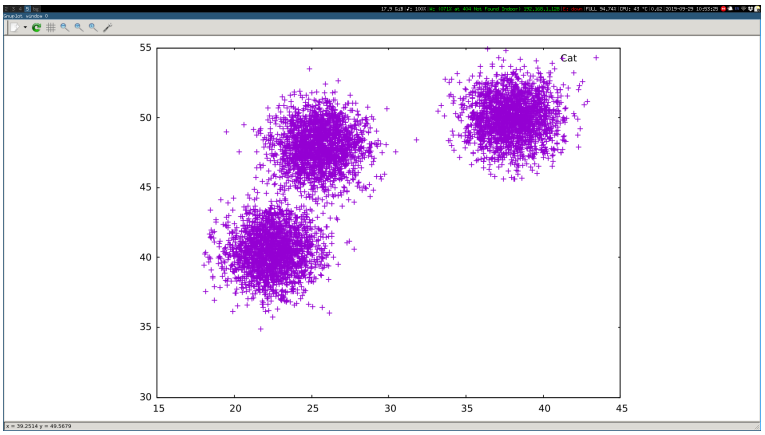

Figure 9.5: A minimal gnuplot

One thing you missed in the previous example is how to get the `x` and `y`. Since you piped the generated CSV to STDOUT, you can read the data using STDIN. This way, you can easily pipe the CSV generates from the previous section into this `plot.rs` script:

```
cat training_data.csv | cargo run --bin plot
```

The CSV reading code is very similar to the writing part, just reversing the write with reading (Listing 9.4).

TODO: Add filename

```
// ...
use std::io;

fn main() -> Result<(), Box<dyn Error>>{
    let mut x: Vec<f64> = Vec::new();
    let mut y: Vec<f64> = Vec::new();

    let mut reader = csv::Reader::from_reader(io::stdin());
    for result in reader.records() {
        let record = result?;
        x.push(record[0].parse().unwrap());
        y.push(record[1].parse().unwrap());
    }

    // Drawing the figure

    Ok(())
}
```

First, you create a `Reader` that reads from `io::stdin()`. Then you can easily iterate through the rows of the file by `for result in reader.records()`. However, the items yield by the iterator (i.e., the `result`) has the type of `Result<StringRecord, Error>`, so you need to get the `StringRecord` out of the `Result` with the `?` operator:

```
let record = result?;
```

You can access an individual column in a `StringRecord` using an index like `record[0]`, this yields a `str`. Because the `x` and `y` requires the type `Vec<f64>`, you can convert the `str` to `f64` by calling `.parse().unwrap()`. You push the parsed x and y coordinate values into the `x` and `y` vector.

To make the figure easier to understand, you can add titles, legend, and axes label to the graph as in Listing 9.4.

TODO: Add filename

```
fg.axes2d()
    .set_title("Cat body measurements", &[])
    .set_legend(Graph(0.9), Graph(0.1), &[], &[])
```

```
    .set_x_label("height (cm)", &[])
    .set_y_label("length (cm)", &[])
    .points(x, y, &[Caption("Cat")]);
```

Most of the function names are self-explanatory. Only the `set_legend` might look a little confusing. The first two parameters for `set_legend` is the x and y coordinate of the legend, you set it to (90% of the graph's x width, 10% of the graph's y height), which is the bottom-right corner of the graph. Most of the functions take some optional flags in an array. You are sticking with the defaults, so you pass empty arrays for the optional parameters. If you rerun the script, the generated figure will look like Figure 9.6.

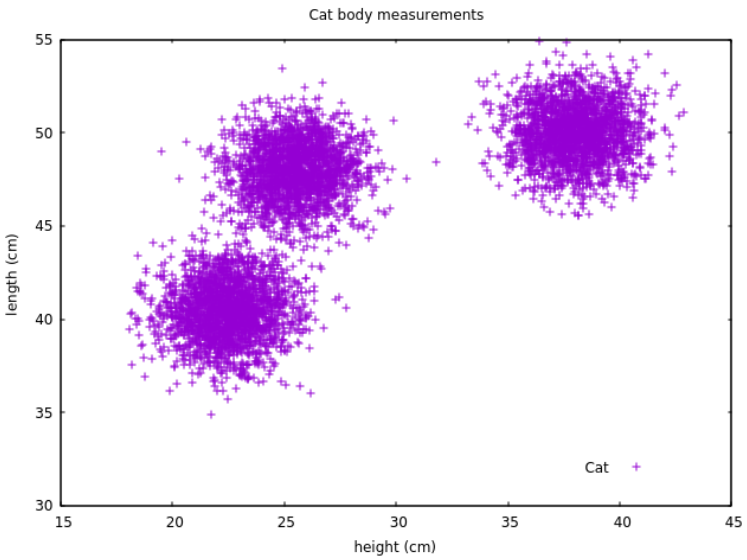

Figure 9.6: The plot with title, legend and axes labels

Setting up K-means

As you can see from Figure 9.6, the cat body measurements form three clusters. We'd expect the K-means algorithm to cluster them into three groups. The K-means model is located in `rusty_machine::learning::k_means::KMeansClassifier`. All of the unsupervised models, including K-means, implements the `rusty_machine::learning::UnSupModel` trait. This trait has a very simple interface, as shown in Listing 9.4.

TODO: Add filename

```
pub trait UnSupModel<T, U> {
```

```
    fn train(&mut self, inputs: &T) -> LearningResult<()>;
    fn predict(&self, inputs: &T) -> LearningResult<U>;
}
```

The `train()` function will take the training data `input` and learn from it. The "knowledge" is stored in the model itself. After the model is trained, you can use the `predict()` function to predict (in the case, cluster) new data based on the knowledge learned from the training data.

Before you can call `train()`, you need to configure the K in the name K-means. K is the number of clusters you expect it to cluster into. From Figure 9.6, you can clearly see there are three clusters. Therefore, you are going to set $k = 3$. As you mentioned in Section 9.2, you create a separate binary for the K-means training and clustering: `src/bin/cluster.rs`. You can start by writing a simple `main()` function as in Listing 9.4.

TODO: Add filename

```
extern crate rusty_machine;

use rusty_machine::learning::k_means::KMeansClassifier;
use rusty_machine::learning::UnSupModel;

const CLUSTER_COUNT: usize = 3;

fn main() {
    let samples = read_data_from_stdin().unwrap(); // will
    ↪ discuss later

    let mut model = KMeansClassifier::new(CLUSTER_COUNT);
    model.train(&samples).unwrap();

    let classes = model.predict(&samples).unwrap();

    export_result_to_stdout(samples, classes.into_vec()).unwrap
    ↪ ();
}
```

The steps in the `main()` function are pretty straightforward. You load the CSV data from STDIN using a helper function `read_data_from_stdin()`, which you'll discuss soon after. You initialize a `KMeansClassifier` with the configuration K = `CLUSTER_COUNT`. Then you do `model.train()` and `model.predict()` for the same data. The `model.train()` step will preform the clustering and store the centroids in the model, then `model.predict()` will return us the cluster ID label (i.e., the cat breed)for each data point. You then write the result to STDOUT using another helper function `export_result_to_stdout()`. That's all you need for training a complicated mathematical model in Rust!

The helper functions for reading and outputting data are similar to the ones you saw

in the generation and visualization script. The `read_data_from_stdin()` function (Listing 9.4) is almost the same as the `plot.rs` function (Listing 9.4), except that you convert the output to a `Matrix`. This is because the `KMeansClassifier.train()` expects a `Matrix`, while gnuplot expects a `Vec`.

TODO: Add filename

```
fn read_data_from_stdin() -> Result<Matrix<f64>, Box<dyn Error>>
    ↪ {
    let mut reader = csv::Reader::from_reader(io::stdin());
    let mut data: Vec<f64>= vec!();
    for result in reader.records() {
        let record = result?;
        data.push(record[0].parse().unwrap());
        data.push(record[1].parse().unwrap());
    }

    Ok(Matrix::new(&data.len() / 2, 2, data))
}
```

The output function `export_results_to_stdout()` is also some simple call to a `csv::Writer` (Listing 9.4). The key in this function is that you want to output the original 2-D body measurement data along with the `classes` data from the clustering result. Imagine you have three cats like:

$$\begin{bmatrix} 22.5 & 40.5 \\ 38.0 & 50.0 \\ 25.5 & 48.0 \end{bmatrix}$$

They are clustered into class 0, 1, and 2[4], respectively:

$$\begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix}$$

You want the output CSV to be the two matrixes "stitched" together:

$$\begin{bmatrix} 22.5 & 40.5 & 0 \\ 38.0 & 50.0 & 1 \\ 25.5 & 48.0 & 2 \end{bmatrix}$$

This is achieved by the line `samples.iter_rows().zip(classes)`. The `.zip()` function does exactly the stiching you want.

TODO: Add filename

[4]The class IDs are arbitrary integers; they are categorical, so the number doesn't convey any mathematical meaning.

```
fn export_result_to_stdout(samples: Matrix<f64>, classes: Vec<
    ↪ usize>) -> Result<(), Box<dyn Error>> {
    let mut writer = csv::Writer::from_writer(io::stdout());
    writer.write_record(&["height", "length", "class"])?;
    for sample in samples.iter_rows().zip(classes) {
        writer.serialize(sample)?;
    }
    Ok(())
}
```

You export the result to a CSV file for visualization:

```
cat training_data.csv | cargo run --bin cluster > results.csv
```

To make it clear which point belongs to which class, you can use different point symbols and colors for different classes. You can tweak the original visualization script into Listing 9.4.

TODO: Add filename

```
use std::error::Error;
use std::io;
use gnuplot::{Figure, Caption, Graph, Color, PointSymbol};
use gnuplot::AxesCommon;

fn main() -> Result<(), Box<dyn Error>>{
    let mut x: [Vec<f64>; 3] = [Vec::new(), Vec::new(), Vec::new
    ↪ ()];
    let mut y: [Vec<f64>; 3] = [Vec::new(), Vec::new(), Vec::new
    ↪ ()];

    let mut reader = csv::Reader::from_reader(io::stdin());
    for result in reader.records() {
        let record = result?;
        let class:usize = record[2].parse().unwrap();
        x[class].push(record[0].parse().unwrap());
        y[class].push(record[1].parse().unwrap());
    }

    let mut fg = Figure::new();
    fg.axes2d()
        .set_title("Cat breed classification result", &[])
        .set_legend(Graph(0.9), Graph(0.1), &[], &[])
        .set_x_label("height (cm)", &[])
        .set_y_label("length (cm)", &[])
        .points(
                    &x[0],
                    &y[0],
```

```
                        &[Caption("Cat breed 1"), Color("red"),
    ↪ PointSymbol('+')],
                )
                .points(
                        &x[1],
                        &y[1],
                        &[Caption("Cat breed 2"), Color("green"),
    ↪ PointSymbol('x')],
                )
                .points(
                        &x[2],
                        &y[2],
                        &[Caption("Cat breed 3"), Color("blue"),
    ↪ PointSymbol('o')],
                );
    fg.show();
    Ok(())
}
```

Most of the code is the same as in Listing 9.4. However, this time, the `x` and `y` is a nested array of three individual arrays. Each sub-array contains the x (or y) coordinates for a specific cluster. For example, `x[0]` contains the x coordinates of the cat cluster 0, and x[1] is for cluster 1 and x[2] for cluster 2.

To plot the points in different symbol and color, you use some optional parameters for the `text`:

- `Caption("Cat breed 1")` sets the caption for that cluster of points.
- `Color("red")` sets the point color to red.
- `PointSymbol("+")` make the point to be represented as a + symbol, so it's easier to distinguish if the graph is printed in black-and-white.

This gives as Figure 9.7.

If you zoom in to the border between the breed 2 (`x`) and breed 3 (`o`), you can see the points are almost split by a straight line (Figure 9.8). You know that a normal distribution will probably not look like this, some of the breed 2 points might fall in the breed 3 "cloud" and vice versa. But this is an inherited limited of K-means. Given only the height and length, but without a proper ground-truth tagging, the algorithm can only predict based on the nearest mean (i.e., centroid), resulting in a seemingly clear-cut line between the clusters.

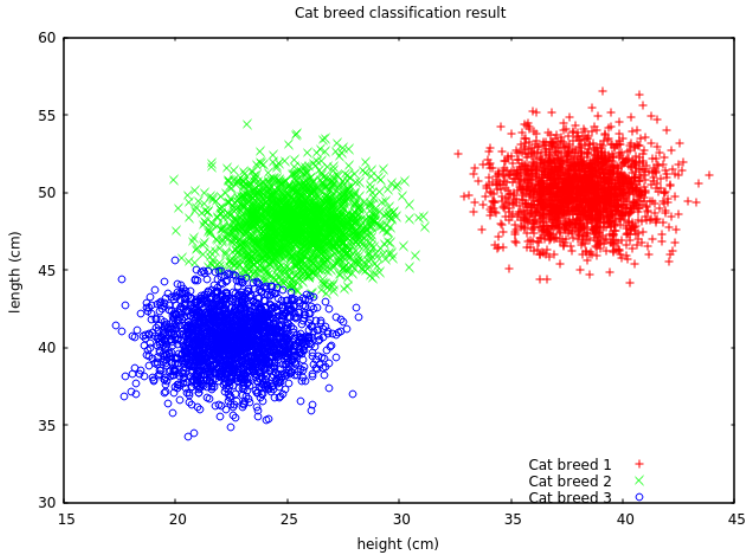

Figure 9.7: Clustering result for the K-means algorithm

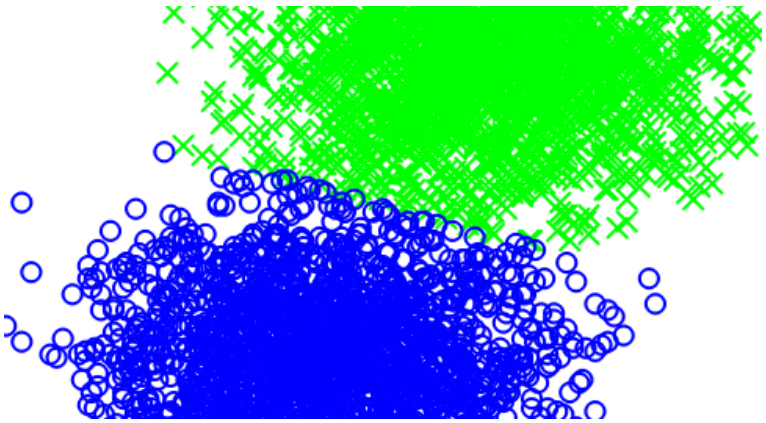

Figure 9.8: Clear cut between breed 2 and 3

9.5 Detecting cats versus dogs with the neural network

Introduction to neural networks

We've seen how unsupervised model work, now you can shift the attention to a supervised model. The supervised model you are going to introduce is the Artificial Neural Network (ANN) model or Neural Network for short. Neural Network draws its inspiration from how a human brain works. The human brain consists of neurons. Each neuron takes stimuli and decides if it should be "activated" or not. An activated neuron will send an electrical signal to other connected neurons. So if you have a big network of interconnected neurons, they can learn to react to different inputs by adjusting the way they connect, and how sensitive they are to the stimuli.

Modern neural network models maintain the same guiding principle, but it focuses more on solving empirical questions using data, rather than trying to model a human brain accurately. One of the key components of a neural network model is the neuron (or sometimes called a *node*; shown in Figure 9.9). A node consists of one or more inputs (x_i), their weights (w_i), an input function, and an activation function[5]. The input function takes the weighted sum of all the inputs and passes it to the activation function. The weights are adjusted during the learning process to amplify or dampen signals according to the significance of the input signal to the things you want to learn. The result of the input function will pass to the activation function to determine if the node should be activated or not.

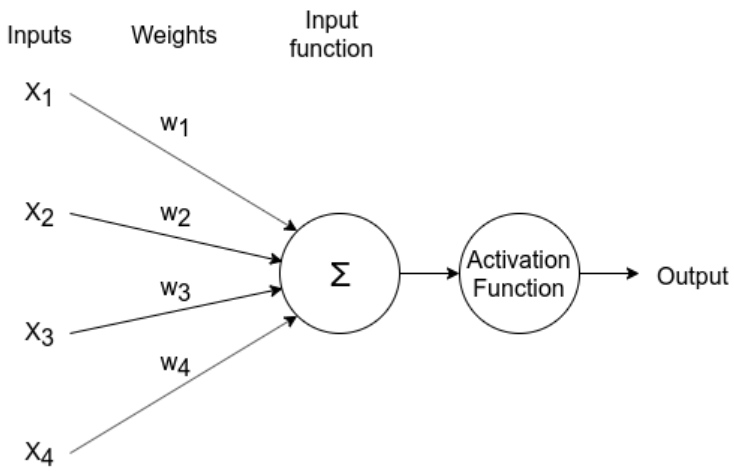

Figure 9.9: Structure of a neuron/node

[5]The `rusty-machine` uses the Sigmoid function by default.

The nodes need to be combined into a network. A simple example is Figure 9.10, which contains two input nodes, two nodes in the middle layer, and one output node. For the dog-or-cat example, you can send the height and length value to the two input nodes, and the output node should give us a signal to indicate if the input data belongs to a dog or cat. Each node will determine if it should be activated based on the input it got from the previous stage, combined by the weights.

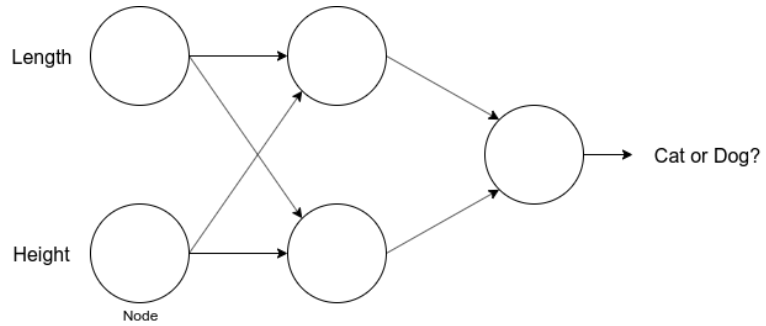

Figure 9.10: A simple neural network

In the beginning, you can randomly set the weights, but this won't give any better results than merely guessing. To learn from the training data, you need to adjust the weights in the nodes according to the training data. Because you have the ground-truth answer, you can compare the output of the neural network with the ground-truth; if the output is far off, that means you need to adjust the weights to make a better prediction next time. You evaluate how good (or bad) the model is currently performing through a loss function. The `rusty-machine` default is the *Binary Cross-Entropy* loss function, which will give a higher value if the output (cat or dog) does not match the ground-truth and a lower value vice versa. The goal is to adjust the weights to minimize the loss (i.e., make the loss function return the smallest value possible). To achieve this, you use an algorithm called *gradient descent*. Gradient descent will adjust the weight towards the direction by which the loss will go down.

So if you provide a lot of data to the neural network, it will feed a small batch of data into the network and check the network's output against the ground-truth answer. Then it will try to adjust the weights to make the loss go down. Then it will feed the next batch of data and continue to adjust the weights. Once all the data are fed, the neural network should have nodes with weights that captures the characteristics of the training data. When you get a new body measurement, you can feed it to the network. The output will then tell us if the body measurement belongs to a dog or cat.

Preparing the training data and testing data

To keep the example simple, you are going to use the same kind of input as the K-means example: height and length. You'll create 2000 samples for cats and 2000 for dogs. But there is some difference between this data and the K-means data:

- You'll need to provide the "answer", or the ground-truth labels for each sample.
- You need to generate two sets of data: the training data and testing data.

The reason that you need the ground-truth labels is that a neural network is a supervised model, which means that it needs to learn by comparing its prediction with the ground-truth and try to improve the accuracy.

You also need to split the data into a training set and a testing set. The training set is used to train the model. The testing set is used to verify how accurate the trained model is. One key point is that the model should never see the data in the testing set during the training phase. The reason is that since you have the answer already, you can't let the model see the testing data during training. Otherwise, it already knows the answer and can quickly achieve 100% accuracy by memorizing the training data. Even if the algorithm doesn't intentionally memorize the answer, using the testing data in training will usually lead to overfitting. Overfitting is when the model tries to accommodate the particular training data set too much and fails to generate a model that is general enough. That means the model will work very well on the same set of training data but fails miserably for any data that it hasn't seen before. Therefore you need to split the data into two sets.

To genenrate the training data, the overall code structure looks the same as in Listing 9.4. The only difference is in the `generate_data()` function, presented in Listing 9.5.

TODO: Add filename

```
use␣serde::Serialize;

#[derive(Debug,␣Serialize)]
struct␣Sample␣{␣//␣[1]
␣␣␣␣height:␣f64,
␣␣␣␣length:␣f64,
␣␣␣␣category_id:␣usize
}

fn␣generate_data(centroids:␣&Matrix<f64>,
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣points_per_centroid:␣usize,
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣noise:␣f64)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣->␣Vec<Sample>␣{␣//␣[2]
␣␣␣␣//␣input␣validation
␣␣␣␣let␣mut␣samples␣=␣Vec::with_capacity(points_per_centroid);

␣␣␣␣let␣mut␣rng␣=␣thread_rng();
```

```
    let normal_rv = Normal::new(0f64, noise).unwrap();

    for _ in 0..points_per_centroid {
        // Generate points from each centroid
        for (centroid_id, centroid) in centroids.iter_rows().
    ↪ enumerate() {
            let mut point = Vec::with_capacity(centroids.cols())
    ↪ ;
            for feature in centroid.iter() {
                point.push(feature + normal_rv.sample(&mut rng))
    ↪ ;
            }

            samples.push(Sample { // [3]
                height: point[0],
                length: point[1],
                category_id: centroid_id,
            });
        }
    }

    samples
}
```

Each row in the new data now has three columns: length, height, and the cat-or-dog label. The cat-or-dog label will be an integer, 0 represents dog, and 1 represents cat[6]. You can serialize all the fields into `f64`, but the integer label will become `0.0` or `1.0` in the CSV file. To force it to serialize to a nice looking `0` or `1`, you need to define the "schema". The schema is simply a struct that implements the `serde::Serialize` trait. You define this in the `struct Sample` ([1]). The `generate_data()` function will return a `Vec<Sample>` instead of a `Matrix<f64>`. When you call `csv::Writer.serialize()` with a `Sample`, it will use Serde to serialize it to something like `25.24, 60.03, 1`.

You'll create a binary: `src/bin/generate_data.rs` to generate the training and testing data by running it twice. You generate a total of 4000 training samples and 4000 testing samples.

[6]This number is assigned arbitrarily. There is no special meaning behind the numbers.

Setting up the neural network model

After the training and testing data are generated, you need to build the model training and predicting code. You are going to put them in a new binary `src/bin/train_and_predict.rs`. What this binary has to do is:

- Read and parse the training data into a `Vec`, and shape it into a `Matrix`.
- Normalize the training data.
- Initialize the neural network model.
- Feed the normalized training data into the model for training.
- Read and parse the testing data into a `Vec`, and shape it into a `Matrix`.
- Normalize the testing data using the same parameter for normalizing the training data.
- Use the trained model to make predictions on the testing data.

You'll discuss each task in the following sections.

Reading the training and testing data

In the K-means example (Listing 9.4), you read the CSV input from STDIN. However, in a supervised model, you need two input files: the training and testing data. So this time, you are going to give the CSV files' paths as CLI arguments and read them directly from the file. Using `StructOpt` you introduced in Chapter 2, you create two arguments `training_data_csv` and `testing_data_csv` (Listing 9.5).

TODO: Add filename

```
use structopt::StructOpt;

#[derive(StructOpt)]
struct Options {
    #[structopt(short = "r", long = "train", parse(from_os_str))
    ↪ ]
    /// Training data CSV file
    training_data_csv: std::path::PathBuf,

    #[structopt(short = "t", long = "test", parse(from_os_str))]
    /// Testing data CSV file
    testing_data_csv: std::path::PathBuf,
}

fn main() -> Result<(), Box<dyn Error>>{
```

```
    let options = Options::from_args();

    //...

    Ok(())
}
```

You might recall how to serialize the data from a Rust struct into CSV in section 9.5. Now you need to do the opposite: to deserialize the CSV data back to Rust structs. For this, you need to define the same data schema in a Rust struct and provide it to `csv::Reader`(Listing 9.5).

TODO: Add filename

```
extern crate rusty_machine;

use serde::Deserialize;
use csv;
use rusty_machine::linalg::Matrix;

#[derive(Debug, Deserialize)]
struct SampleRow { //[1]
    height: f64,
    length: f64,
    category_id: usize,
}

fn read_data_from_csv(file_path: std::path::PathBuf)
    -> Result<(Matrix<f64>, Matrix<f64>), Box<dyn Error>> {
    let mut input_data = vec![];
    let mut label_data = vec![];
    let mut sample_count = 0;
    let mut reader = csv::Reader::from_path(file_path)?; //[2]
    for raw_row in reader.deserialize() { //[3]
        let row: SampleRow = raw_row?;
        input_data.push(row.height);
        input_data.push(row.length);
        label_data.push(row.category_id as f64);
        sample_count += 1
    }

    let inputs = Matrix::new(sample_count, 2, input_data);
    let targets = Matrix::new(sample_count, 1, label_data);
    return Ok((inputs, targets))
}

fn main() -> Result<(), Box<dyn Error>>{
```

```
    let options = Options::from_args();

    let (inputs, targets) = read_data_from_csv(options.
        ↪ training_data_csv).unwrap();
    //...
}
```

The schema you define is the struct `SampleRow` ([1]), exactly the same as in the `generate_data.rs`. But this time, you derive the `Deserialize` trait on it. You created a utility function `read_data_from_csv()` to read the data from CSV file path. The line that actually reads the CSV file is on [2], where you use `csv::Reader::from_path()`. The path parameter is the `PathBuf` you get from the CLI options. Once the file is loaded into memory, you loop through the rows obtained by calling `reader.deserialize()` ([3]). This will deserialize the CSV line into `Result<SampleRow, Error>`.

In the loop, you put the rows into two vectors, `input_data` and `label_data`. You put the height and length into `intput_data`; and you put the `category_id` into `label_data`[7]. These two vectors are then converted to the `Matrix` type that `rusty-machine` models accept.

Normalizing the training data

Before you feed the data into the neural network model, there is a less-obvious step you have to take, which will significantly speed up the training and accuracy of the model. This step is called *normalization*. The goal of the normalization is to shift and scale the input data, so it has a mean of 0 and a standard deviation of 1. This is very helpful for models like a neural network because when you do gradient descent, a normalized data set means the optimization process will not be dominated by one single dimension that has a much larger scale than the others. It also means that the cost function will have a smoother shape, which means the gradient descent process will be faster and smoother.

The normalization process involves two steps:

- Calculate the mean of the data set and subtract all the data points by the mean. This shifts the mean of the dataset to 0.
- Calculate the standard deviation of the data set, divide each data point by the standard deviation. This scales the data set to a standard deviation of 1.

You must keep the mean and standard deviation of the *training* data at hand. Because when you normalize the *testing* data, you are going to use the mean and standard deviation from the *training* data. This is because all the parameters in the neural network model will

[7] You might notice that you convert the `category_id` from `usize` to `f64`. You might wonder why you don't just use `f64` in the CSV format? That's because this is a categorical integer. If you use `f64` in CSV, they'll become `0.0`, `1.0` and so on, which doesn't look nice if you want to check the training/testing data with spreadsheet software.

be trained for the normalized training data. If the testing data has a different mean and standard deviation, the model's prediction might be off.

You don't need to write this part of the code ourselves. The `rusty-machine` contains a handy `rusty_machine::data::transforms::Standardizer`. The `Standardizer` implments the `Transformer` trait, which defines a shared interface for commonly-used data pre-processing transformations.

The `Standardizer` can be initialized with the `new()` function with two options: the desired mean and standard deviation. The normalization process you described requires a mean of 0 and a standard deviation of 1, but the `Standardizer` can scale the data to any other means and standard deviations. Once initialized, the `Standardizer` instance has two functions defined by the `Transformer` trait:

- `fit()`: Calculates the mean and standard deviation from the input data and store it inside the `Standardizer` instance.
- `transform()`: Do the transformation on the provided data using the mean and standard deviation learned in the `fit()` step.

You can see the `Standardizer` in action in Listing 9.5.

TODO: Add filename

```
fn main() -> Result<(), Box<dyn Error>>{
    // Loading the training_inputs

    let mut standardizer = Standardizer::new(0.0, 1.0);

    standardizer.fit(&training_inputs).unwrap();
    let normalized_training_inputs = standardizer.transform(
    ↪ training_inputs).unwrap();

    // Train the model with normalzied_training_inputs

    // Read the testing_inputs
    let normalized_test_cases = standardizer.transform(
    ↪ testing_inputs).unwrap();

    // Run the prediction with normalized_test_cases

    Ok(())
}
```

You first run `Standardizer.fit()` with the training data to learn its mean and standard deviation. Then you use this configuration to perform `Standardizer.transform()` on both the training data and testing data. You then feed the model with the normalized data instead of the raw ones directly read from CSV files.

Training and predicting

Finally, with all these efforts for reading, parsing, and normalizing data, you are ready to build the neural network model. The neural network model takes a little more configuration then the K-means, which only has one configuration: the k. For the neural network model, you have the option to set the following:

- The number of layers and the number of nodes per layer.
- The criterion, including an activation function and a loss function
- The optimization algorithm

The `rusty_machine::learning::nnet::NeuralNet` has a `::default()` function. If you look under the hood, it chooses the configurations like Listing 9.5.
TODO: Add filename

```
use rusty_machine::learning::nnet::{NeuralNet, BCECriterion};
use rusty_machine::learning::optim::grad_desc::StochasticGD;
use rusty_machine::learning::SupModel;

fn main() -> Result<(), Box<dyn Error>>{
    // Loading training data and pre-processing

    let layers = &[2, 2, 1]; // [1]
    let criterion = BCECriterion::default(); // [2]
    let gradient_descent = StochasticGD::new(0.1, 0.1, 20); //
    ↪ [3]
    let mut model = NeuralNet::new(layers, criterion,
    ↪ gradient_descent);

    model.train(&normalized_training_inputs, &targets).unwrap();

    // Testing

    Ok(())
}
```

Let's break this down line-by-line. In [1], you define the layers as `[2, 2, 1]`, which means a three-layer architecture. The first layer has two input neurons, the middle layer has two neurons, and the output layer has one neuron. By default, the `NeuralNet` chooses the binary cross-entropy criterion (`BCECriterion`)([2]), which uses the Sigmoid activation function and the cross-entropy error as the loss function. Finally, the Stochastic Gradient descent (`StochasticGD`) is chosen as the optimization algorithm in [3]. The Stochastic Gradient descent has three parameters:

- Momentum (default: 0.1)

- Learning rate [8] (default: 0.1)
- Number of iterations (default: 20)

All these parameters have some impact on how the neural network model performs. Since this is not a book on machine learning, you are not going to discuss how to tune them in detail, and you stick to the defaults. A vital skill for a machine learning expert is to understand the mathematical meaning of these parameters and how to tune them to make the model accurate and robust.

You collect all these configurations (layers, criterion, and gradient descent algorithm) and pass them to `NeuralNet::new()` to create the model. The `NeuralNet` model implements the `SupModel` trait. `SupModel` also has the `.train()` and `.predict()` functions. The only different of `SupModel` from `UnSupModel` is that its `.train()` function takes an extra `target` parameter, which contains the ground truth labels:

```
pub trait SupModel<T, U> {
    fn train(&mut self, inputs: &T, targets: &U) ->
    LearningResult<()>;
    //                                  ^---------- extra parameter
    //...
}

pub trait UnSupModel<T, U> {
    fn train(&mut self, inputs: &T) -> LearningResult<()>;
    //...
}
```

Therefore, you train the model by calling:

```
model.train(&normalized_training_inputs, &targets).unwrap();
```

This step will usually take some time to run because the neural network is doing all the complicated mathematical computations and training the model. Once trained, it will store all the learned weights and other parameters inside itself, and you are ready to use it to make predictions.

Making the prediction

To check if the model is trained properly, you prepared 4000 new data points using the `generate_data.rs` script. This time you are only going to pass the height and length to the trained neural network model. The neural network model will pass these inputs into the network and calculate all the signals all the way to the output node. Then the output node will give us a signal between 0 to 1. A zero means the model believes that input is most likely a dog, and a one means it's a cat. You can compare this prediction with the generated answer and see if the model is correct or not.

[8] Actually, the second argument is the square root of the raw learning rate.

You load the testing data from CSV and normalize it using the `Standardizer` you created during training. Then you can use `model.predict()` on it to get a list of predicted labels (Listing 9.5).

TODO: Add filename

```
fn main() -> Result<(), Box<dyn Error>>{
    // Training the model

    // Testing ====================
    let (testing_inputs, expected) = read_data_from_csv(options.
    ↪ testing_data_csv).unwrap();

    // Normalize the testing data using the mean and variance of
    ↪ the training data
    let normalized_test_cases = standardizer.transform(
    ↪ testing_inputs).unwrap();

    let res = model.predict(&normalized_test_cases).unwrap();
}
```

The `res` generated by `model.predict()` will be a list of labels 0 or 1, which is the result you are looking for. Notice that you only use the height and length part of the testing data (i.e., `testing_inputs`). The actual label `expected` is not given to the neural network. Otherwise, it'll know the answer. If you compare `res` with `expected`, you'll see that almost all predictions are correct. This is not usually the case in real-life applications. The reason that you can achieve a 100% accuracy is that the training and testing data are artificially generated to be easy for a neural network model, and it's free of noise. But this example shows us how to train a supervised model using `rusty-machine`.

9.6 Other Alternatives

As you can see in the examples of this chapter, machine learning is not just about training the model. There are many data-related operations before and after you train the model. This kind of operations include:

- Reading and writing CSV or other structural data formats.
- Pre-processing the data (e.g., normalization).
- Setting and loading model configurations and parameters.
- Visualizing the data

It's not really practical to write all these code from scratch for every machine learning application. You need a strong ecosystem with many pre-built crates to help us quickly

and efficiently implements the learning part without worrying about fundamental tasks like linear algebra and data manipulation. Similar to other fields in Rust, there is a "Are you learning yet?" page[9] page tracking how the ecosystem is doing. As stated on the page, the machine learning landscape in Rust is "ripe for experimentation, but the ecosystem isn't very complete yet."

For the foundational mathematical crates, `nalgebra` and `ndarray` are starting to get wider adoption. They provide linear algebra and array/matrix operation similar to the `ndarray` in python. Many machine learning algorithms also relies on the field call high-performance computing (HPC), that harness the power of the hardware (CPU, GPU, etc.) and parallelism. There are much experimentation in this field like `std::simd`, `RustCUDA`, and `rayon`, just to name a few.

If you consider traditional machine learning ("traditional" in the sense that it's not deep learning), there are `rusty-machine`, `rustlearn`, and `Juice`. `Rustlearn` is very similar to `rusty-machine`, containing many traditional machine learning models. `Juice` is an attempt to revive the ambitious `leaf` project, which was abandoned after the startup building it was closed. They all have implemented several commonly used traditional machine learning models, but the development seems to be slow in recent years.

As for deep learning, there is no mature library built from scratch using Rust. So to tap into the field of deep learning, the best bet now is to use Rust bindings to mature libraries written in other languages. There are Rust bindings in the TVM project, which is an open-source deep learning compiler stack. There is also `tensorflow/rust` for TensorFlow and `tch-rs` for PyTorch, which are two mainstream deep learning frameworks.

Rust has excellent potential to enable high performance and safe machine learning applications. But there is much more work to get the ecosystem ready for production use.

[9] https://www.arewelearningyet.com/