

[Python Learning Course](#)[Spam Filter](#)

FOLLOW:



# Building a Spam Filter Using Machine Learning

BY SUPRIYO BISWAS · MARCH 27, 2017

```
{  
  "total_spam": 3021,  
  "total_ham": 2966,  
  "spammicity": {  
    "replica": 0.833144,  
    "singles": 0.44352833,  
  }  
}
```

Machine learning is everywhere. From self driving cars to face recognition on Facebook, it is machine learning behind the scenes that drives all of it. If you've ever used GMail or Yahoo Mail, you must have seen a folder named "Spam" where all unwanted mail goes in. Have you ever wondered how that works? That's machine learning at work, too!

In this article, we're going to develop a simple spam filter in node.js using a machine learning technique named "Naive Bayes". The filter will be able to determine whether an email is spam by looking at its content.

## The basics of machine learning

The word "machine learning" has a certain aura around it. Journalists and entrepreneurs talk about it as if something out of the world happened. In reality though, it is much simpler.

Machine learning is a field of computer science where computers can learn to do something, without the need to explicitly program them for the task. First, the algorithm is made to look at a certain set of data, in order to train it for the task. Then, we give the algorithm data it has never seen before, and perform the task on this data.

### NEXT STORY

[Adding HTTPS to Your Site with Cloudflare](#)

### PREVIOUS STORY

[8 Critical Security Issues to Avoid in Your Website](#)

### SPONSORED SEARCHES

[Python Learning Course](#)[Spam Filter](#)[Machine Learning Spam Detecti](#)[Data Analysis Machine Learnin](#)[Machine Learning Algorithms Tut](#)

### RECENT POSTS

[cURL Command Tutorial with Examples](#)[OpenVZ vs KVM vs Xen – Virtualization Technologies Explained](#)[How to Download Files and Web Pages with Wget](#)[How to Kill a Process in Linux](#)[A Guide to the Linux "Find" Command](#)

Thus, a machine learning algorithm can be thought to have two phases: “training” and “prediction”. For each of these phases, we use various mathematical methods.

There are a wide variety of machine learning algorithms. Depending upon how these algorithms “learn”, they can be categorized as:

- In **supervised learning**, the algorithm is provided with data, along with the correct answer for it. So, if we were to develop an algorithm to predict house prices, and you gave the size of the land and the price to the algorithm, it would fall into this category.
- In **unsupervised learning**, the algorithm is provided with data, but the answers are not provided to it. It is upon the algorithm to find structure in the data, and figure out things from there. They are commonly used in places such as market segment analysis. We don’t know what kind of market segments are there for your product — and the algorithm must figure it out.

Again, based on the type of output that a machine learning algorithm produces, we can categorize them into two types:

- **Classification:** These algorithms produce outputs that categorize the data. For example, an algorithm which takes in medical information about a patient and produces a diagnosis that may be only one of “no cancer”, “lung cancer” or “colon cancer” would be of this type.
- **Regression:** In regression, the output types are continuous valued. For example, consider the previous example of predicting house prices. The predicted price would depend on the size of the land. Unlike regression, we don’t have outputs that nicely categorize the data.

As you’ll see later in this article, we’ll train our filter using a collection of spam and non-spam(aka “ham”) emails. So, we’ll provide right answers to train the filter, and later in the prediction phase, its output for a given message would be either “spam” or “ham”. So, this filter is an example of a supervised classification algorithm.

Now that you know the types of machine learning algorithms, let us see what we would need to build the filter.

## The training phase

As we’ve already said, our filter would analyze the content of an email to tell if it’s spam. Let us now take a deeper look at the problem.

If you take a look at the typical spam email, you’d find words such as “replica”, “loans” and “singles”. These words are hard to come across in ham, though. Similarly, words such as “presentation” and “manager” is typical in ham, but hard to find in spam.

During the training phase, we’d tell our program to look at a set of spam or ham emails. For every distinct word in the email, it would note down the probability with which the word occurs in either spam or ham. In this article, we’ll refer to these values as “spammicity” and “hammicity”. If we were to define spammicity in a more verbose way, we could say it was the probability of a word occurring, given



that the mail is spam. We'll represent this as  $P(W|S)$ . Similarly, we could represent hamcity as  $P(W|H)$ .

So, for example, you have 10000 emails in your training dataset, out of which 6000 mails are spam and the rest is ham. The word "replica" occurs in 2000 spam emails and 10 ham emails. So,  $P("replica"|S) = \frac{2000}{6000} = 0.333$  and  $P("replica"|H) = \frac{10}{4000} = 0.0025$ .

You might be thinking as to why we need the hamcity value at all. If both your spam and ham emails contain the word "car" frequently, we'd have high spamcity and hamcity values. If we didn't take the hamcity of the emails into consideration, then we'd classify every email containing "car" as spam.

Next, we'll discuss how the prediction phase would work.

## The prediction phase

We now know the spamcity and hamcity values for every word. We can also easily find out the probability of a message being spam or ham. Considering our previous example, the probability that of a message is spam is  $\frac{6000}{10000} = 0.6$  and the probability of a message being ham is  $\frac{4000}{10000} = 0.4$ . We'll refer to these probabilities as  $P(S)$  and  $P(H)$ , respectively.

Now, let's say, we've got a new email containing an arbitrary word  $W$  in it. Our job is to find out whether the message is spam, given that it contains  $W$ . This is an exact opposite situation of finding the spamcity, and can be represented as  $P(S|W)$ .

Here's where we can use [Bayes Theorem](#), and calculate the probability as:

$$P(S|W) = \frac{P(W|S)P(S)}{P(W|S)P(S) + P(W|H)P(H)}$$

Continuing with the "replica" example, the probability of the message being spam would be:

$$P(S|"replica") = \frac{0.333 \times 0.6}{0.333 \times 0.6 + 0.0025 \times 0.4} = 0.9925$$

So far, we've considered a single word. An email consists of multiple words, and we have to use Bayes theorem once again to find out the overall probability of it being spam. This model assumes all words are equally likely to appear at any given position in the message. However, due to the grammar rules of a language, this never happens; so this model is less than perfect. However, it is good enough for our purposes, and many spam filters use this model due to its simplicity.

It may so happen that the filter may encounter a new word when classifying an unknown email. We'd first discard these words, because there's no way we can make predictions about them. After this, assume there are  $n$  distinct words in the email,  $W_1, W_2, \dots, W_n$ . Then, we find out  $p_1 = P(S|W_1), p_2 = P(S|W_2), \dots, p_n = P(S|W_n)$ . We can then find out the overall probability of the email being spam like so:

$$p = \frac{p_1 p_2 \dots p_n}{p_1 p_2 \dots p_n + (1 - p_1)(1 - p_2) \dots (1 - p_n)}$$

If  $p$  contains a sufficiently large value (say,  $> 0.5$ ), we'll assume the email to be spam.

## A few practical considerations

Before we write the code, there are a few practical considerations to make so that our filter can work better.

In the formula for  $P(S|W)$ , we've considered the values for  $P(S)$  and  $P(W)$ . However, in real situations,  $P(S)$  can be high as 0.8, which would lead to misclassifications of ham emails as spam. Given a new email, we have no prior reason to suspect it may be spam, so we'll use  $P(S) = P(W) = 0.5$ . In other words, we assume it is equally probable for the message to be either spam or ham. Thus, the formula for  $P(S|W)$  would get reduced to:

$$P(S|W) = \frac{P(W|S)}{P(W|S) + P(W|H)}$$

There's another optimization we can make. In every email, you'd find some common words like "if", "be", "then" and so on. There's no need to consider these words because they aren't helpful to find out if the message is spam.

Now, take a look at the formula for calculating the overall probability the email being spam. Probability values always lie between 0 and 1, and computers don't handle multiplying such small values well. As a result, to avoid errors creeping into the value of  $p$ , we rewrite the formula like so:

$$p = \frac{1}{1 + e^\eta}$$

where:

$$\eta = \sum_{i=1}^n [\log_e(1 - p_i) - \log_e(p_i)]$$

(You can read the [derivation](#) if you're interested.)

## Writing the classifier

Enough with the theory! We'll now write the spam filter. We'll begin off by writing:

```
#!/usr/bin/env node

"use strict";

const fs = require("fs");

let classifier = {
};
```

Here, we've used the `fs` module, which helps us to read from files and write to them. The rest of the code discussed in this section would go into the `classifier` object. This object will be responsible for learning from the training examples and for classifying new emails.

Now, we need to define a data structure to hold information about previously seen emails. For this purpose, we define a `dataset` object like so:

```
dataset: {  
  total_spam: 0,  
  total_ham: 0,  
  spammicity: {},  
  hammicity: {}  
},
```

Initially, we have seen exactly zero spam and ham emails, and we don't know anything about the words in an email. This is why `total_spam` and `total_ham` are initialized to zero, and the `spammicity` and `hammicity` objects are empty. After training, the `spammicity` and `hammicity` objects will contain words and their corresponding values like the example below:

```
spammicity: {  
  "replica": 0.8831255,  
  "watches": 0.910244  
}
```

We would also need to load the training data from the disk and save it when we're done. In order to do this, we define two functions, `load()` and `save()`. These functions save this data as JSON into `training.json`.

```
load: function() {  
  try {  
    this.dataset =  
    JSON.parse(fs.readFileSync(`${__dirname}/training.json`, "utf-8"));  
  }  
  catch (e) {  
    // nothing to do here.  
  }  
},  
  
save: function() {  
  fs.writeFileSync(`${__dirname}/training.json`,  
  JSON.stringify(this.dataset));  
},
```

Now, say, we've loaded the data of a spam email into a string. We have to ignore the common words in this string, and create a data structure consisting of all the distinct words inside the string. For this purpose, we've defined a function,

`createTable()`:

```
createTable: function(str) {  
  let table = {}, result, word;
```

```

let rgx = /\b([a-z]{2,}-)*[a-z]{3,}/gi;

while ((result = rgx.exec(str)) !== null) {
    word = result[0].toLowerCase();

    if (!this.isCommonWord(word)) {
        table[word] = true;
    }
}

return table;
},

```

This function returns an object, which contains the list of words like so:

```

{
  "hello": true,
  "world": true
}

```

At this point, you might be thinking, why not use an array? Using an array would require checking to ensure there are no duplicate elements in it. Using the key-value association of Javascript objects is a much simpler and faster way of doing this, because duplicate keys cannot exist in an object.

The regular expression `/\b([a-z]{2,}-)*[a-z]{3,}/gi` fetches words with more than two letters and phrases (like “out-of-the-box”) from the string. Words with less than three letters are ignored because it would match words like “an”, “by” and “of” which are very common, and aren’t helpful for our purpose.

Despite this initial level of filtering, words such as “the” and “for” would still get through. Thus, we’ve defined another function, `isCommonWord()`, which filters out common names and some common words.

```

isCommonWord: function(str) {
    return
    /^(?:the|and|that|have|for|not|with|you|this|but|his|from|they|we|
    say|her|she|will|one|all|would|there|their|what|out|about|who|get|
    which|when|make|can|like|time|just|him|know|take|people|into|year|
    your|good|some|could|them|see|other|than|then|now|look|only|come|i
    ts|over|think|also|back|after|use|two|how|our|work|first|well|way|
    even|new|want|because|any|these|give|day|most|ever|among|stand|yet
    |often|hour|talk|might|start|turn|help|big|small|keep|old|out|high
    |low|ask|should|down|thing|aaron|adam|alan|albert|alice|amanda|amy
    |andrea|andrew|angela|ann|anna|anne|annie|anthony|antonio|arthur|a
    shley|barbara|benjamin|betty|beverly|billy|bobby|bonnie|brandon|br
    enda|brian|bruce|carl|carlos|carol|carolyn|catherine|charles|chery
    l|chris|christina|christine|christopher|clarence|craig|cynthia|dan
    iel|david|deborah|debra|denise|dennis|diana|diane|donald|donna|dor
    is|dorothy|douglas|earl|edward|elizabeth|emily|eric|ernest|eugene|
    evelyn|frances|frank|fred|gary|george|gerald|gloria|gregory|harold
    |harry|heather|helen|henry|howard|irene|jack|jacqueline|james|jane
    |janet|janice|jason|jean|jeffrey|jennifer|jeremy|jerry|jesse|jessi
    ca|jimmy|joan|joe|john|johnny|jonathan|jose|joseph|joshua|joyce|ju
    an|judith|judy|julia|julie|justin|karen|katherine|kathleen|kathryn
    |kathy|keith|kelly|kenneth|kevin|kimberly|larry|laura|lawrence|lil
    lian|linda|lisa|lois|lori|louis|louise|margaret|maria|marie|marily
    n|mark|martha|martin|mary|matthew|melissa|michael|michelle|mildred
    |nancy|nicholas|nicole|norma|pamela|patricia|patrick|paul|paula|pe
    ter|philip|phillip|phyllis|rachel|ralph|randy|raymond|rebecca|rich
    ard|robert|robin|roger|ronald|rose|roy|ruby|russell|ruth|ryan|samu

```

```
el|sandra|sara|sarah|scott|sean|sharon|shawn|shirley|stephanie|ste
phen|steve|steven|susan|tammy|teresa|terry|theresa|thomas|timothy|
tina|todd|victor|virginia|walter|wanda|wayne|william|willie)$.tes
t(str);
},
```

## Learning from examples

It's time to define the `learnSpam()` function. It takes the contents of an email as a string, gets the list of words with `createTable()`, and adjusts the `dataset` values.

```
learnSpam: function(str) {
  let table = this.createTable(str);
  let total = this.dataset.total_spam;

  for (let word in this.dataset.spammicity) {
    let old_spammicity = this.dataset.spammicity[word];

    if (table[word] === undefined) {
      this.dataset.spammicity[word] = (total *
old_spammicity)/(total + 1);
    }
    else {
      this.dataset.spammicity[word] = (total *
old_spammicity + 1)/(total + 1);
      delete table[word];
    }
  }

  for (let word in table) {
    this.dataset.spammicity[word] = 1/(total + 1);
  }

  this.dataset.total_spam = total + 1;
},
```

This function is a bit complex, so let's take an example. Suppose, our classifier has seen five emails. Three of them contain “replica” and two of them contain “loans”. Thus, we would have:

$$P(\text{"loans"}|S) = \frac{2}{5} = 0.4$$

$$P(\text{"replica"}|S) = \frac{3}{5} = 0.6$$

Now, suppose we have an email which reads: “replica watches”. The value of `total_spam` and the denominators of all the fractions would be incremented by 1.

Now:

- The filter has never seen “watches” but it occurs in the new message. So, the numerator will be equal to one, because after the training, the filter would have seen it exactly once.
- The filter has seen “loans”, but it does not occur in this message. So, the new numerator won't increase; it will be equal to the previous numerator. However, since we didn't store the old numerator in the dataset, we have to calculate it indirectly with `total * old_sapmmicity`.

- The word “replica” occurs both in the new message and in the previously seen messages. Thus, we must add one to the old numerator.

We'll get these new values, and you can manually verify that these are indeed correct:

$$P(\text{"watches"}|S) = \frac{1}{5+1} = \frac{1}{6} = 0.167$$

$$P(\text{"loans"}|S) = \frac{0.4 \times 5}{5+1} = \frac{2}{6} = 0.333$$

$$P(\text{"replica"}|S) = \frac{0.6 \times 5 + 1}{5+1} = \frac{4}{6} = 0.666$$

The `learnHam()` function is similar, except for the fact that it works on hammicity values.

## Making predictions

The `predict()` function creates a list of words with `createTable()`, and calculates  $P(S|W)$  for each word. It then combines them with the alternative formula for  $p$  we discussed above. We indicate that the message is spam by returning true when the  $p > 0.5$ .

```
predict: function(str) {
  let table = this.createTable(str), word, p = {};

  for (word in table) {
    let spammicity = this.dataset.spammicity[word] || 0;
    let hammicity = this.dataset.hammicity[word] || 0;

    if (spammicity !== 0 && hammicity !== 0) {
      p[word] = spammicity / (spammicity + hammicity);
    }
  }

  let eta = 0;

  for (word in p) {
    eta += (Math.log(1 - p[word]) - Math.log(p[word]));
  }

  let p_final = 1 / (1 + Math.pow(Math.E, eta));

  return (p_final > 0.5);
}
```

When a spammicity or hammicity entry cannot be found for a given word, we assume it to be zero. This might seem strange, but it is the right thing to do. For example, if the word “replica” never occurs in ham messages, there would be no entry in the hammicity table. By assuming a hammicity of zero, we get  $P(S|\text{"replica"}) = \frac{1}{1+0} = 1$ , which is the right answer.



If both of spammicity and hammicity values are zero, then, we'll ignore the word as it has never been seen by the filter.

## The driver program

This completes our classifier. Now, we need some code to read files and to call the functions defined in `classifier`. This code should be defined outside the classifier object and below it. The code is fairly self-explanatory, and we won't describe it here.

```
let arg_length = process.argv.length;

if (arg_length === 2) {
  console.log(`Usage: ${process.argv[0]} ${__filename} {-s|-h|-p} {filename}`);

  process.exit(0);
}

let actions = ["-s", "-h", "-p"];
let type = actions.indexOf(process.argv[2]);

if (arg_length === 3 || type === -1) {
  console.log(`Incorrect arguments, please run
"${process.argv[0]} ${__filename}" for help.`);
  process.exit(1);
}

classifier.load();

switch (type) {
  case 0:
    for (let i = 3; i < arg_length; i++) {
      let file_contents =
fs.readFileSync(process.argv[i], "utf-8");
      classifier.learnSpam(file_contents);
    }

    classifier.save();
    break;

  case 1:
    for (let i = 3; i < arg_length; i++) {
      let file_contents =
fs.readFileSync(process.argv[i], "utf-8");
      classifier.learnHam(file_contents);
    }

    classifier.save();
    break;

  case 2:
    for (let i = 3; i < arg_length; i++) {
      let file_contents =
fs.readFileSync(process.argv[i], "utf-8");
      console.log(process.argv[i] + ": " +
classifier.predict(file_contents));
    }
    break;
}
```

The functions of the classifier can now be accessed with the help of the command line switches. Assuming that you saved the file as `spamfilter.js`, the following functions will be available:

- `node spamfilter.js -s <list of files>` trains the filter to learn the contents of the given files as spam.
- `node spamfilter.js -h <list of files>` trains the filter to learn the contents of the given files as ham.
- `node spamfilter.js -p <list of files>` predicts if the content of each given file is spam or ham.

## The entire program

The source for the entire program is as below.

```
#!/usr/bin/env node

"use strict";

const fs = require("fs");

let classifier = {
  dataset: {
    total_spam: 0,
    total_ham: 0,
    spammicity: {},
    hammicity: {}
  },

  load: function() {
    try {
      this.dataset =
JSON.parse(fs.readFileSync(`${__dirname}/training.json`, "utf-8"));
    }
    catch (e) {
      // nothing to do here.
    }
  },

  save: function() {
    fs.writeFileSync(`${__dirname}/training.json`,
JSON.stringify(this.dataset));
  },

  isCommonWord: function(str) {
    return
/^(:the|and|that|have|for|not|with|you|this|but|his|from|they|we|
say|her|she|will|one|all|would|there|their|what|out|about|who|get|
which|when|make|can|like|time|just|him|know|take|people|into|year|
your|good|some|could|them|see|other|than|then|now|look|only|come|i
ts|over|think|also|back|after|use|two|how|our|work|first|well|way|
even|new|want|because|any|these|give|day|most|ever|among|stand|yet
|often|hour|talk|might|start|turn|help|big|small|keep|old|out|high
|low|ask|should|down|thing|aaron|adam|alan|albert|alice|amanda|amy
|andrea|andrew|angela|ann|anna|anne|annie|anthony|antonio|arthur|a
shley|barbara|benjamin|betty|beverly|billy|bobby|bonnie|brandon|br
enda|brian|bruce|carl|carlos|carol|carolyn|catherine|charles|chery
l|chris|christina|christine|christopher|clarence|craig|cynthia|dan
iel|david|deborah|debra|denise|dennis|diana|diane|donald|donna|dor
is|dorothy|douglas|earl|edward|elizabeth|emily|eric|ernest|eugene|
evelyn|frances|frank|fred|gary|george|gerald|gloria|gregory|harold
|harry|heather|helen|henry|howard|irene|jack|jacqueline|james|jane
|janet|janice|jason|jean|jeffrey|jennifer|jeremy|jerry|jesse|jessi
ca|jimmy|joan|joe|john|johnny|jonathan|jose|joseph|joshua|joyce|ju
an|judith|judy|julia|julie|justin|karen|katherine|kathleen|kathryn
|kathy|keith|kelly|kenneth|kevin|kimberly|larry|laura|lawrence|lil
lian|linda|lisa|lois|lori|louis|louise|margaret|maria|marie|marily
n|mark|martha|martin|mary|matthew|melissa|michael|michelle|mildred
|nancy|nicholas|nicole|norma|pamela|patricia|patrick|paul|paula|pe
ter|philip|phillip|phyllis|rachel|ralph|randy|raymond|rebecca|rich
```

```

ard|robert|robin|roger|ronald|rose|roy|ruby|russell|ruth|ryan|samuel|sandra|sara|sarah|scott|sean|sharon|shawn|shirley|stephanie|stephen|steve|steven|susan|tammy|teresa|terry|theresa|thomas|timothy|tina|todd|victor|virginia|walter|wanda|wayne|william|willie)$/ .test(str);
    },

    createTable: function(str) {
        let table = {}, result, word;
        let rgx = /\b([a-z]{2,})*[a-z]{3,}/gi;

        while ((result = rgx.exec(str)) !== null) {
            word = result[0].toLowerCase();

            if (!this.isCommonWord(word)) {
                table[word] = true;
            }
        }

        return table;
    },

    learnSpam: function(str) {
        let table = this.createTable(str);
        let total = this.dataset.total_spam;

        for (let word in this.dataset.spammicity) {
            let old_spammicity = this.dataset.spammicity[word];

            if (table[word] === undefined) {
                this.dataset.spammicity[word] = (total * old_spammicity) / (total + 1);
            } else {
                this.dataset.spammicity[word] = (total * old_spammicity + 1) / (total + 1);
                delete table[word];
            }
        }

        for (let word in table) {
            this.dataset.spammicity[word] = 1 / (total + 1);
        }

        this.dataset.total_spam = total + 1;
    },

    learnHam: function(str) {
        let table = this.createTable(str);
        let total = this.dataset.total_ham;

        for (let word in this.dataset.hammicity) {
            let old_hammicity = this.dataset.hammicity[word];

            if (table[word] === undefined) {
                this.dataset.hammicity[word] = (total * old_hammicity) / (total + 1);
            } else {
                this.dataset.hammicity[word] = (total * old_hammicity + 1) / (total + 1);
                delete table[word];
            }
        }

        for (let word in table) {
            this.dataset.hammicity[word] = 1 / (total + 1);
        }
    }
}

```

```

        this.dataset.total_ham = total + 1;
    },

    predict: function(str) {
        let table = this.createTable(str), word, p = {};

        for (word in table) {
            let spammicity =
this.dataset.spammicity[word] || 0;
            let hammicity =
this.dataset.hammicity[word] || 0;

            if (spammicity !== 0 && hammicity !== 0) {
                p[word] = spammicity/(spammicity +
hammicity);
            }
        }

        let eta = 0;

        for (word in p) {
            eta += (Math.log(1 - p[word]) - Math.log(p[word]));
        }

        let p_final = 1/(1 + Math.pow(Math.E, eta));

        return (p_final > 0.5);
    }
};

let arg_length = process.argv.length;

if (arg_length === 2) {
    console.log(`Usage: ${process.argv[0]} ${__filename} {-s|-
h|-p} {filename}`);

    process.exit(0);
}

let actions = ["-s", "-h", "-p"];
let type = actions.indexOf(process.argv[2]);

if (arg_length === 3 || type === -1) {
    console.log(`Incorrect arguments, please run
"${process.argv[0]} ${__filename}" for help.`);
    process.exit(1);
}

classifier.load();

switch (type) {
    case 0:
        for (let i = 3; i < arg_length; i++) {
            let file_contents =
fs.readFileSync(process.argv[i], "utf-8");
            classifier.learnSpam(file_contents);
        }

        classifier.save();
        break;

    case 1:
        for (let i = 3; i < arg_length; i++) {
            let file_contents =
fs.readFileSync(process.argv[i], "utf-8");
            classifier.learnHam(file_contents);
        }

        classifier.save();
        break;

    case 2:
        for (let i = 3; i < arg_length; i++) {

```

```
        let file_contents =
fs.readFileSync(process.argv[i], "utf-8");
        console.log(process.argv[i] + ": " +
classifier.predict(file_contents));
    }
    break;
}
```

## Training the filter

Now that we've developed our filter, we now need to train it. Fortunately, there are already many email datasets available for this purpose. We'll be using the [CSDMC2010 dataset](#) in this article.

Download the ZIP archive attached to the file and extract it. We'll do this through the command line, although you're free to do this through a GUI if that's what you're comfortable with.

```
$ wget -O CSDMC2010_SPAM.zip "http://csmining.org/index.php/spam-
email-datasets-.html?
file=tl_files/Project_Datasets/task2/CSDMC2010_SPAM.zip"
$ unzip CSDMC2010_SPAM.zip
```

The dataset contains a set of training files and labels (spam/ham) and a set of test files containing emails. These emails are contained in EML files, and we need to extract the bodies of these emails. In order to do this, you should pull in the `mailparser` module with:

```
$ npm install mailparser
```

Next, save the script below as `eml2txt.js`. This script takes in the name of a directory containing EML files, decodes them and saves them in another directory.

```
#!/usr/bin/env node

"use strict";

let arg_length = process.argv.length;

if (arg_length === 2) {
    console.log(`Usage ${process.argv[0]} ${__filename} {eml-
dir} {txt-dir}`);
    process.exit(0);
}

if (arg_length !== 4) {
    console.log(`Incorrect arguments, run "${process.argv[0]}
${__filename}" for help.`);
    process.exit(1);
}

const fs = require("fs");
const path = require("path");
const simpleParser = require('mailparser').simpleParser;

try {
    let res = fs.statSync(process.argv[3]);
```

```

        if (!res.isDirectory()) {
            console.log(`${process.argv[3]} is not a
directory.`);
            process.exit(1);
        }
    }
    catch (e) {
        fs.mkdirSync(process.argv[3]);
    }

    fs.readdir(process.argv[2], function(err, files) {
        if (err) {
            console.log(`Failed to open input directory:
${process.argv[2]}`);
            process.exit(1);
        }

        let l = files.length;

        for (let i = 0; i < files.length; i++) {
            let file = files[i];
            let abs_file = path.join(process.argv[2], file);

            fs.stat(abs_file, function(err, stat) {
                if (err) {
                    console.log(`Error reading file:
${abs_file}`);
                    return;
                }

                if (stat.isFile(abs_file)) {
                    fs.readFile(abs_file,
function(err, content) {
                        if (err) {
                            console.log(`Error
reading file: ${abs_file}`);
                            return;
                        }

                        simpleParser(content,
function(err, mail) {
                            fs.writeFile(path.join(process.argv[3], file), mail.html ||
mail.text, function(err) {
                                if (err) {
                                    console.log(`Error writing file: ${file}`);
                                }
                            });
                        });
                    });
                }
            });
        }
    });
}
});

```

Let us now convert the training and test EML files to text files.

```

$ node eml2txt.js CSDMC2010_SPAM/CSDMC2010_SPAM/TESTING/ testing/
$ node eml2txt.js CSDMC2010_SPAM/CSDMC2010_SPAM/TRAINING/
training/

```

You'll find two new directories, `training` and `testing`. These folders contain the decoded files. Now, we can train our filter by using the labels from the

`SPAMTrain.label` file provided. Files with a label of 0 are spam, while those with a label of 1 are ham.

```
$ sed -nr 's#^0 (.*)#training/\1#p'
CSDMC2010_SPAM/CSDMC2010_SPAM/SPAMTrain.label | xargs node
spamfilter.js -s
$ sed -nr 's#^1 (.*)#training/\1#p'
CSDMC2010_SPAM/CSDMC2010_SPAM/SPAMTrain.label | xargs node
spamfilter.js -h
```

The training might take some time. When it finishes, you'll find a file, `training.json`, which contains the `dataset` object, saved by the call to `classifier.save()`.

## Testing the filter

Now that you've trained the filter, it's now time to test it. You can test it out on random emails with:

```
$ node spamfilter.js -p $(printf "testing/TEST_%05d.eml" $((RANDOM
% 4292)))
```

Feel free to test this on a few messages. You'll find that the filter gets it right most of the time. If we train it on a more comprehensive dataset, the results would improve. In addition, you can also play around with the threshold value of 0.5 in `predict()` and see if it improves things.

## Improving the design

To keep things easy to understand, we've kept our spam filter simple. However, we could add a variety of bells and whistles to make it more accurate. For example, in an actual implementation, we could assume a higher value of  $P(S)$  if the email originates from an IP address known for sending out spam prolifically.

Again, our filter is not resilient to letter substitutions that a spammer could make. For example, the spammer could write “replica” as “replīca”. Humans can still read it, but our implementation would have trouble detecting this as a word. However, even with a filter that can detect it as a word, “replīca” might still be considered different than “replica”. An actual implementation could detect words with such mixed character sets and automatically assign them high spammicity values, even though the modified word has never been seen before.

There are many more ways a determined spammer could work their way around our filter. For example, they could write their spam messages like so:

```
<p><!-- A short spam message --></p>
<div style="display: none;"><!-- The entire Wikipedia article
about horses.--></div>
```

The `<div>` containing the article about horses is hidden from the user, but it can be still seen by the filter. Unfortunately, due to the large amount of legitimate text, the value of  $p$  would be low, and the message would be marked as ham. A robust spam filter would probably have its own HTML and CSS parser, remove invisible regions from the text, and find out  $p$  for the remaining text.

## Conclusion

This was a really long article. If you've made it this far: congratulations on building your first machine learning based spam filter! Let that sink in — you designed an algorithm and showed it examples of spam and ham messages. After a bit of training, the algorithm has learned how to distinguish between them!

It is no wonder thus that machine learning is making inroads everywhere. Performing tasks without the need for programming things explicitly is what makes machine learning so powerful.

 AdChoices

[Python Learning Course](#)


[Spam Filter](#)

If you liked this post, please share it 😊



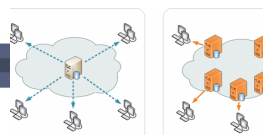
Tags: [machine learning](#)

### YOU MAY ALSO LIKE...




A Guide to SSH Port Forwarding/Tunnelling

UPDATED FEBRUARY 10, 2018



What is a CDN and How to Use It

FEBRUARY 27, 2017



Customizing and coloring the bash prompt

UPDATED FEBRUARY 20, 2018



[Comments](#) [Community](#) [1 Login](#) ▾[♥ Recommend](#)[🐦 Tweet](#)[f Share](#)[Sort by Best](#) ▾

LOG IN WITH

OR SIGN UP WITH DISQUS [?](#)**19975** • 22 days ago

Hi, I got stuck on this part

If a word  $w$  is only in one email,  
which happens to be spam,  $P(S|w)$   
should be 1.

But when you are computing eta:  
 $\text{eta} += (\text{Math.log}(1 - p[\text{word}]) -$   
 $\text{Math.log}(p[\text{word}]))$



[Privacy Policy](#) | [Terms of Service](#) | Copyright © 2018 booleanworld.com. All  
rights reserved.

