ALGORITHMIA

Five Algorithms

EVERY WEB DEVELOPER CAN USE & UNDERSTAND

By Sheldon Kreger

Table of Contents

Welcome to 5 Algorithms Every Web Developer Can Use and Understand!

We have compiled a short primer for web developers on how and when to harness the power of algorithms in your website or other web applications. Each chapter features one algorithm available with the Algorithmia API. Using this tool, developers with limited background in mathematics can access these amazing utilities. In fact, only basic, high-school level mathematics is used in this book. However, even if you are an advanced mathematician, the examples may still prove to be useful, because they clearly display how to use the API within your application.

The world of algorithms is as endless as it is fascinating. Working as a programmer encourages us to pick up new tools and bring them into our applications. However, if we never take the time to explore the areas we are not familiar with, then we could be missing opportunities to write better code. I believe that the five algorithms featured here are some of the most practical for web engineers to use in their applications. Harnessing the Algorithmia API takes these complex challenges - related to both computation and infrastructure - and turns them in to low hanging fruit.

PageRank

What is it?

PageRank is an algorithm used primarily for rating the popularity of web pages. Although one may initially think 'Page' refers to the web page, it actually refers to the inventor, Larry Page (a founder of Google). This is why you'll find the capital 'P' wherever you find a reference to the algorithm.

The basic assumption is that the more inbound links a web page has across the web, the more valid the content the page contains. It's a crowd-sourcing algorithm of sorts; it relies on everybody online to create a network of links to different web pages, and classifies the validity of each page based on its overall popularity.

Applications

The most obvious application of PageRank is the Google search engine. In fact, much of the company's initial success may be attributed to the effectiveness of PageRank in organizing search results on Google.

It is important to note, however, that it is not only web pages which may use PageRank. Any data which can be modeled as a directional graph can be analyzed with PageRank.

The Math

Consider a directional graph, where web pages are modeled as nodes, and the links are modeled as vertices. Each vertex represents a link from one page to another.

While it may be tempting to simply count the total number of links pointing to each web page to discover which page is the most popular, PageRank is a bit more sophisticated.

We begin by assigning each node a score between zero and one. This score represents a probability distribution. If you are unfamiliar with probability distributions, don't worry. This example will make sense regardless.

The initial score is:

```
PR(N) = 1/n
```

This reads as "The PageRank of each node is one divided by the total number of nodes in the network." Each node starts with the same initial score.

Let's use an example network with five nodes: A, B, C, D, and E. In this case, each node has an initial score of 0.2. Now imagine a link:

```
A -> C
```

This would transfer all of A's scores to C, leaving A with 0 and C with 0.4.

Imagine A with two outbound links:

```
A -> C
A -> D
```

In this case, A would still have 0, but C and D would split the 0.2 from A. This leaves D with 0.3 and C with 0.3.

Be aware that this is an example of a simple Page Rank algorithm. More advanced implementations will use tools such as a damping factor.

Example 1: Calculating Simple PageRank on an Array of Outbound Links

In this example, we will use Node.js to make our request. Keep in mind you may perform any Algorithmia API request with any tools at your disposal. See Appendix A to learn how to configure your machine for development with the Algorithmia API in Node.js.

```
var algorithmia = require("algorithmia");
var client = algorithmia(process.env.ALGORITHMIA_API_KEY);
var input = [[1,2,3],[0],[0],[0,4,5,6,7],[],[],[],[]];

client.algo("thatguy2048/PageRank/0.1.0").pipe(input).then(function(output) {
    if (output.error) {
        console.log(output.error);
    } else {
      console.log(output.result);
    }
});
```

The input for the algorithm call is a representation of the outbound links. If we describe them as such:

> page 0 links to pages 1, 2, 3
> page 1 links to page 0
> page 2 links to page 0
> page 3 links to pages 0, 4, 5, 6, 7
> page 4 has no links
> page 5 has no links
> page 6 has no links
> page 7 has no links

we can translate the outbound links into the input format. The first object in the array is `[1,2,3]`, representing page 0's outbound links while the last object is `[]`, representing that page 7 has no outbound links. This is how we end up with an input like so:

```
var input = [[1,2,3],[0],[0],[0,4,5,6,7],[],[],[],[]];
```

This particular call will return an array of PageRank scores:

```
[ 0.9142543787391981,
0.4089334112608257,
0.4089334112608257,
0.4089334112608257,
0.21947767079447258,
0.21947767079447258,
0.21947767079447258,
0.21947767079447258 ]
```

As you can see, the first result has the highest score because it had the most inbound links of all the pages.

Note that the API key and input data should be modified to match your needs.

Example 2: Calculating PageRank for a Real Website Using the URL

Of course, it is not convenient to construct an array of outbound links as shown above if you want to ascertain the PageRank of a real website. The Algorithmia team has conveniently implemented a complete tool to calculate PageRank on all the pages of a website with a single API call.

The PageRank algorithm for the web calculates the PageRank based on the internal links within the input domain. It does not consider links from outside this source. Because this algorithm must complete a traversal of the site, the first run of the algorithm may take a few

moments to complete.

```
var algorithmia = require("algorithmia");
var client = algorithmia(process.env.ALGORITHMIA_API_KEY);
var input = 'https://your-domain.com';

client.algo("/web/PageRank/0.1.0").pipe(input).then(function(output) {
    if (output.error) {
      console.log(output.error);
    } else {
      console.log(output.result);
    }
});
```

Try running the script on several different URLs as the input string. The return value `output.result` will be in the form of an array containing values such as:

```
'url' : score
```

The `'url'` will be unique to the page it is ranking, and the score will be represented as a number between 0 and 1. You'll also see that the results are ranked starting with the URL with the highest PageRank, followed by the rest of the URLs in decending order.

Language Detection

What is it?

A language detection algorithm is pretty self-explanatory: we'll take text as input and decide which human language the text is written in. We'll be using Algorithmia's Language Identification algorithm to give it a try.

Applications

In Natural Language Processing (NLP), one may need to work with data sets which contain documents in various languages. Many NLP algorithms only work with certain languages, usually because the training data they rely upon is in a single language. It can be a valuable time saver to determine which language your data set is in before you run more algorithms on it.

A second example application of the Language Detection algorithm lies in the web search arena. A web crawler will hit pages which are potentially written in one of many different languages. If this data is to be used by a search engine, the results are going to be most helpful to the end user if the language used in the search is the same as the results. You can easily see how a web developer who must work with content in multiple languages would want to implement language detection as part of a search functionality.

Spam filtering services which support multiple languages must be able to identify the language that emails, online comments, and other input is written in before the true spam filtering algorithms can be applied. Without such detection, content originating from specific countries, regions, or areas suspected of generating spam cannot be properly eliminated from online platforms.

The Math

Language classifications rely upon using an primer of specialized text called a 'corpus'. There is one corpus for each language the algorithm can identify. Speaking in summary, input text is compared to each corpus and pattern matching is used to identify the strongest correlation to a corpus.

Because there are so many potential words to profile in every language, computer scientists use algorithms called 'profiling algorithms' to create a subset of words for each language, to be used for the corpus. The most common strategy is to choose very common words. In English, we might choose words like "the", "and", "of" and "or".

This approach works well when the input data is relatively lengthy. The shorter the phrase in the input text, the less likely these common words are to appear, and the less likely the algorithm will classify correctly. In fact, some languages don't have spaces between written words, making such isolation impossible.

Facing this problem, researchers tried to use character sets in a general manner, rather than relying on them being split into words. Even if the words have spaces between them, relying on the natural words alone often causes problems when analyzing short phrases.

One is then left to experiment with how many letters to analyze. The N-gram algorithm implemented by Apache Tika uses 3 letters, and is therefore called a 3-gram. This algorithm is available conveniently through the Algorithmia API with a single HTTP request.

Example: Identifying Language of a Single String

Making an Algorithmia API Request Using Node.js

In this example, we will use Node.js to make our request. Keep in mind you may perform any Algorithmia API request with any tools at your disposal. See Appendix A to learn how to configure your machine for development with the Algorithmia API in Node.js.

```
var algorithmia = require("algorithmia");
var client = algorithmia(process.env.ALGORITHMIA_API_KEY);
var input = "This is a demo sentence for language detection.";

client.algo("/nlp/LanguageIdentification/0.1.0").pipe(input).then(function(output) {
    if (output.error) {
        console.log(output.error);
    } else {
        console.log(output.result);
    }
});
```

Note that the API key should be modified to match your configuration.

This will return the string `en` to indicate English. You may read more about the implementation and find a list of supported languages on the algorithm's page.

Nudity Detection

What is it?

Nudity Detection takes an image as an input and returns a value telling us whether the image contains nudity, and a confidence level for this prediction. Because there may be false positives and negatives, adjusting the confidence level allows us to control how strict we want filtered results to be.

This particular algorithm operates under a few assumptions. First, it assumes that nude images contain many skin colored pixels, and that these pixels are mostly interconnected. Unbroken regions of skin colored pixels increase the probability that an image is nude. Second, it assumes that the nude regions are indeed skin colored, rather than black and white. Therefore, black and white nude images are not correctly identified by this algorithm.

Applications

Countless websites allow users to upload images, which are then displayed publicly on the site. In many cases, the task of manually reviewing every image upload can be too great for a team to handle. Even when this is possible, users may not want to wait for a manual review process every time they publish new content online. Not only is this method cumbersome for both the administrators and the users, but it serves as a deterrent to active user participation in content generation.

Think of an internet forum, where users have ‘avatars’ displayed in their posts. Or perhaps users are allowed to write blog posts and can attach images inside them. If this content is immediately public-facing, a web team might want to prevent users from publishing content they wish to restrict, such as pornographic images. This is also effective in preventing bots that spam websites with said pornographic images from flooding websites with advertisements and images, which can harshly impact the quality of an online community.

The Math

While most image classification algorithms work by utilizing training data, then comparing fresh input images with the pre-categorized data set, this method has a few drawbacks. The main issue is that it requires human time and effort to gather training data, and then categorize hundreds (sometimes thousands) of images by hand.

A newer method was recently published in the academic paper An Algorithm for Nudity Detection, which outlines a different way of tackling the nudity problem, specifically. The Algorithmia API utilizes a variation of this technique, which we will explore in more detail.

The primary characteristic this algorithm relies on is skin color. Analyzing the colors contained in an image is not only effective, it is computationally inexpensive. The main goal is to differentiate between pixels in an image which are skin-toned, and those which are not. Nudity detection models exist for all common skin colors. Furthermore, image analysis techniques today can compensate for differences in luminescence and focus strictly on color. This increases the accuracy when the algorithm classifies each pixel as either skin-toned or not.

There is a four step process, outlined in the paper:

1. Detect skin-colored pixels in the image.
2. Locate or form skin regions based on the detected skin pixels.
3. Analyze the skin regions for clues of nudity or non-nudity.
4. Classify the image as nude or not.

The first step uses an 'skin color distribution model' to analyze the image pixel by pixel to see if each matches a known skin color. The algorithm then performs sophisticated analysis of the percentage of skin colored regions, their relative size, and their relative locations. Many factors are taken into account and ultimately a boolean value is returned, along with a confidence score.

The implementation in the Algorithmia API also utilizes face detection to help eliminate false positives. Face detection methods are not within the scope of this chapter, but you can learn more about face detection by looking at the face detection algorithms on Algorithmia.

Example: Censoring Online Photos

Step 1: Make Algorithmia API Request Using Node.js

In this example, we will use Node.js to make our request. Keep in mind you may perform any Algorithmia API request with any tools at your disposal. See Appendix A to learn how to configure your machine for development with the Algorithmia API in Node.js.

```
var algorithmia = require("algorithmia");

var client = algorithmia(process.env.ALGORITHMIA_API_KEY);
var input = "http://www.lenna.org/full/len_full.jpg";

client.algo("sfw/NudityDetection/1.0.x").pipe(input).then(function(output) {

    if (output.error) {
        console.log(output.error);
    } else {
        console.log(output.result);
    }
});
```

As you can see in the above code sample, the input that we are sending to the algorithm through the API call is a URL to an image. Replace this with the URL to any image of your liking to get a feel for how the algorithm works. You should see that the `output.result` will return: `{ nude: 'true', confidence: 1 }`. You can then evaluate the JSON to determine inside your application about whether or not the image is nude and how confident you are.

Note that the API key should be modified to match your data.

Step 2: Decide a Confidence Level for Filtering

Let's say you want to be extra careful when evaluating images for nudity. You can change the change the threshold for the confidence level, allowing you a convenient way to mark some images as "Uncertain" or "Needs review". This way you can let images that are above the confidence level threshold through to your application, while noting which images might need moderation before posting.

```
var algorithmia = require("algorithmia");

var client = algorithmia(process.env.ALGORITHMIA_API_KEY);
var input = "http://www.lenna.org/full/len_full.jpg";

client.algo("sfw/NudityDetection/1.0.x").pipe(input).then(function(output) {

  if (output.error) {
    console.log(output.error);
  } else {

    var result = output.result;
    confidence = result.confidence;

    if (confidence < 0.85) {
      console.log("Uncertain");
    } else if (result.nude) {
      console.log("Nude");
    } else {
      console.log("Not nude");
    }
  }
});
```

Here we have an example where we've parsed out the confidence level from the algorithm's result, then added in a conditional check. Where `confidence < 0.85`, we can take special action such as adding the image to a moderation queue or adding a warning that the content might contain nudity.

By using the Nudity Detection algorithm and adjusting the threshold to our needs, you can easily see how valuable the algorithm can be for web developers. It allows you another layer of safety and control when working with applications that include user-uploaded content.

Sentiment Analysis

What is it?

People who speak a language can easily read through a paragraph and quickly identify whether the writer had an overall positive or negative impression of the topic at hand. However, for a computer, which has no concept of natural spoken language, this problem must be reduced to mathematics. Without any context of what words actually mean, it cannot simply deduce whether a piece of text conveys joy, anger, frustration, or otherwise. Sentiment analysis seeks to solve this problem by using natural language processing to recognize keywords within a document and thus classify the emotional status of the piece.

Applications

Businesses today often seek feedback on their products and services. Before online content and social media data became abundant, companies would ask for direct feedback from their customers in a variety of ways. They may have used hand written forms submitted on-location or via mail. Or, they may have hired a telephone survey company to call customers and ask questions directly. Today companies can mine online data to gain insight on customer sentiment of their products and services. Use of sentiment analysis algorithms across product reviews lets online retailers know what consumers think of their products and respond accordingly.

Sociologists and other researchers can also use this kind of data to learn more about public opinion. A great example is MemeTracker, an analysis of online media about current events. Political organizations often want to understand people's overall opinion of a particular politician or political conundrum in order to develop a strategy during election season. Comment sections on news websites are frequent targets for said groups, as people who take time to respond are prone to be more politically engaged than other citizens. This in turn serves as a form of low-cost, soft polling.

The Math

Basic sentiment analysis algorithms use natural language processing (NLP) to classify documents as positive, neutral, or negative. Programmers and data scientists write software which feeds documents into the algorithm and stores the results in a way which is useful for

clients to use and understand.

Keyword spotting is the simplest technique leveraged by sentiment analysis algorithms. Input data is scanned for obviously positive and negative words like 'happy', 'sad', 'terrible', and 'great'. Algorithms vary in the way they score the documents to decide whether they indicate overall positive or negative sentiment. Different algorithms have different libraries of words and phrases which they score as positive, negative, and neutral. When exploring these algorithms, you might run into the nickname for these libraries of words: "bag of words".

This technique has a few shortcomings. Consider the text, "The service was terrible, but the food was great!" This sentiment is more complex than the algorithm can really take into account, because it contains both positive and negative words. More advanced algorithms will split sentences when words like 'but' appear. Such a case is called 'constrastive conjunction'. Such a result then becomes, "The service was terrible" AND "But the food was great!" The sentence thus generates two or more scores, which then must be consolidated. This is called binary sentiment analysis.

Keep in mind that due to the complexity of organic language, most sentiment analysis algorithms are about 80% accurate, at best.

Example: Sentiment Analysis of Twitter Data

Let's build a sentiment analysis of Twitter data to show how you might integrate an algorithm like this into your applications. We'll first start by choosing a topic, then we will gather tweets with that keyword and perform sentiment analyis on those tweets. We'll end up with an overall impression of whether people view the topic positively or not.

Step 1: Gather Tweets

First, choose a topic you wish to analyze. Inside `sentiment-analysis.js`, you can define `input` to be whatever phrase you like. In this example, we'll use a word we expect to return positive results.

```
var algorithmia = require("algorithmia");
var client = algorithmia(process.env.ALGORITHMIA_API_KEY);
var input = "happy";
var no_retweets = [];

console.log("Analyzing tweets with phrase: " + input);
client.algo("/diego/RetrieveTweetsWithKeyword/0.1.2").pipe(input).then(function(output) {
    if (output.error) {
        console.log(output.error);
    } else {
        var tweets = [];
        var tweets = output.result;
        for (var i = 0; i < output.result.length; i++) {
          // Remove retweets. All retweets contain "RT" in the string.
          if (tweets[i].indexOf('RT') == -1) {
            no_retweets.push(tweets[i]);
          }
    }
}

// We will cover this function in the next step.
analyze_tweets(no_retweets);
});
```

As you can see, first we use the Algorithmia API to pass our topic to the algorithm RetrieveTweetsWithKeyword as our input. This will grab tweets containing our phrase. Second, we clear out the retweets so that we don't have duplicate data throwing off our scores. Twitter conveniently includes "RT" at the beginning of each tweet, so we find tweets with that string and remove them from our data set. This leaves us with a convenient set of tweets in the array `no_retweets`.

Step 2: Perform Sentiment Analysis on Tweets

After gathering and cleaning our data set, we are ready to execute the sentiment analysis algorithm on each tweet. Then, we will calculate an average score for all the tweets combined.

```
var analyze_tweets = function(no_retweets) {
    var total_score = 0;
    var score_count = 0;
    var final_score = 0;

    // Execute sentiment analysis on every tweet in the array, then calculate average sco
    for (var j = 0; j < no_retweets.length; j++) {
        client.algo("nlp/SentimentAnalysis/0.1.1").pipe(no_retweets[j]).then(function(out
        if(output.error) {
            console.log(output.error);
        } else {
            console.log(output.result);
            score_count = score_count + 1;
            total_score = total_score + output.result;
        }
        // Calculate average score.
        if (score_count == no_retweets.length) {
            final_score = total_score / score_count;
            console.log('final score: ' + final_score);
        }
    })
  }
}
```

In the above code, we iterated through each tweet in `no_retweets` to send that as input to the Sentiment Analysis algoritm. Then with the results from that API call, we added the output result to a `total_score` variable. We keep track of how many tweets we've gone through with the variable `score_count`, so that when it reaches the same number as the number of tweets we wanted to analyze we then calculate the final score by averaging the `total_score`. This final result returns a number in the range [0-4] representing, in order, very negative, negative, neutral, positive, and very positive sentiment.

TF-IDF: Term Frequency-Inverse Document Frequency

What is it?

TF-IDF (Term Frequency-Inverse Document Frequency) is a text mining technique used to categorize documents. Have you ever looked at blog posts on a web site, and wondered if it is possible to generate the tags automatically? Well, that's exactly the kind of problem TF-IDF is suited for.

It is worth noting the differences between TF-IDF and sentiment analysis. Although both could be considered classification techniques for text, their goals are distinct. On the one hand, sentiment analysis aims to classify documents into opinions such as 'positive' and 'negative'. On the other hand, TF-IDF classifies documents into categories inside the documents themselves. This would give insight about what the reviews are about, rather than if the author was happy or unhappy. If we analyzed product review data from an e-commerce site selling computer parts, we would end up with groups of documents about 'laptop', 'mouse', 'keyboard', etc. We would gain a large amount of data about the types of reviews that had been written, but would not learn anything about what the users thought of those products. Although the algorithms are similar in that they classify text, the results of each give us unique insights.

Applications

This algorithm is useful when you have a document set, particularlly a large one, which needs to be categorized. It is especially nifty because you don't need to train a model ahead of time and it will automatically account for differences in lengths of documents.

Imagine a large corporate website with tens of thousands of user contributed blog posts. Depending on the tags attached to each blog post, the item will appear on listing pages on various parts of the site. Although the authors were able to tag things manually when they wrote the content, in many cases they chose not to, and therefore many blog posts are not categorized. Empirics show that only a small fraction of users will take the time to manually add tags and assist with categorization of posts and reviews, making voluntary organization unsustainable. Such a document set is an excellent use-case for TF-IDF, because it can generate tags for the blog posts and help us display them in the right areas of our site. Best

of all, no intern would have to suffer through manually tagging them on their own! A quick run of the algorithm would go through the document set and sort through all the entries, eliminating a great deal of hassle.

The Math

TF-IDF computes a weight which represents the importance of a term inside a document. It does this by comparing the frequency of usage inside an individual document as opposed to the entire data set (a collection of documents).

The importance increases proportionally to the number of times a word appears in the individual document itself--this is called Term Frequency. However, if multiple documents contain the same word many times then you run into a problem. That's why TF-IDF also offsets this value by the frequency of the term in the entire document set, a value called Inverse Document Frequency.

```
TF(t) = (Number of times term t appears in a document) / (Total number of terms in the do
IDF(t) = log_e(Total number of documents / Number of documents with term t in it).
Value = TF * IDF
```

TF-IDF is computed for each term in each document. Typically, you will be interested either in one term in particular (like a search engine), or you would be interested in the terms with the highest TF-IDF in a specific document (such as generating tags for blog posts).

Example: Tagging Blog Posts

Step 1: Generate Scores for Each Document

Let's say you have a 100 word blog post with the word "JavaScript" in it 5 times. The calculation for the Term Frequency would be:

```
TF = 5/100 = 0.05
```

Next, assume your entire collection of blog posts has 10,000 documents and the word "JavaScript" appears at least once in 100 of these. The Inverse Document Frequency calculation would look like this:

```
IDF = log(10,000/100) = 2
```

To calculate the TF-IDF, we multiply the previous two values. This gives us the final score:

```
TF-IDF = 0.05 * 2 = 0.1
```

Step 2: Decide a Threshold to Tag

After running this algorithm against all 100 of the blog posts with the word "JavaScript", you end up with a score for each.

This is where you will have a chance to exercise the creative aspect of being a data scientist. Let's assume that you have a wide range of scores, ranging from 0.05 to 0.5.

Continuing a simple example from this collection, 0.05 would be a 100 word document with 1 instance of "JavaScript" and 0.5 would be a 100 word document with "JavaScript" appearing 25 times. To determine if the document will be tagged with "JavaScript", you need to decide on a threshold score.

The score you choose will vary depending on your data set. A document with only one instance of "JavaScript" (score 0.05) is unlikely to be focused on JavaScript, but obvioulsy the high score of 0.5 is probably on topic.

TF-IDF Code Example Using Node.js

See Appendix A to learn how to configure your machine for development with the Algorithmia API in Node.js.

The Keywords For Document Set algorithm implements TF-IDF for a function to retrieve keywords in a document set. This function expects a set of documents, each as a separate string, and an integer to define how many keywords to return.

The following example passes three documents and returns the top two keywords from each document, including the TF-IDF score:

```
var algorithmia = require("algorithmia");
var client = algorithmia(process.env.ALGORITHMIA_API_KEY);

var input = [["badger badger buffalo mushroom mushroom mushroom mushroom mushroom mushroo

client.algo("nlp/KeywordsForDocumentSet/0.1.7").pipe(input).then(function(output) {
  if (output.error) {
      console.log(output.error);
  } else {
      console.log(output.result);
  }
});
```

This returns:

```
[ { mushroom: 0.5187490272120597, badger: 0.8078365072138199 },
{ antelope: 0.47712125471966244, buffalo: 0.17609125905568124 },
{ bannana: 0.47712125471966244 } ]
```

As you can see, the algorithm calculated the frequency of each word. If you do the math, you'll find that these TF-IDF scores are not just a simple average, but are weighted as explained in this chapter. Note that because the last document only contained one value, the return value for the final document only contains one value.

Conclusion

While we couldn't cover all of our favorite algorithms, we hope you've found the explanations of some common algorithms in this book helpful. We believe that web developers of all kinds, from the formally trained to the self-taught, can benefit greatly from the use of algorithms in their websites and web applications. With Algorithmia, we try to let the power of algorithms work for you through a simple API so that you can harness their power without investing time & energy learning to write the complicated algorithms yourself. Now you can make MVPs, iterate on your products, and leverage complex computations quicker than ever.

What's next?

We'd love to hear from you! If you've found this book useful, please head on over to Algorithmia and check out some of the other algorithms we offer. You can also find more examples of Algorithmia algorithms at work on our blog.

If you've built something using the Algorithmia API, drop us a line and let us know! We'd love to feature your work on our blog.

Find a bug in the code? Can't get the examples working? Submit an issue on the book's repository and we'll help you get things up and running correctly.

Appendix A: Setting Up A Development Environment

This section explains how to set up a development environment to utilize the Algorithmia API using Node.js. Whether you are using the API in a larger project, or simply working through the examples in this book, the setup process is the same.

The Algorithmia NPM Package

Algorithmia provides seamless access to all algorithms in the API library. If you are familiar with Node.js, using the Algorithmia package is effortless. If you are less familiar with Node.js, read on to the next section to continue the setup process.

You can find the package documentation on the official npm page.

Simply run `npm install --save algorithmia` in your terminal from the project root folder.

Alternatively, you can add the package dependency in your package.json file:

```
{
    "dependencies": {
        "algorithmia": "^0.2.1"
    }
}
```

Then run `npm install` from your project root folder.

The package contains a folder called `examples` where you will find more demonstrations.

Configure Algothmia API Key

To utilize the Algorithmia API, you will need to set up an account and configure your environment to use your API key.

Visit Algorithmia.com to set up a new account. Your API key is available on your account credentials page. Find it by navigating to your user dashboard or by replacing "your-username" in the url below with the username you chose:

https://algorithmia.com/users/your-username#credentials

The scripts used in this book use an environment variable to access the API key. Once you know your API key, there are two ways to use it in your scripts. The first way is to simply pass the API key as an argument when you invoke the Node.js script.

```
ALGORITHMIA_API_KEY=your-key node your-script.js
```

In Unix systems, you may define `ALGORITHMIA_API_KEY` as an environment variable. This means you don't have to pass it to the scripts every time.

Place this at the bottom of `.bashrc`, usually located at `~/.bashrc`:

```
# Algorithmia API Key
export ALGORITHMIA_API_KEY=your-key
```

Now, when you invoke the Node.js scripts, they will be able to access `process.env.ALGORITHMIA_API_KEY` on their own. Simply run the script:

```
node your-script.js
```

Verify it Works

Run any of the example scripts in this book's repository to verify your environment is set up correctly. You can also find more support through the official documentation.