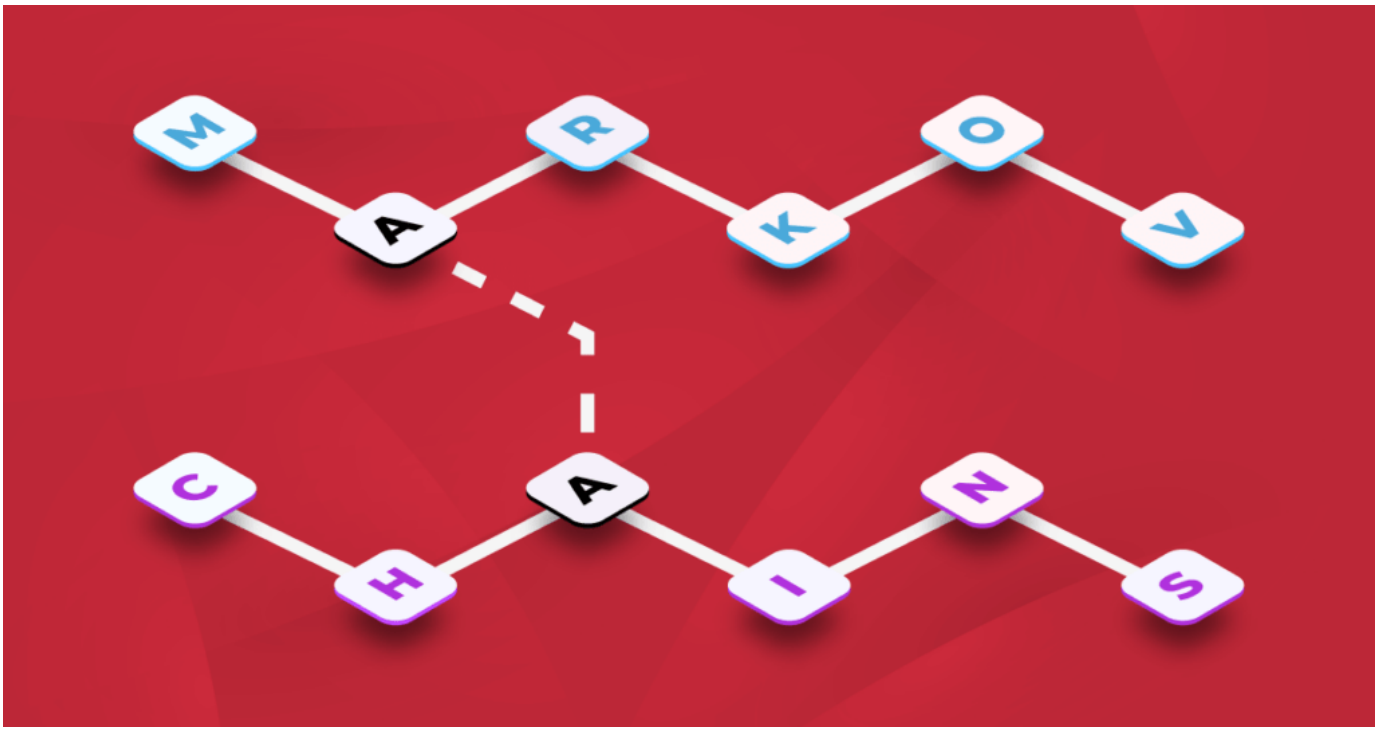
sitepen

Exploring the Creative Possibilities of Markov Chains for Text Generation

Rory Mulligan | February 23, 2023



AI ALGORITHMS MARKOV CHAINS

Introduction

News about AI is everywhere right now. Whether it's ChatGPT or Stable Diffusion, we're using AI algorithms to generate text and images. Before the recent explosion of neural networks though, people have been using computers as word smiths for decades. Today we are going to look at an algorithm, Markov chains, and learn how we can use it for our own simple texts. Markov chains are often used for text generation, generating anything

from poetry to fiction. We'll start exploring the use of Markov chains in a very simple setting by creating our own fictional text.

You can view the final project, or follow along and build your own as we go.

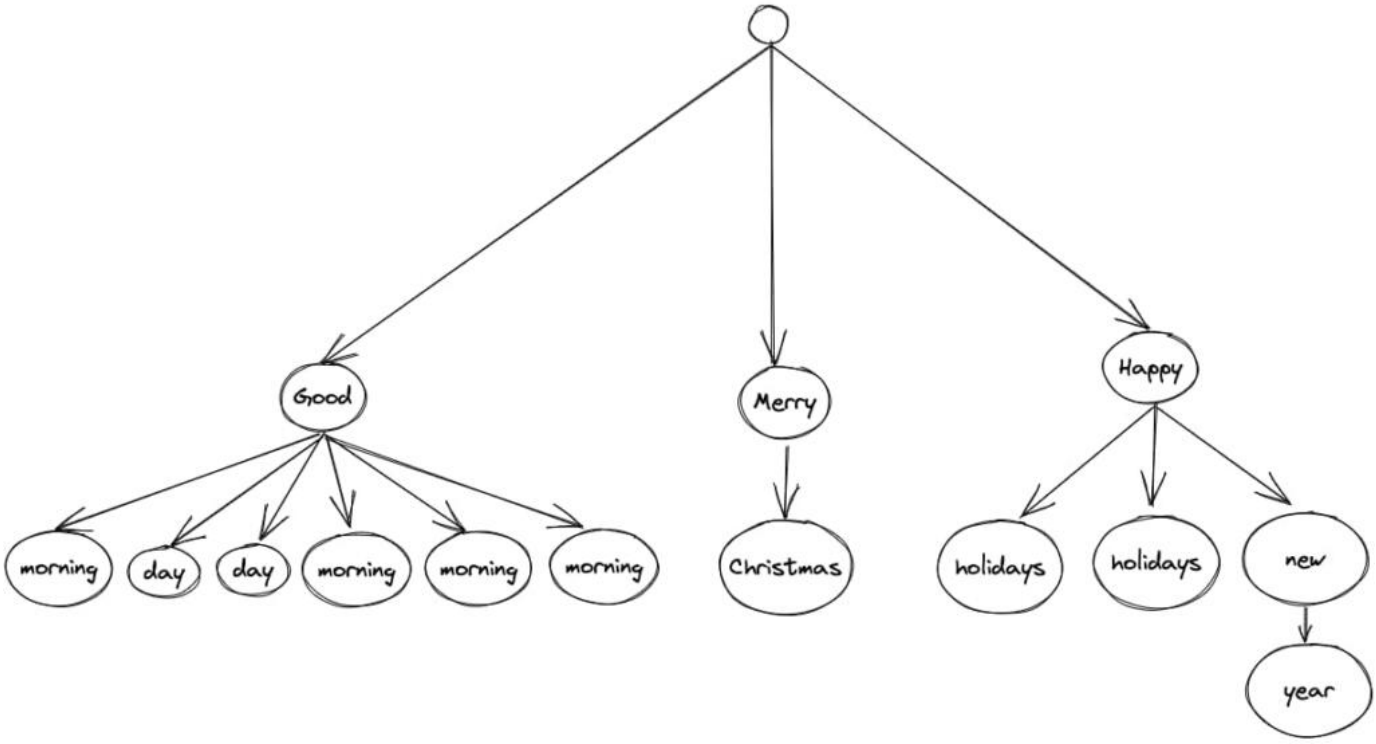
What are Markov chains?

Markov chains are a method for predicting sequential data and their use spans multiple disciplines. As such, reading about Markov chains can be intimidating as explanations tend to descend quickly into theory and statistical jargon. Let's set all of that aside and try to take a more intuitive approach...

Let's say you are a normal lizard-person, ready for his turn to live a normal secret life on Earth. Your covert lizard-scouts have already compiled a list of standard earth greetings by interacting with 10 different real humans (during the holiday season).

1. Good morning
2. Good day
3. Merry Christmas
4. Good morning
5. Good morning
6. Good day
7. Happy holidays
8. Happy New Year
9. Happy holidays
10. Good morning

We can visualize the greetings by connecting each word together in a graph according to how they appear in the greeting.

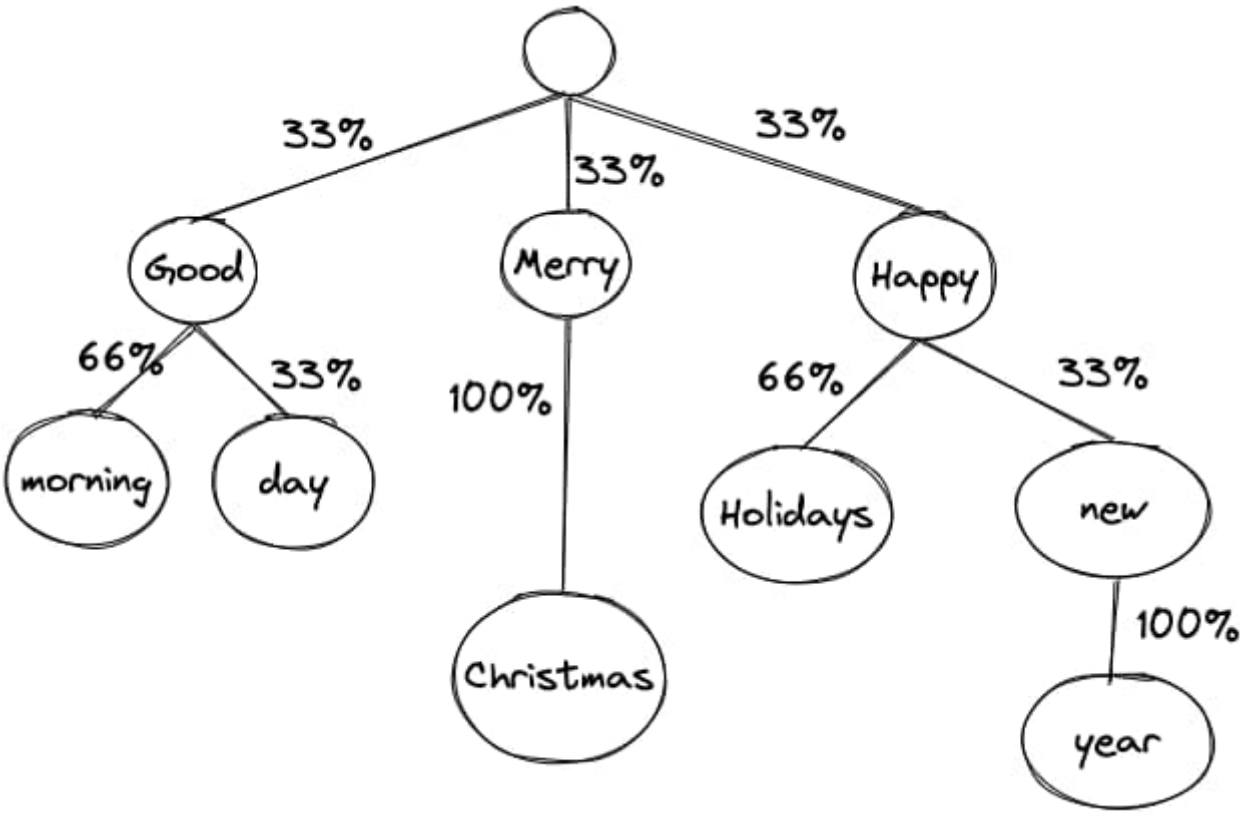


Now, in order to pass yourself off as a normal human being, you are going to have to give some normal human greetings, but not the same one every time, as that would be abnormal.

If we start at the top of our tree and randomly pick any child node, we should come up with an acceptable, though varied, greeting. Going through that exercise we can make a few conclusions:

- There is a 1/3 chance to start with “good”, “merry”, or “happy” respectively.
- If we say “good”, there is a 1/3 chance we should say “day” and a 2/3 chance we should say “morning”.
- If we say “merry”, we should say “Christmas”.
- If we say “happy”, there is a 2/3 chance we should say “holidays” and a 1/3 chance we should say “New Year”.

If we redraw that graph to display those percentages, we get a new graph.



That's basically a Markov chain. For every node, there is an X% chance that you will go to one node or a Y% chance you will go to another node.

> *Note: We've drawn lines here to make it look like a tree, but more complicated Markov chains are not tree structures at all, as nodes can have paths to any other node, including themselves.*

Text Generation

The use case of Markov chains that we are interested in today is that of text generation, but to do so, we need to give the algorithm some text input to work with. This is similar to how we generated human greetings, but at a much larger scale. Instead of having 10 input strings, we'll now have 100,000. Before we can generate random text, we'll need to accomplish a few different things.

- Find a text source
- Tokenize the text
- Build the Markov chain
- Traverse the Markov chain

Find a text source

You can build a Markov chain using whatever text source you want, but in general, the more text, the better. More text means more nodes in the graph, which means more possibilities in your output. Remember, a Markov chain is only ever going to pick words that it has seen before, so, the more words in more places, the more varied and exciting our results are going to be.

You can choose any text source, but some texts are better than others. We'll be generating our text in "blocks." A block can be whatever you want, as long as it's convenient for you to work with. A block could be a sentence, a text message, or a Tweet, it's totally up to you. I've created Markov chains from books, haikus, and even other peoples' instant messages.

For our task today, I've decided to choose an input text from "The Time Machine" by H.G. Wells. It's got a few things going for it: it's free, it's in plain text format, it's not too long (so it's good for a demo!), and it's a classic! You can go ahead and pick up a copy here. There are loads of other free text resources you can choose from. I've gone ahead and cleaned up the text a bit, by opening up the file and deleting the bits we don't want to parse.

Tokenize the text

Once you have your text source, we need to tokenize the text. Tokenization is the process of returning things you care about, "tokens," from a stream. In our case, a token would be an individual word contained in our text. There are a lot of ways you can do this, but one of the simplest is to just use a regular expression. We'll first convert our text to blocks by splitting the input by punctuation, then tokenize each block.

```
const wordsRegex = /([a-z']+)|([.!?])/igm;

const START_TOKEN = '<start>';
const END_TOKEN  = '<end>';

export function parseTokens(line: string, nGramLength = 1) {
    return [
        // 1
        ...Array.from(new Array(nGramLength - 1)).map(() => START_TOKEN),
        // 2
            Array.from(line.matchAll(wordsRegex), m => m[0].toLowerCase()).fil
```

```
        ];
}
```

Let's assume for now that `nGramLength` is 1. We'll be using that later, but for now, it's not important.

1. Prefix the token stream with a number of start markers. These will be important later, but for an `nGramLength` of 1, this won't do anything.
2. Split the string by words and convert them to lowercase, removing empty tokens.
3. Add an "end" token. This is important because it will let us know when a text block ends in the input text.

There are a lot of more interesting and smarter ways you can tokenize words, and I would encourage you to look up other approaches that might better meet your needs, such as supporting hyphenated words. This simple approach will do for this example.

Build the Markov chain

Now that we have a stream of tokens, we can store the relationships between the words. We can store each token as a key in our object, with the value being the number of times that key occurs.

For example, given a sentence like, "If you don't challenge yourself, you won't change yourself." We would create an object that looks like this.

```
{
    "if": 1,
    "you": 2,
    "don't": 1,
    "challenge": 1,
    "yourself": 2,
    "won't": 1,
    "change": 1,
    ".": 1,
    "<end>": 1
}
```

Notice how the word "yourself" has a value of 2, since it appears twice. We start by picking a random key from the object, where keys with a higher value have a higher probability of

being selected. We can then continue to choose words using the same rule until we hit an `<end>` token.

```
// 1
export interface MarkovChain {
    [key: string]: MarkovChain | number;
}

export function buildMarkovChain(chain: MarkovChain, tokens: string[], nGramLe
    let queue: string[] = [];
    tokens.forEach((word) => {
        // 2
        if(queue.length < nGramLength) {
            queue.push(word);
        }
        else {
            queue.push(word);
            queue.shift();
        }

        if(queue.length >= nGramLength) {
            // 3
            let root = chain;
            queue.slice(0, -1).forEach((key) => {
                if(!root[key]) {
                    root[key] = {};
                }

                root = root[key] as MarkovChain;
            });

            // 4
            if(!(word in root)) {
                root[word] = 0;
            }

            (root[word] as number)++;
        }
    });
    return chain;
}
```

1. Our Markov chain will be an object of one or more levels of Markov chains. For an nGramLength of 1 this will essentially be a { [key: string]: number }

2. This queue will keep track of where we are in the tree. It will point to the last word picked.
3. We descend the tree based on the history we've kept in the queue. For an `nGramLength` of 1, this will always leave us at the root node. For higher `nGramLength` values, we'll descend into the tree and visit leaf nodes.
4. Increment the number of times this word has occurred and store it in the Markov chain.

Generating text

OK, now the fun part. With our Markov chain created, we can start generating some text! All we need to do is pick a starting word and then pick new words using our stored probabilities until we reach the `<end>` token.

```
// 1
function pickRandom(arr: string[]) {
    return arr[Math.floor(Math.random() * arr.length)];
}

// 2
function expandProbabilitiesToArray(obj: { [key: string]: number }) {
    return Object.entries(obj).reduce((acc, [key, value]) => {
        return acc.concat(Array(value).fill(key));
    }, [] as string[]);
}

export function generateText(chain: MarkovChain, nGramLength = 1) {
    // 3
    const words: string[] = Array.from(new Array(nGramLength - 1)).map(() => S

    // 4
    while (1) {
        let root = chain;
        for (let i = words.length - nGramLength + 1; i < words.length; i++) {
            root = root[words[i]] as MarkovChain;
        }

        const w = pickRandom(expandProbabilitiesToArray(root as Record<string,
        if (w === END_TOKEN) {
            break;
        }
```

```
    }

    let result = words.splice(nGramLength - 1).join(" ");
    // 5
    result = result.replace(/\s([.?!,])/g, '$1');

    return result;
}
```

1. A small utility function to help us pick a random value from an array.
2. A utility function that will take an array of occurrences and expand each key that number of times into an array. There are more clever ways to pick a random key, but this is simple and good enough for our needs. Given an input of `{ one: 1, two: 2 }`, our output would be `['one', 'two', 'two']`.
3. With an `nGramLength` of 1, this won't do anything, but with larger values, this will pre-populate our word chain with starting tokens.
4. We continue to pick new words until we get to the end token.
5. Since we joined all tokens with spaces, our punctuation looks a bit off, as in "Hello World ." This little regex will just remove the white space before punctuation.

OK, great! Let's put this all together and run it a few times.

```
const chain: MarkovChain = {};
text.split(/[\.\?\!]/).forEach(line => {
    const tokens = parseTokens(line);
    buildMarkovChain(chain, tokens);
});

generateText(chain);
```

- her you came said absence strong the i cannot
- huge saw that two my little me up when to the were at
- i it make support intense the engagement in seen strike a

OK, that's almost English. It's definitely got words in it, and that's a good start!

Improving the results

Think back to our lizard-people example, and how the chain was represented in that tree.

whole lot of sense in the way we are putting them together. The thing about the tree that intuitively made sense is that it kept some sense of where we have been, as you traveled down the tree. Well, we can fine-tune our text generator to do something similar with that mysterious `nGramLength` parameter. An "n-gram" is the number of tokens required to construct a key for our chain.

Remember the original "If you don't challenge yourself, you don't change yourself" chain?

```
{
    "if": 1,
    "you": 2,
    "don't": 1,
    "challenge": 1,
    "yourself": 2,
    "won't": 1,
    "change": 1,
    ".": 1,
    "<end>": 1
}
```

Here is what that looks like with an `nGramLength` of 2.

```
{
    "if": {
        "you": 1
    },
    "you": {
        "don't": 2
    },
    "don't": {
        "challenge": 1,
        "change": 1
    },
    "challenge": {
        "yourself": 1
    },
    "yourself": {
        "you": 1,
        ".": 1
    },
    "change": {
        "yourself": 1
    }
```

```
        "<end>": 1
    }
}
```

Notice we've gone another level deep now. Now, the root keys store all of the words that can follow those words, along with their probabilities. When we are generating text, and have come across the word "don't", we will now have only two options, "challenge" and "change." Once we pick one, it will become the new root, and the only option after each of those is "yourself."

Let's re-run our text generator but dial the nGramLength up to 2.

- i was the door at once have shown the morning made of man but in one place and i felt his cigar and mutual tenderness still to the means of my best of what it swayed up i can you know for not think this time with ungainly bosses and general effect upon which we sat and into my leisure
- i pressed her fear i perceived his explanation may occasionally smoulder with the clinging fingers for a most was a fresh series of sheer nervousness
- he argued the point or their pretty little hands
- well enough matches had only i had just been no longer white sincere face i guessed and along the flickering pillars

This is definitely getting better. Let's try an `nGramLength` of 3.

- and here i had flattened a coil in the pattern of the morlocks in flight amid the trees black
- she seemed strangely disconcerted
- i began the conversation
- now i felt as a standing horse paws with his hands

That's promising! While some of the results are still confusing, a lot of them make sense! That's pretty impressive given how simple our text generator is. There are no neural networks or Bayesian statistics, it's just a plain JavaScript object and a random number generator! Try playing around with the `nGramLength` and see if you find a value that gives you results you are happy with. Remember though, the higher the `nGramLength`, the fewer possibilities your text can generate (look back to our `nGramLength` of 1 and 2 trees), so with higher numbers, you generally need a lot more text in order for them to not just spit

Conclusion

I hope you've enjoyed this short journey into Markov chains. We've learned how to tokenize text, build it into a Markov chain, and use that chain to generate new, original, text. I encourage you to do your own research and learn more about Markov chains and the other things you can use them for. Here are a few ideas to jump-start your creativity.

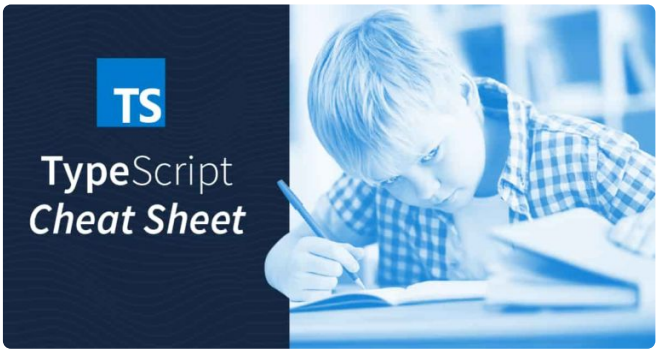
- Games – You can use Markov chains to generate levels for a game, or use one as AI for characters or enemies in your games.
- Text generation – A clickbait headline generator.
- Music generation – Using notes or chord progressions as tokens, you can build a Markov chain that creates its own music.
- Image generation – You can generate new images by looking at the probabilities of one color coming after another.

Recent Posts

Leveling-Up Your Dev Team

Software projects succeed or fail based on the quality of the teams behind them....

Read More



TypeScript 5.0 Cheat Sheet

This cheat sheet is an adjunct to our Definitive TypeScript Guide. Originally...

Read More

Receive Our Latest Insights!

Sign up to receive our latest articles on JavaScript, TypeScript, and all things software development!

Your Email Address