Language Model

ICS2203 Natural Language Processing: Methods and Tools Project Task 1

Luke Ellul
ID: 42498M

Contents

Getting Started

This program requires Node.js[1] 6 or later and npm (Node package manager) which is usually downloaded with Node.js itself. Then, navigate to the root directory (there's supposed to be file **package.json**), open the terminal and type **npm install**. This will install all the dependencies needed for the program to run.

The documentation will explain which files to run to use the language model. For now, it's sufficing to say that to run a Node file you need to navigate to your file's directory with the terminal or whatever, and issue the command **node** followed by whatever file you intend to run.

[1] https://nodejs.org/en/

Corpus Processing

The English corpus[2] was used for this language model. The directory **corpus-cleaner** contains **cleaner.js** which includes the code responsible for parsing the xml files of the corpus and deriving the sentences. The derived sentences are saved as plain text files in the directory **cleanTexts** which are then processed by the language modeler.

To derive the appropriate text data from the xml files **cheerio[3]** was used. Cheerio is a Node.js module which is primarily used to scrape content from html pages but can also be used for xml.

For this project the files in directories A/A0 and B/B0 were used to generate the language model.

[2] http://ota.ox.ac.uk/desc/2554

[3] https://www.npmjs.com/package/cheerio

Processing Raw Text Files

The file **probFunctions.js** takes care of the mathematics behind the language model. The function **cond** calculates the conditional probability of an event given other events. For example:

Cond(A, B, C)

Calculates the probability of A given that B and C also occur. The function accepts two other arguments: **P** and **V**. P is the probability function, the function responsible for measuring the probability of some word occurring in a text file for example. V is a number which is plugged in the formula for Laplace smoothing. In our case, V is the size of our vocabulary.

In our case, we want to calculate the probability of a word occurring given that other words occur before it, or none at all if we want a unigram. This is where **iterateTokens** kicks in. The first argument to **iterateTokens** is **n** which accepts a number that defines the amount of words we want before a word (including the word) to calculate its conditional probability. So, if we have the sentence:

You are a duck

and we want to calculate the probability that *duck* occurs given there's *a* before it we set n to 2. Likewise, we set n to 3 if want to calculate the probability that *duck* occurs given there's *a* and *are* before it and so on. We set n to 1 if we simply want the probability that there's *duck*.

The next argument to **iterateTokens** is **tokens** which is simply a list of words. Each of the words will be processed by a probability function along with *n* previous words. The next argument, **P**, is the probability function which is injected into the conditional probability function defined earlier. This function better be fast because it will be called with every single word in the list of tokens.

The next argument is **fn** which is also a function that will be called with the processed data for each token in the list. The processed data includes the conditional probability of a word and its previous words, the word itself, an array of the *n* words that come before the processed word and the probability of the processed word on its own.

The last argument is **V** which defines the vocabulary size used for Laplace smoothing.

The function **iterateTokens** returns a mapping of the tokens to their processed data. So, for the sentence above, **iterateTokens** will return an array containing the processed data for each word.

Say we have the sentence:

Travel around the globe

The bigram stores the processed data as follows:


```
{
    "travel":
    {
        "bi":
        {
            "around": 1
        },
        "P": 0.006024096385542169
    }
},
{
    "around":
    {
        "bi":
        {
            "the": 1
        },
        "P": 0.04216867469879518
    }
},
{
    "the":
    {
        "bi":
        {
            "globe": 0.14285714285714288
        },
        "P": 0.006024096385542169
    }
}
```


Each word is converted into a JavaScript object which contains the word itself as the key and an object with two keys. For the bigram, the key **bi** contains an object which contains a map of all the words that come after the processed word. In this map, each word is mapped with a

number defining the probability that this word is included after the initial word. So in the above example, there's approximately a 0.14 probability that the word *globe* comes after the word *the*. The key *P* shows the probability that the word *the*, for example, occurs on its own. This is used for the bigram.


```
{
        "miami": {
            "tri": {
                "at": {
                    "the": 1
                }
            },
            "P": 0.04216867469879518
        }
    },
    {
        "at": {
            "tri": {
                "the": {
                    "drop": 1
                }
            },
            "P": 0.006024096385542169
        }
    }
```


The trigram processes data in a similar fashion. In the above case, we can see that there's a full chance that the word *the* comes after the words *miami* and *at*.

Storing the Processed Data

The processed data for each word are stored in LevelDB[4], a light-weight, fast, key-value database developed by Google engineers where keys are sorted lexicographically. LevelDB has bindings for Node.js, Python and C++ which means this language model can be used with these languages as well.

For each word and its processed data, the word is added to the database as the key and its value will be the processed data. When using the ngrams, to check for example which word comes after *the*, the database searches for the key *word* and gets its value. The value will contain a map of all the words that may come after *the* each with its respective conditional probability. For the function **GenerateText** the most likely word will be chosen and for **WrongWord** the most unlikely word will be chosen. To include variety in the chosen words, an algorithm will choose one of the words at random but according to their probability. So, for **GenerateText**, a word which has a higher probability will have a higher chance of being chosen. The inverse is true for **WrongWord**.

Take , for example, the value for the key **writer** (this is an incomplete processed corpus, the results here don't reflect the results of the proper language model).


```
{
        "bi": {
            "in": 0.1,
            "with": 0.025,
            "was": 0.025,
            "from": 0.05,
            "honestly": 0.025,
            "can": 0.075,
            "stops": 0.025,
            "of": 0.025,
            "something": 0.025,
            "shops": 0.025,
            "who": 0.1
        },
        "P": 0.08590042461077346
    }
```


[4] http://leveldb.org/

The words *in* and *who* have the highest probability of being chosen from **GenerateText** for the word *writer*. The word *can* has the second highest chance of being chosen and so on. For **WrongWord** the words with the lowest probability have the highest chance of being chosen.

Each ngram has its own database. To add more processed data o the database the function **buildNGram** located in **buildModel.js** can be used. The function accepts two arguments, the ngram function (unigram, bigram or trigram) and an array of the location of plain text files that you want to process. The function will log the percentage of data processed. If there are words in the database which have already been keyed, their values will be merged.

Laplace smoothing

Laplace smoothing is achieved by using the **laplace** function in **processing.js**. This function accepts an argument **V** which defines the vocabulary size. The vocabulary size is calculated by counting the amount of keys in the bigram or trigram databases. The second argument to **laplace** accepts the ngram function that we want to apply the smoothing on (bigram or trigram). **laplace** will return a function which can be injected into **buildNGram** to build a database for Laplace smoothing.

To calculate the vocabulary size the **calcVocab** function in **model.js** is used. You already need to have a database of a bigram or a trigram to use **calcVocab**. **calcVocab** accepts a database name (in our case either *trigram* or *bigram*) and returns the number of keys stored in the database.

Building the Ngrams

The ngrams are built using the **buildNGram** function in **buildModel.js**. The function accepts an ngram function and the location of the plain text files. Let's say we want to build an ngram from file **B0B.txt**:

```
//location of plain text file
const location = './corpus-cleaner/cleanTexts/B0B.txt';

//build bigram
buildNGram(bigram)([location])
.then(
    () => console.log('done building bigram ' + location),
    err => console.log(err)
);

//build trigram
buildNGram(trigram)([location])
.then(
    () => console.log('done building trigram ' + location),
    err => console.log(err)
);

//build bigram with laplace smoothing
calcVocab('bigram')
.then(
    i => buildNGram(laplace(i)(bigram))([location])
)
.then(
    () => console.log('done building laplace bigram' + location),
    err => console.log(err)
);

//build trigram with laplace smoothing
calcVocab('trigram')
.then(
    i => buildNGram(laplace(i)(trigram))([location])
)
```

```
.then(
    () => console.log('done building laplace trigram' + location),
    err => console.log(err)
);
```

The functions **bigram, trigram** and **laplace** need to be imported from **processing.js**. Since JavaScript is single threaded and the data processing is synchronous it's not a good idea to run all these functions on the same process. Instead, simply run **node buildModel** for each one of these functions (you can comment out the other ones) and they will all run in parallel, assuming that your computer has four or more cores. Of course, this should be scripted (using the Node.js **cluster** module). I simply didn't have time, my bad.

The databases for the processed ngrams are stored in the directory **lang-model-storage/databases**.

Using the Language Model

The functions **GenerateText** and **WrongWord** find the next most likely word or most unlikely word respectively after a phrase. These functions take an ngram function as the first argument and a sentence, phrase or whatever as the second argument. The functions will return the answer wrapped in a Promise. A Promise is a JavaScript object used to return objects asynchronously. When using these functions, we are using LevelDB to find our keys and since LevelDB works asynchronously, we return our results asynchronously wrapped in a Promise which can then be accessed later in time when the results are actually found. Node.js works asynchronously (it's actually its main selling point). This way, we can calculate the next word of several sentences using different ngram models all the same time using a single thread.

Here are some examples:

```
//unigram
GenerateText(unigram)('since there are a lot of').then(
    nextWord => console.log(nextWord),
    err => console.log(err)
);

//bigram
GenerateText(bigram)('since there are a lot of').then(
    nextWord => console.log(nextWord),
    err => console.log(err)
);

//trigram
GenerateText(trigram)('since there are a lot of').then(
    nextWord => console.log(nextWord),
    err => console.log(err)
);
```

```
//bigram with laplace smoothing
calcVocab('bigram').then(
    n => GenerateText(laplace(n)(bigram))('since there are a lot
of').then(
        nextWord => console.log(nextWord),
        err => console.log(err)
    )
);

//trigram with laplace smoothing
calcVocab('trigram').then(
    n => GenerateText(laplace(n)(trigram))('since there are a lot
of').then(
        nextWord => console.log(nextWord),
        err => console.log(err)
    )
);
```

The calculate the next wrong word simply use the **WrongWord** function wherever **GenerateText** is used.

Using the Backoff Model

The **backoff** function in **model.js** finds the next word in a sentence using the Backoff model. The function accepts a sentence as its first input and the **GenerateText** or **WrongWord** functions as its second input.

```
//backoff model
backoff('since there are a lot of')(WrongWord)
.then(
    nextWord => console.log(nextWord),
    err => console.log(err)
);
```

I also implemented a function, **talk**, which uses to Backoff function. **talk** takes a sentence or a phrase as its one and only argument and every second it appends a new word to that sentence. This will go on forever, but unlike some people out there it will stop talking when you tell it to.

Code

buildModel.js

```
const { unigram, bigram, trigram, laplace } = require('./processing');
const { generateCountFn, matchWords } = require('./tokens');
const { storeInDatabase, openStore } = require('./lang-model-
storage/store');
const { calcVocab } = require('./model');
const { iterateThroughFiles } = require('./fileOps');

const buildNGram = nGram => texts =>
    storeInDatabase(
        (
            (x, y) => (x === 1 ? 'unigram' :
                x === 2 ? 'bigram' : 'trigram') +
                (y ? 'Laplace' : '')
        )(nGram('getN'), nGram('isLaplace'))
    )
        (
        nGram(
            generateCountFn(texts)
        )(
            ...(() => {
                let theseWords = [];
                iterateThroughFiles(texts)(
                    text =>
theseWords.push(...matchWords(text.toString()))
                );
                return theseWords;
            })()
            )
        );

// //location of plain text file
// const location = './corpus-cleaner/cleanTexts/B0B.txt';

// //build bigram
```

```
// buildNGram(bigram)([location])
// .then(
//     () => console.log('done building bigram ' + location),
//     err => console.log(err)
// );

// //build trigram
// buildNGram(trigram)([location])
// .then(
//     () => console.log('done building trigram ' + location),
//     err => console.log(err)
// );

// //build bigram with laplace smoothing
// calcVocab('bigram')
// .then(
//     i => buildNGram(laplace(i)(bigram))([location])
// )
// .then(
//     () => console.log('done building laplace bigram' + location),
//     err => console.log(err)
// );

// //build trigram with laplace smoothing
// calcVocab('trigram')
// .then(
//     i => buildNGram(laplace(i)(trigram))([location])
// )
// .then(
//     () => console.log('done building laplace trigram' + location),
//     err => console.log(err)
// );
```

fileOps.js

```
const fs = require('fs');

function iterateThroughFiles(locations, origLocs = locations) {
    locations.length === 0 || this[locations[0]] ||
    (this[locations[0]] = fs.readFileSync(locations[0]));
    return cb => locations.length === 0 ? origLocs :
        (cb(
            this[locations[0]], locations[0]),
            iterateThroughFiles(locations.slice(1), origLocs)(cb)
        );
}

module.exports = {
    iterateThroughFiles
}
```

model.js

```
const { unigram, bigram, trigram, laplace } = require('./processing');
const { generateCountFn, matchWords } = require('./tokens');
const { storeInDatabase, openStore } = require('./lang-model-
storage/store');
const { fromJS } = require('immutable');

const location = './lang-model-storage/databases/';

const getExpo = n => {
    const expo = n.toExponential();
    const index = expo.lastIndexOf('-') === -1 ? expo.lastIndexOf('+')
: expo.lastIndexOf('-');
    return Number(expo.slice(index));
}

const calcNextWord = (inverse = false) => o => {
    const obj = fromJS(o);
    let i = 0;
    const newObj = obj.map(v => [i, (i = i + (!inverse ? v || 0 : !v ?
0 : (1 / v)), !inverse || v ? i : 0)]);
    const randomNum = Math.random() * i;
    return newObj.findKey(v => v[1] === 0 && Math.random() < 0.7) ||
newObj.findKey(v => randomNum >= v[0] && randomNum < v[1]);
}

const nextWord = calcNextWord();
const nextWrongWord = calcNextWord(true);

const operateModel = fn => nGram => sentence => {
    const nGramNo = nGram('getN');
    if (nGramNo === 1) {
        const rand = Math.random() * (1 - 0.5) + 0.5;
        return new Promise(res => {
            openStore(location + 'bigram')
                .then(db =>
                    db.createReadStream()
                        .on('data', data =>
```

```
                            !data.key.match(/\d/) &&
                            data.value.P > (rand * Math.pow(10,
getExpo(data.value.P))) &&
                            Math.random() < 0.005 &&
                            res(data.key)
                        )
                )
        });
    }
    const nGramType = (x => nGram('isLaplace') ? x + 'Laplace' :
x)(nGramNo === 1 ? 'unigram' :
        nGramNo === 2 ? 'bigram' : 'trigram');
    const lastWords =
matchWords(sentence.toLowerCase()).slice((nGramNo * -1) + 1);
    return new Promise((res, rej) => {
        openStore(location + nGramType)
            .then(db =>
                db.get(lastWords[0], (err, data) => {
                    if (err && err.notFound) {
                        rej('word not found');
                    }
                    else if (nGramNo === 2)
                        res(fn(data.bi || (lastWords[0] ===
'undefined' && data.tri['undefined'])));
                    else if (nGramNo === 3)
                        (x => x ? res(fn(x)) : rej("trigram not
found"))(data.tri[lastWords[1]] || (lastWords[0] === 'undefined' &&
data.tri['undefined']))
                })
            )
    });
}

const calcVocab = databaseName => {
    let i = 0;
    return new Promise((res, rej) => {
        openStore(location + databaseName).then(
            db => db.createReadStream()
                .on('data', () => i++)
```

```
                .on('end', () => res(i))
        );
    });
}

const GenerateText = operateModel(nextWord);
const WrongWord = operateModel(nextWrongWord);

const backoff = sentence => fn => new Promise((res, rej) => {
    fn(trigram)(sentence).then(
        v => res(v),
        v => v === "trigram not found" && fn(bigram)(sentence).then(
            v => res(v),
            v => v === 'word not found' && fn(unigram)(sentence).then(
                v => res(v),
                err => console.log('backoff error\n' + err) ||
rej(err)
            )
        )
    )
});


//backoff model
backoff('we have become')(GenerateText)
.then(
    nextWord => console.log(nextWord),
    err => console.log(err)
);

function talk(sentence) {
    return new Promise(res => {
        backoff(sentence)(GenerateText)
            .then(
            nextWord => console.log(nextWord) ||
                setTimeout(
                    () => res(sentence + ' ' + nextWord),
                    1000
                )
```

```
            )
    })
        .then(res => talk(res));
}

//using talk
//talk("we have become");

//unigram
GenerateText(unigram)('since there are a lot of').then(
    nextWord => console.log(nextWord),
    err => console.log(err)
);

//bigram
GenerateText(bigram)('since there').then(
    nextWord => console.log(nextWord),
    err => console.log(err)
);

//trigram
GenerateText(trigram)('since there').then(
    nextWord => console.log(nextWord),
    err => console.log(err)
);

//bigram with laplace smoothing
calcVocab('bigram').then(
    n => GenerateText(laplace(n)(bigram))('since there are a lot
of').then(
        nextWord => console.log(nextWord),
        err => console.log(err)
    )
);

//trigram with laplace smoothing
calcVocab('trigram').then(
    n => GenerateText(laplace(n)(trigram))('since there are a lot
of').then(
```

```
        nextWord => console.log(nextWord),
        err => console.log(err)
    )
);

module.exports = {
    GenerateText,
    WrongWord,
    backoff,
    calcVocab
};
```

probFunctions.js

```
const product = (...N) => N.reduce((num, n) => num * n);

const cond = (A, ...B) => (P, V) =>
    (P(...B, A) + (V ? 1 : 0)) / (P(...B) + (V ? V : 0));

const iterateTokens = n => (...tokens) => (P, fn, V) => {
    let count = 0;
    let length = tokens.length;
    return tokens.map(
        (t, i) => {
            console.log(((++count / length) * 100) + '%');
            const slicedTokens = n !== 1 && tokens.slice(
                (j => j < 0 || n === 0 ? 0 : j)(i - (n - 1)), i);
            const Pt = P(t);
            return (fn || (v => v))
                (
                    n === 1 || i === 0 ? Pt : cond(t,
...slicedTokens)(P, V),
                    t,
                    slicedTokens,
                    Pt
                )
        });
    }

const markov = n => (...tokens) => P =>
    product(...iterateTokens(n)(...tokens)(P));

const chain = markov(0);

module.exports = {
    markov,
    chain,
    iterateTokens
}
```

processing.js

```
const R = require('ramda');
const { Map, List } = require('immutable');
const { iterateTokens } = require('./probFunctions');

const organize = (v, t, historyTokens, PoV) => [
    t.toLowerCase(),
    List([v, historyTokens.map(v => v.toLowerCase())]),
    PoV
];

const processOrganizedInfo = v => {
    const preTokens = v[1].get(1);
    const afterToken = v[0];
    const PoV = v[2];
    return Map({
        [preTokens[0]]: preTokens.length === 1 ?
            Map({
                'bi': Map({
                    [afterToken]: v[1].get(0)
                }),
                P: PoV
            }) :
            Map({
                'tri': Map({
                    [preTokens[1]]: Map({
                        [afterToken]: v[1].get(0)
                    })
                }),
                P: PoV
            })
    });
}

const nGram = (n, V) => countFn => countFn === 'getN' ? n : countFn
=== 'isLaplace' ? V : (...tokens) =>
    iterateTokens(n)(...tokens)(
        countFn,
```

```
        R.pipe(
            organize,
            processOrganizedInfo
        ),
        V
    );

const unigram = nGram(1);
const bigram = nGram(2);
const trigram = nGram(3);

const laplace = V => ngram => nGram(ngram('getN'), V);

module.exports = {
    unigram,
    bigram,
    trigram,
    laplace
};
```

tokens.js

```
const { iterateThroughFiles } = require('./fileOps');
const memoize = require('memoize-immutable');

const matchWords = memoize(text => text.match(/\w+/gi));

const getWordCount = text => (matchWords(text) || []).length;
const getTokenCount = tokens => (tokens || []).length;

const joinTokens = (...tokens) => tokens.join(' ');

const removePunctuation = text => joinTokens(...matchWords(text) ||
[]);

const getPhrases = memoize(text =>
    (text.match(/[\w\d][^.!?,:;()]*[.!?,:;()]/g) || [])
        .map(removePunctuation));

const getWordCountOfTexts = textLocations => {
    let count = 0;
    iterateThroughFiles(textLocations)(
        text => count = count + getWordCount(text.toString())
    );
    return count;
}

const generateCountFn = textLocations => {
    const textWordCount = getWordCountOfTexts(textLocations);
    return (...tokens) => {
        let tokenCount = 0;
        iterateThroughFiles(textLocations)(
            text => tokenCount =
getPhrases(text.toString()).reduce((currentTokenCount, sentence) =>
                currentTokenCount + getTokenCount(sentence.match(
                    new RegExp(
                        removePunctuation(joinTokens(...tokens)),
                        'gi'
                    )
```

```
                )), tokenCount)
        );
        return tokenCount / textWordCount;
    }
}

module.exports = {
    generateCountFn,
    matchWords
};
```

Directory corpus-cleaner

cleaner.js

```
const cheerio = require('cheerio');
const {iterateThroughFiles} = require('../fileOps');
const fs = require('fs');

const clean = textsLocations => iterateThroughFiles(textsLocations)(
    (text, textLocation) => {
        const $ = cheerio.load(text.toString(), {
            xmlMode: true
        });
        const cleanText = $('s').map(
            (i, el) => $(el).children().map(
                (i, el) => $(el).text()
            ).get().join('')
        ).get();
        fs.writeFileSync(
            './cleanTexts' +
textLocation.slice(textLocation.lastIndexOf('/'),
textLocation.lastIndexOf('.')) + '.txt',
            cleanText.join(' ')
        );
    }
);

const cleanText = rawXml => {
    const $ = cheerio.load(rawXml.toString(), {
        xmlMode: true
    });
    const cleanText = $('s').map(
        (i, el) => $(el).children().map(
            (i, el) => $(el).text()
        ).get().join('')
    ).get();
    return cleanText.join(' ');
}
```

```
module.exports = {
    clean,
    cleanText
};
```

lang-model-storage directory

store.js

```
const level = require('level');
const { Map, List, fromJS } = require('immutable');

const getKey = map => map.keys().next().value;

const getValue = map => map.first();

const mergeValues = (v1, v2) => v1.mergeDeep(v2);

function openStore(storeLocation) {
    this[storeLocation] = this[storeLocation] || level(storeLocation,
{ valueEncoding: 'json' });
    return Promise.resolve(this[storeLocation]);
}

const storeInDatabase = location => list =>
    list.reduce(
        (store, map) => {
            const key = getKey(map);
            const value = getValue(map);
            return new Promise((res, rej) => {
                store.then(
                    db => {
                        db.get(key).then(
                            v => db.put(key, mergeValues(fromJS(v),
value)).then(() => res(db)),
                            err => err.message.includes('Key not
found') ? db.put(key, value).then(() => res(db)) :
                                        console.log(err)
                        )
                    }
                )
            })
        },
        openStore('./lang-model-storage/databases/' + location)
```

```
    );

module.exports = {
    storeInDatabase,
    openStore
}
```