N-gram 문장 확률 구하기

20130940 장용훈

1. UnigramSentence 분석

필요한 인풋은 다음과 같다. 유니그램 빈도수가 나타나 있는 파일, 그리고 분석하고자 하는 문장들의 문서이다. 그리고 결과를 쓸 result가 필요하다. 이전 과제에서 만든 uniFileRead로 유니그램을 읽는다.

```
while (iss>>num>>data)
{
unitotal += num;//모든 유니그램의 빈도수를 더해준다.
pair<string, int> p = make_pair(data, num);
v.push_back(p);
}

sort(v.begin(), v.end());//문장의 순서를 기준으로 정렬 만약 동일한 순서면 횟수에 따라 정렬
```

uniFileRead는 유니그램 형식으로 나타나 있는 데이터를 읽어서 벡터에 저장하여 활용 하는 것이다. 이때 추후 검색을 위해 벡터를 sort해주었다. STL에서 제공해주는 함수인데, NlogN의 속도를 가지고 있어서 매우 빠르다. 이때 유니그램 파일을 읽는 방법은 아래에 설명되어 있는 방법과 동일하다.

```
void unigramSentence(const char* _path, const char* _targetPath, const char* _pathResult)
{
    uniFileRead(_path);//유니그램을 읽는다.
    FILE * pFile = fopen(_targetPath, "rb");
    fstream myfile;
    myfile = fstream(_pathResult, std::ios::out | std::ios::binary | std::ios::app);

    fseek(pFile, 0, SEEK_END);// 파일 포인터를 파일의 끝으로 이동
    long lSize = ftell(pFile);// 파일 시작부터 맨 끝까지의 거리를 return
    fseek(pFile, 0, SEEK_SET);// 파일 포인터를 파일의 처음으로 이동

    char *buff = (char*)malloc(sizeof(char)*lSize);

    long long totnum = 0;
    long long curnum = 0;

    //read all big file
    while ((curnum = fread(&buff[totnum], sizeof(char), lSize - totnum, pFile)) > 0) {
        totnum += curnum;
    }
```

파일은 FOPEN함수를 이용하여 바이너리로 한번에 읽어드린다. 이때 파일 포인터의 처음과 끝에 대한 정보를 얻어, 파일을 제대로 읽었는지 확인을 해준다. 데이터가 아무리 많다고 하더라도 메모리가 허락하는 한 읽는 시간은 불과 몇 초 밖에 걸리지 않는다.

```
if (totnum != lSize) {
    cout << "not read all file" << endl;
}
buff[totnum] = NULL;
string a(buff);
fclose(pFile);
stringstream iss;
stringstream iss2;
iss.str(a);
int num;
string data; string data2; string data3;
stringstream stream;
double total = 1;
double temp = 1;
string temp2;
while (getline(iss, data, '₩n'))
{
    iss2.str(data);
    while (iss2 >> data2)
    {
        num = uniFind(data2);
        temp = (double)(num + 1) / (v.size() + unitotal);
```

읽어온 데이터(문장을 기준으로 WORDCOUNT된 것)는 공백 처리가 되어 있지 않기 때문에 이에 대한 처리를 해준다. 읽어온 데이터를 STRING으로 만들어 준 후, STRINGSTREAM을 이용하여 공백을 제거 해준다. 첫번째 반복문 while은 문장에서 줄바꿈 문자를 기준으로 데이터를 뽑아온다. 이때 더 이상 줄바꿈 문자가 존재 하지 않을 때까지 그 작업이 이루어 진다. 한문장은 data에 저장이 되고, 이 data를 이용해서 다시한번 while문을 실행해 준다. 이 두 번째 while 문은 공백(띄어쓰기)를 기준으로 단어들을 가져온다. 이 단어를 data2에 저장한다. 한번 루프가 돌때마다 data2에대한 unigram수를 uniFind를 통해 얻어 온다.

```
int uniFind(string sentence)
{
    long long left = 0;
    long long right = v.size();
    long long mid = 0;
    while (left < right) {
        mid = left + (right - left) / 2;
        if (sentence > v[mid].first) {
            left = mid + 1;
        }
        else if (sentence < v[mid].first) {
            right = mid;
        }
        else {
            return v[mid].second;
        }
    }
    return false;
}
```

Unifind는 다음과 같다. 단순한 Linear Search 가 아닌 정렬된 벡터 v를 활용하여 LinearSearch를 수행하여 N의 시간 복잡도에서 log N으로 소모시간이 절반가량으로 줄어 들게 된다. 이때 search의 기준은 unigram 단어들의 사전배열식 순서이다.

그 후 검색한 값으로 유니그램 확률을 구해준다. 확률이 0이 나오는 것을 방지하기 위해 Laplace smoothing에서 add one 방법을 이용하였다. 따라서 구한 값+1/전체 단어의 개수+unigram 단어들의 개수를 더하여 유니그램 확률을 구해준다.

```
            total *= temp;
        }
        myfile << total;
        data += "  ";
        data += "₩r₩n";
        myfile.write(data.c_str(), data.size());
        iss2.clear();
        stream.clear();
        temp2.clear();
        total = 1;
    }

    myfile.close();
```

이때 구해준 값을 한문장을 기준으로 모두 곱해준다. 다시 말하자면 두번째 while문이 실행될 때 마다 유니그램 확률을 구해 곱해준다는 것이다. 하나의 문장이 모두 처리가 된다면 두번째 while은 종료가 된다. 이 후 file에 먼저 해당 문장의 확률을 써준 뒤, 뒤이어 확률을 구해준 문장을 파일에 써 준다. 전체 문장에 관하여 이러한 과정들이 반복 된다.(첫번째 while). 파일에 다 쓰고 나서 file stream을 닫아준다.

2. Bigram Sentence 분석

```
void bigramSentence(const char* _unipath, const char* _bipath, const char* _targetPath, const char* _pathResult)
{
    uniFileRead(_unipath);//유니그램을 읽는다.
    biFileRead(_bipath);//바이그램을 읽는다.
    FILE * pFile = fopen(_targetPath, "rb");
    fstream myfile;
    myfile = fstream(_pathResult, std::ios::out | std::ios::binary | std::ios::app);

    fseek(pFile, 0, SEEK_END);// 파일 포인터를 파일의 끝으로 이동
    long lSize = ftell(pFile);// 파일 시작부터 맨 끝까지의 거리를 return
    fseek(pFile, 0, SEEK_SET);// 파일 포인터를 파일의 처음으로 이동

    char *buff = (char*)malloc(sizeof(char)*lSize);

    long long totnum = 0;
    long long curnum = 0;

    //read all big file
    while ((curnum = fread(&buff[totnum], sizeof(char), lSize - totnum, pFile)) > 0) {
        totnum += curnum;
    }
```

1번과 로직은 거의 유사하다. 다만 다른 점이 있다면, 유니그램 파일 뿐만 아니라 바이그램 파일 또한 읽어 온다는 점이다. unifileRead는 동일하고 bifileRead만이 추가 되었다.

```
while (iss >> num >> data)
{

    bitotal += 1;
    pair<string, int> p = make_pair(data, num);
    v2.push_back(p);
}
sort(v2.begin(), v2.end());//문장의 순서를 기준으로 정렬 만약 동일한 순서면 횟수에 따라 정렬
```

사실상 함수의 이름만 다를 뿐이지, bifileRead 또한 그 방법이 동일하다. 데이터 축적을 위하여 v2라는 벡터에 모든 바이그램 데이터를 pair 형식으로 저장 후 sort해 주었다.

또한 문장 형식으로 읽어오는 부분 역시 1번과 동일하다. 하지만 읽어온 문장을 처리하는 방식은 다르다.

```
while (getline(iss, data, '₩n'))
{
    iss2.str(data);
    iss2 >> data2;
    uni = uniFind(data2);
    bi = biFind(data2);
    total *= ((double)(bi + 1) / (uni + bitotal));
```

문장을 처리하는 큰 틀은 동일하다. 첫번째 while은 전체 텍스트를 하나의 문장으로 계속 해서 나누어 준다. 그 후 두번째 while이 시작되기전에 문장에서 하나의 데이터를 읽어와서 이를 유니그램 빈도와 바이그램 빈도 수를 구해준다. 왜냐하면 문장이 있을 때, 첫번째 단어에 대한 바이그램 역시 구해주어야 하기 때문이다. 이 때 바이그램 빈도를 찾는 방법은 1번 방법과 똑 같은 binary search로 구현해 주었다.

```
int biFind(string sentence)
{
    long long left = 0;
    long long right = v2.size();
    long long mid = 0;
    while (left < right) {
        mid = left + (right - left) / 2;
        if (sentence > v2[mid].first) {
            left = mid + 1;
        }
        else if (sentence < v2[mid].first) {
            right = mid;
        }
        else {
            return v2[mid].second;
        }
    }
    return false;

}
```

마찬가지로 Linear Search와 비교 했을 때 우수한 성능을 보여준다. 이때 search 의 기준이 되는 데이터는 바이그램 데이터들이 모여 있는 v2이다.

첫 단어에 대한 유니그램과 바이그램 빈도수를 구한 뒤, 바이그램 빈도수+1/유니그램 빈도수+바이그램 총 단어수 로 바이그램의 확률을 구해준다. 이 값을 total 에 곱해준다. 그 후 두번째 while이 시작된다. 무한루프로 특정 조건이 만족시에만 동작이 마치게 되어 있다.

```
while (1)
{
    if (iss2 >> data3) {
        uni = uniFind(data2);
        bi = biFind(data2+data3);
        total *= ((double)(bi + 1) / (uni + bitotal));
        data2 = data3;
    }
    else {
        uni = uniFind(data2);
        bi = biFind(data2);
        total *= ((double)(bi + 1) / (uni + bitotal));
        break;
    }
}
```

If 조건안에서 문장에 관한 단어를 얻는다. 만약 if 가 참이 아니라면, 단어를 얻어 오는 것이 실패 했다는 것이고, 이 말은 즉 문장의 마지막 단어까지 모두 얻어왔다는 것이다. If 문의 조건, 단어를 성공적으로 얻어 왔다면 이를 중심으로 바이그램 확률을 구해준다. 항상 첫번째 단어는 data2에 있기에, 이를 이용하여 uniFind를 통해 유니그램 빈도수를 구해준다. 바이그램의 빈도수는 biFind 함수를 이용하여 구한다. 이때 입력 값으로 합 쳐진 두단어 data2+data3를 입력으로 주어서 이에 해당하는 빈도수를 얻어온다. 그 후 앞서 바이그램 확률을 구한 방법과 동일한 방법으로 구한후 total에 곱해준다. 마지막단어 까지 모두 얻은 후 else 가 실행이 된다. 마지막 단어(data2)에 관하여 유니그램 확률을 구해준 후 곱한다. 그후 while문을 종료 해준다. 이로서 한 문장에 관한 작업들이 모두 끝이 난다.

```
            }
            myfile << total;
            temp += " ";
            data += "₩r₩n";
            temp += data;
            myfile.write(temp.c_str(), temp.size())
            iss2.clear();
            temp.clear();
            total = 1;
        }

        myfile.close();

    }
```

문장에 대한 작업이 끝난 후, 파일에 확률값과 해당하는 문장을 써준다. 이 작업이 모든 문장에 대하여 이루어 진다면, while을 종료해주고 파일을 닫아준다.

3. Trigram sentence 분석

```
void trigramSentence(const char* _bipath, const char* _tripath, const char* _targetPath, const char* _pathResult)
{
    biFileRead(_bipath);//바이그램을 읽눈다.
    triFileRead(_tripath);//트라이그램을 읽는다
    FILE * pFile = fopen(_targetPath, "rb");
    fstream myfile;
    myfile = fstream(_pathResult, std::ios::out | std::ios::binary | std::ios::app);

    fseek(pFile, 0, SEEK_END);// 파일 포인터를 파일의 끝으로 이동
    long lSize = ftell(pFile);// 파일 시작부터 맨 끝까지의 거리를 return
    fseek(pFile, 0, SEEK_SET);// 파일 포인터를 파일의 처음으로 이동

    char *buff = (char*)malloc(sizeof(char)*lSize);

    long long totnum = 0;
    long long curnum = 0;

    //read all big file
    while ((curnum = fread(&buff[totnum], sizeof(char), lSize - totnum, pFile)) > 0) {
        totnum += curnum;
    }
```

앞서 유니그램과 바이그램의 읽는 방식과 유사하다. 트라이그램 확률을 구하기 위해서는 바이그램과 트라이그램의 빈도수가 필요하다. 그에따라 bifileRead과 TrifileRead로 각각의 파일을 읽어 벡터에 각각 저장해 준다. 이때 1,2 와 같이 search를 위해 벡터들을 sort 해준다.

```
while (getline(iss, data, '₩n'))
{
    iss2.str(data);
    iss2 >> data2>>data3;
    bi = biFind(data2);
    tri = triFind(data2);
    total *= ((double)(tri + 1) / (bi + tritotal));
    bi = biFind(data2+data3);
    tri = triFind(data3);
    total *= ((double)(tri + 1) / (bi + tritotal));
```

바이그램을 분석하는 방식과 유사하다. 읽어온 데이터를 string으로 변환하여 처리한다. 이때 첫번째 while은 문장 단위를 인식한다. 이는 바이그램 방법과 유사한 루틴을 가진다. 먼저 data2와 data3에 문장 처음의 두단어를 읽는다. 이때 data2에 관해서 바이그램 빈도수와 트라이그램의 빈도수를 구해준다.

```
int triFind(string sentence)
{
    long long left = 0;
    long long right = v3.size();
    long long mid = 0;
    while (left < right) {
        mid = left + (right - left) / 2;
        if (sentence > v3[mid].first) {
            left = mid + 1;
        }
        else if (sentence < v3[mid].first) {
            right = mid;
        }
        else {
            return v3[mid].second;
        }
    }
    return false;

}
```

트라이그램 빈도수를 구하는 방법이다. Binary search를 수행하였고, 앞서 설명한 방법과 동일하다. 다시 말해 각 find함수는 이름만 다를 뿐이지 수행하는 기능이 동일하다.

Data2(단어)에 관하여 빈도수를 구해준 후, 이를 이용해 트라이그램 확률을 구해준다. 단어 하나를 이용하여(이것을 트라이그램이라 간주한다) 각각의 빈도수를 구한다.

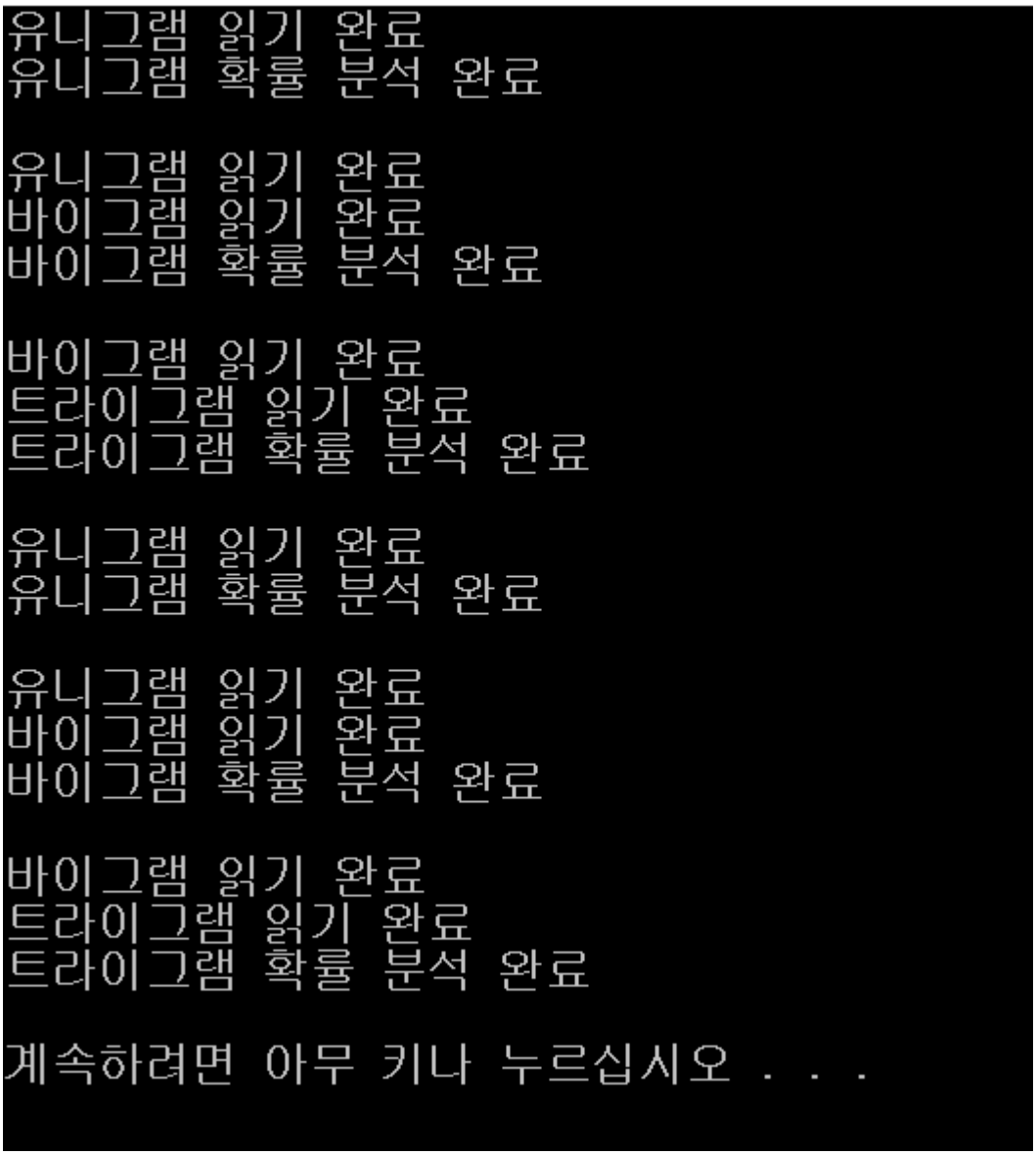
트라이그램 빈도수+1/트라이그램 단어 개수+바이그램 빈도수 수식을 이용하여 트라이 그램 확률을 구해준 후 total에 곱해준다. 그 다음 두번째 반복문을 실행 하기전에 한번 더 수행해 준다. 이번에는 앞의 두단어에 관하여 확률을 구한다. Data2+data3을 합하여 바이그램 빈도를 구하고 data3를 이용하여 트라이그램 빈도수를 구한다. 왜냐하면 data2, data3, (빈칸) 을 하나의 트라이그램 단위로 간주하고 계산하였기 때문이다. 이 확률 역시 total에 곱해준다.

```
while (1)
{
    if (iss2 >> data4) {
        bi = biFind(data2+data3);
        tri = triFind(data2 + data3+data4);
        total *= ((double)(tri + 1) / (bi + tritotal));
        data2 = data3;
        data3 = data4;
    }
    else {
        bi = biFind(data2+data3);
        tri = triFind(data3);
        total *= ((double)(tri + 1) / (bi + tritotal));
        bi = biFind(data3);
        tri = triFind(data3);
        total *= ((double)(tri + 1) / (bi + tritotal));
        break;
    }
}
```

그 다음 마찬가지로 두번째 무한루프 while을 수행해 준다. 한문장을 기준으로 문장의 단어들에 data4에 가져온다. 이를 이용하여 확률을 구한다. Data2+data3을 이용하여 바이그램 빈도수를 data2+data3+data4를 이용하여 트라이그램 빈도수를 구한다. 그 후 빈도수를 이용하여 확률을 구하고 total에 곱해준다. 마지막으로 data를 한칸씩 땡겨준다. 따라서 data2=data3이 되고, data3=data4가 된다. 그럴 경우 루프가 한

번 돌때마다 변수들이 갱신된다. 만약 if문의 조건문이 만족하지 않는다면(문장에 더 이상 단어가 없는 경우) else문이 수행되고 data2 와 data3을 이용하여 마지막으로 트라이그램 확률을 구해주고 두번째 while이 break된다. 한 문장의 확률을 구할 때마다, 문장의 확률값, 문장을 파일에 써준다. 이 작업은 입력파일의 모든 문장에 관하여 이루어진다.

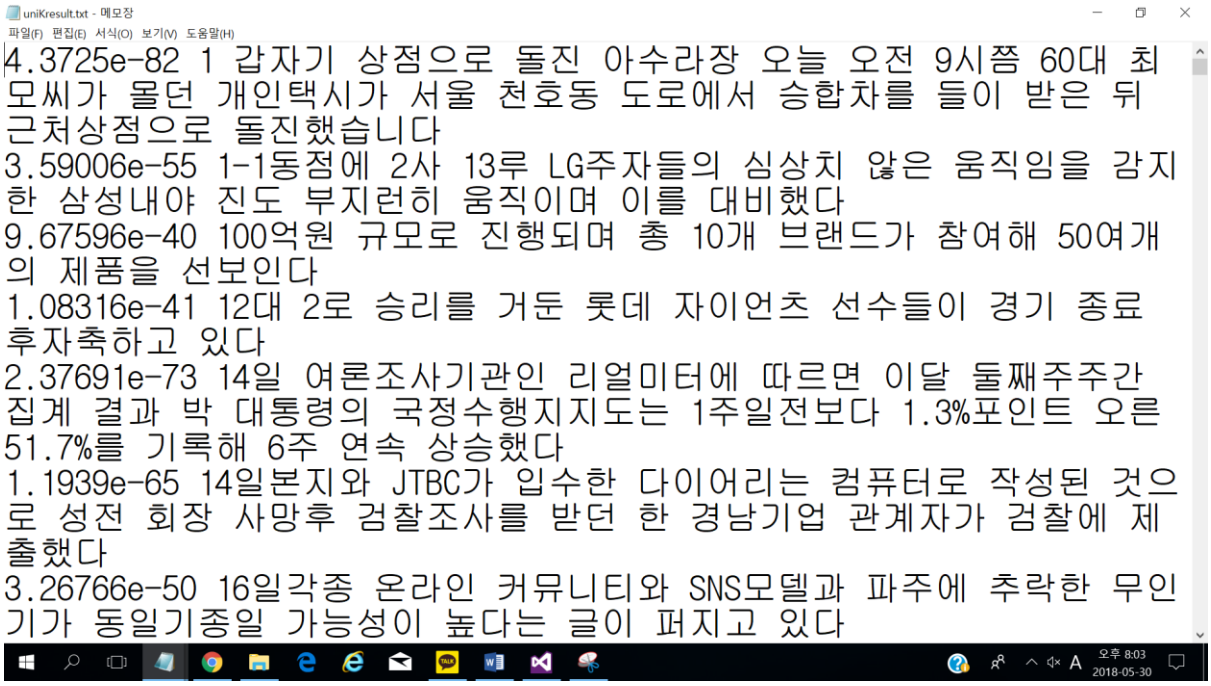
4. 결과값



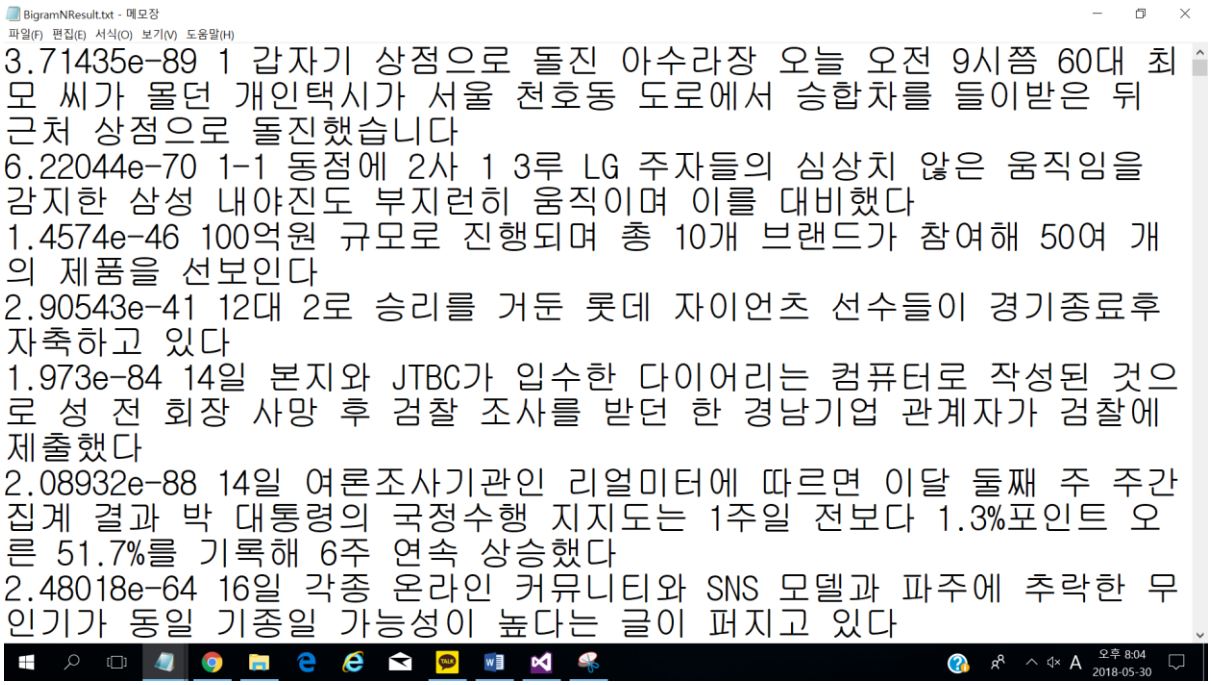
다음과 같이 분석할 입력과 출력이 있다면 한번에 모든 문장 확률을 구할 수 있다.

다음은 testN에 관한 결과 파일들이다.

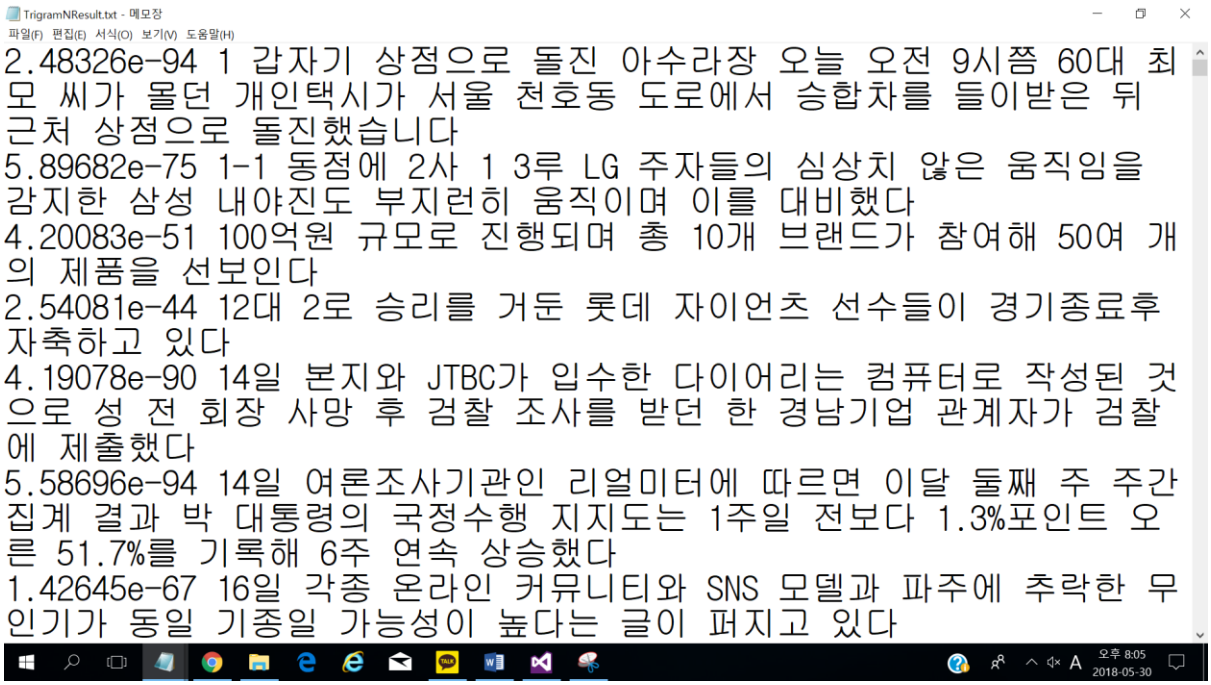
1) unigram



2) bigram

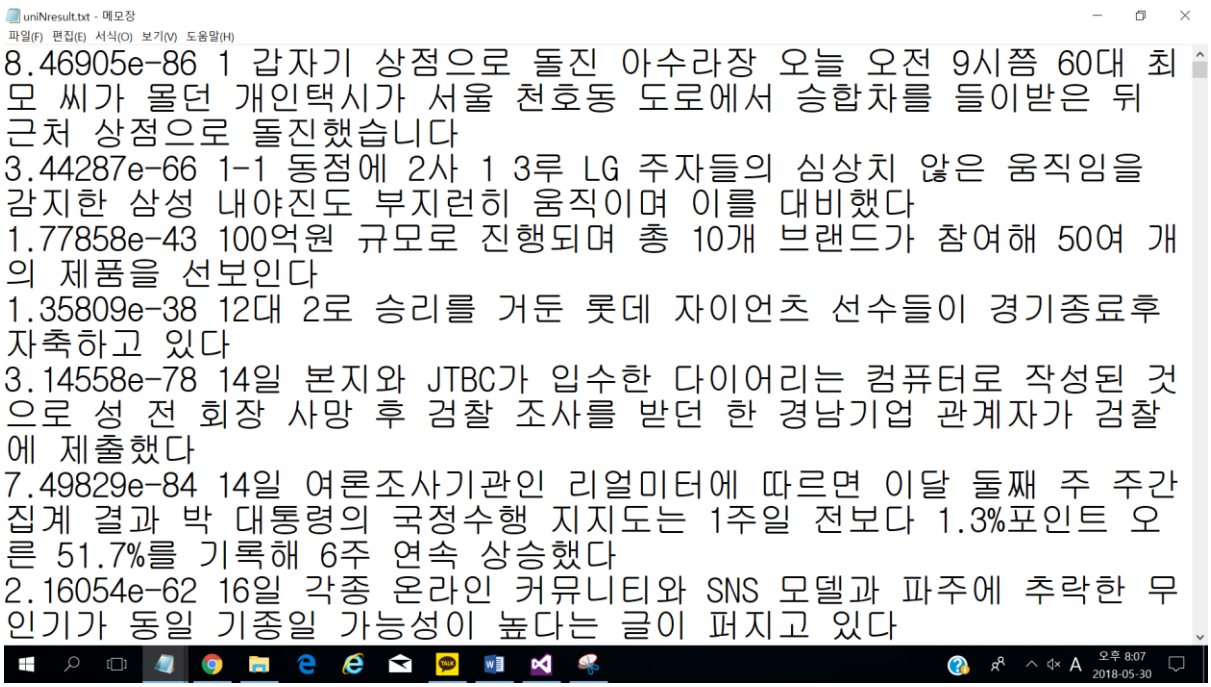


3) trigram

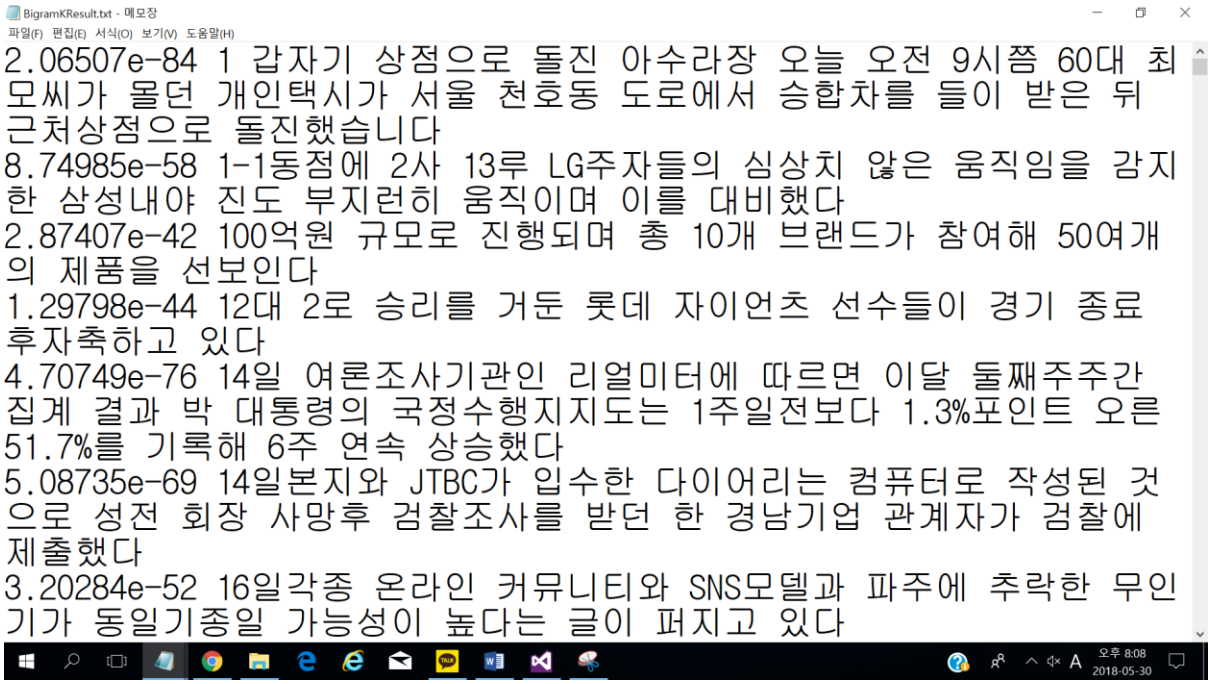


다음은 testK에 관한 결과 파일이다.

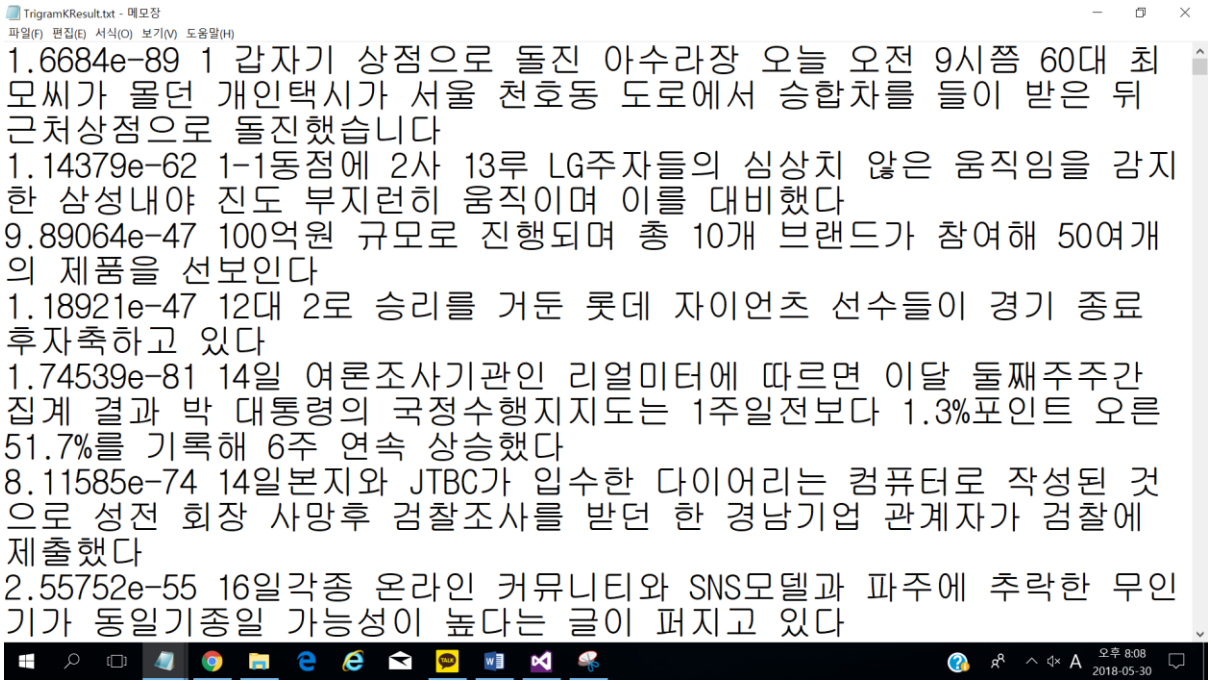
1) Unigram



2) bigram



3) trigram



5. 문장 생성 확률이 더 높은 개수에 관하여

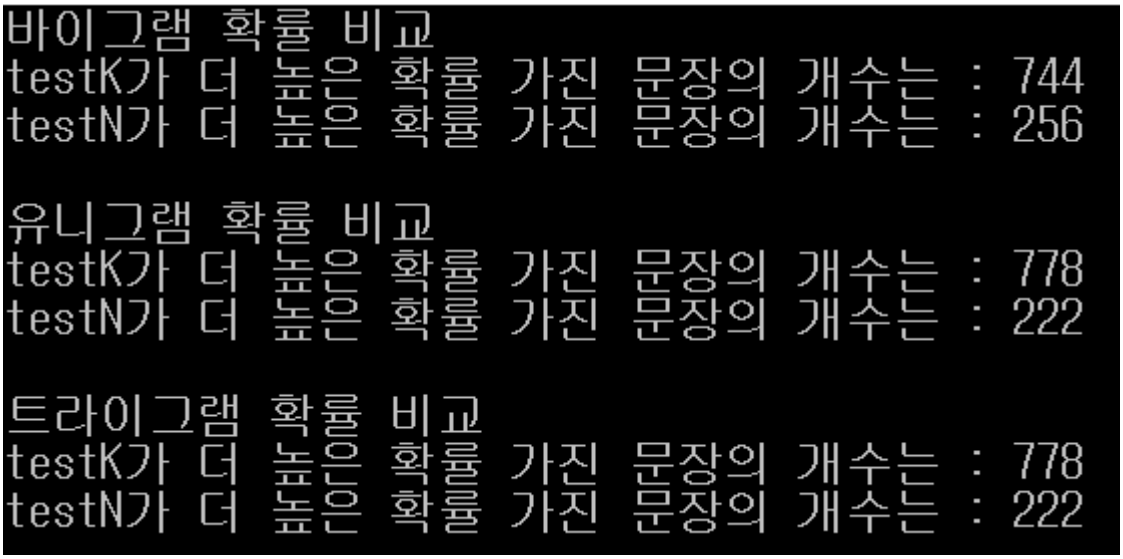
```
void compareSentence(const char* _path1, const char* _path2)
```

```
while (getline(iss, temp1, '₩n') && getline(iss2, temp2, '₩n'))
{
    iss3.str(temp1);
    iss4.str(temp2);
    iss3 >> num1;
    iss4 >> num2;
    if (atof(num1.c_str()) > atof(num2.c_str())) count1++;
    else count2++;
}
```

```
cout << "testK가 더 높은 확률 가진 문장의 개수는 : " << count1 << endl;
cout << "testN가 더 높은 확률 가진 문장의 개수는 : " << count2 << "₩n₩n";
```

확률 분석이 완료된 k와 n파일을 입력으로 받는다. 앞의 문제들과 동일한 방법으로 파일을 읽어 준다. 그 후 비교를 한다. 확률-문장 쌍으로 되어있는 파일을 줄바꿈 문장을 기준으로 각각 temp1, temp2에 가져온다. 그 다음 temp1 과 temp2를 stringstream을 이용하여 변한시킨다. 이것이 iss3, iss4이다. 그 후 확률값을 가져와 비교를 해준다. 모든 문장들에 관하여 반복해서 수행해준다.

유니그램, 바이그램, 트라이그램에 관하여 비교한 결과는 다음과 같다.



모든 경우에서 testK가 더 높은 문장확률을 가지고 있다.