Информационный поиск

Домашнее задание 2

Отчет “Неточная кластеризация”

В данном задании требовалось написать программу которая бы разделила коллекцию страниц http://simple.wikipedia.org/wiki на группы “неточных дублей”. То есть таких, что 64-битные сигнатуры страниц отличаются не более чем в n битах. Так же требовалось предоставить гистограммы распределения размеров кластеров в зависимости от n. И топ-10 результатов главных страниц кластеров по размеру оных.

Код написан на языке C++. После выполнения предыдущего задания было скачано и использовалось в данном задании 10000 страниц. Которые были сохранены(в формате чистого текста без разметки) в 400 файлов по 25 страниц на файл.(Поскольку сохранять 10К 1-килобайтных файлов лично мне критически не хотелось ни под каким предлогом).

Программа состоит из 2-х файлов. Первый из которых проводит парсинг данных 400-от документов и создает симхэши для каждой страницы. Второй соответственно делает кластеризацию для по полученным данным для некоторого n.

Гистограммы созданы средствами ipython-notebook по txt-файлу с данными.

Описание основных деталей алгоритма.

- Отдельной подзадачей является выделение страниц внутри одного из 400 файлов.
  Для этого можно заметить и проверить, что на каждой странице вики есть фраза “Jump to navigation” и встречается она ровно один раз в начале(+- 5 слов).
  По ним можно аккуратно разбить файл на страницы.
- Рассмотрим как строится симхэш для страницы вики без разметки(то есть последовательность слов). Будем действовать **методом гиперплоскостей**. В 37-мерном пространстве с 64-мя разделяющими полуплоскостями с координатами в рамках -100, +100.
- Таким образом, сначала построим 64 случайных 37-мерных вектора. Которые и будут разделителями.
  Далее для каждого слова из документа будем строить 6 ее хэш-функций(полиномиальных с разным основанием). Первые 3 будут

использованы для определения индексов(mod 37), а вторая тройка для обозначения сколько добавлять/отнимать от соответствующей позиции в симхэше.
(в рамках -10 + 10). Все положительные координаты меняются на +1 иначе на 0.

- Преимущества именно такого подхода по отношения например к простейшему сложению 64-битных хэшей для всех слов и нормализации в том что в последнем случае идет очень сильная концентрация 1-ек в младших битах и практически все старшие биты во всех симхэшах остаются 0-ми. Это происходит поскольку в любом документе количество маленьких по длине слов с маленьких хэшом значительнее чем длинных слов.

```
inline unsigned long long getWordHash(string word, unsigned long long powBasis) {
    unsigned long long res = 0,
                       power = 1;
    for (int i = 0; i < word.size(); i ++) {
      res = res + (word[i] - 'a' + 1)*1ull*power;
      power = power * powBasis;
    }
    return res;
}
inline vector<int> getSimHash(vector<string> &words) {
    vector<int> temp(Nsize, 0);
    vector<int> simhash(bits, 0);
    for (int i = 0; i < words.size(); i ++) {
      for (int j = 0; j < 3; j ++) {
        int idx = getWordHash(words[i], powerBasis[j]) % Nsize;
        int add = getWordHash(words[i], powerBasis[j + 1]) % 11 - 5;
        temp[idx] += add;
      }
    }
    for (int i = 0; i < bits; i ++) {
      long long Q = 0;
      for (int j = 0; j < Nsize; j ++)
        Q += temp[j] * 1ull * basis[i][j];
      simhash[i] = (Q > 0);
    }
    return simhash;
}
```

Алгоритмы кластеризации на неточные дубли.

- Сперва заметим что для предложенных n = 5,10, 15 не имеет смысла пользоваться очевидными способами с битовым деревом. Это не даст никакого реального роста скорости по сравнению с сложностью реализации.

Тем не менее **отсортируем страницы по размеру**(количеству слов). Так как нам в любом случае придется находить ребра-отношения дублей и данная оптимизации будет не вредной.

- Очевидно, что можно рассматривать данную структуру как граф с вершинами в виде страниц и ребрами – отношениям дублевости. Экспериментально получены следующие результаты для n >= 13(>= 10) количество ребер в таком графе будет достаточно велико (на выборке 10К страниц) что бы строить их заранее и обрабатывать целостный граф было бы неудобною. Таким образом в данном случае неплохим будет следующий алгоритм.
- Сохраняя все страницы в сет, упорядоченный по количеству слов в документе. Выбираем случайный документ из него(приоритетно ближе к середине), находим для него границы в которых вообще имеет смысл искать полу-дубли и сравниваем со всеми ними. Для этого удобно хранить симхэши либо в битсете, либо сразу в LL.
  Изначально выбранный документ считаем главным, а все найденные к нему полу-дубли образуют группу. Все найденные документы сразу удаляются из сета.

  Именно по такому алгоритму и будут приведены кластеризации и гистограммы.

- Тем не менее представим так же **альтернативный вариант**, которые имеет смысл использовать при малых n.(И не таком большом размере выборки как у меня) Проверяя все пары находим все ребра в явном виде и строим граф(ребер будет асимптотически не много). Далее запускаем на таком графе почти топологическую сортировку. Или, иначе говоря, каждый раз выбираем вершину которая имеет наибольшую выходящую степень. Удаляем группу. А так же обрабатываем все ребра которые выходят из данной группы и декрементируем их вторые концы, для поддержания корректности структуры. Учитывая небольшое количество ребер и log(N) на изменения. Получаем приемлемое время работы и хороший итоговый результат. Поскольку предыдущим алгоритмом слишком часто случайный выбор главной страницы попадает на группу размера 1.

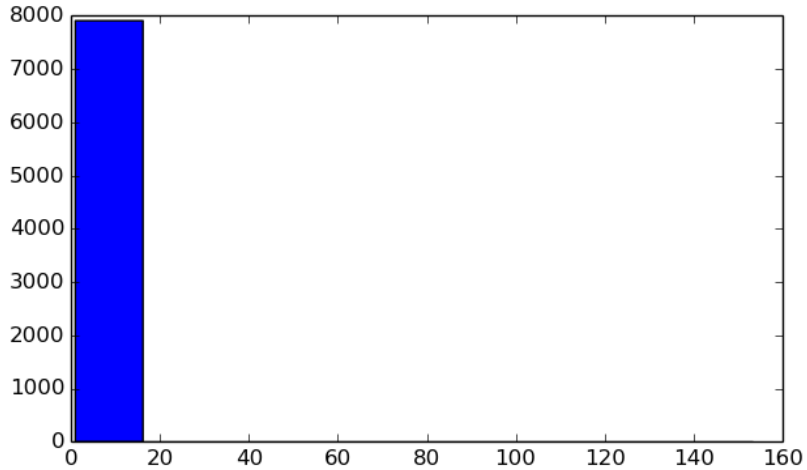
```
int it = 0;
while (!keeper.empty()) {
    int val = 300 + rand() % 600;
    set<record> :: iterator basic= keeper.lower_bound(record(val));

    int groupSize = 1;
    it ++;
    val = basic -> len;
    cluster[it].push_back(basic -> url);
    bitset<bits> current = basic -> simhash;
    keeper.erase(basic);

    set<record> :: iterator leftIT = keeper.lower_bound(record((int)(val * 0.8)));
    set<record> :: iterator rightIT = keeper.lower_bound(record((int)(val * 1.2) + 2));
    while (leftIT != rightIT) {
        bool halfReplica = (current ^ (leftIT -> simhash)).count() <= bitDiff;
        if (!halfReplica) {
          leftIT ++;
          continue;
        }
        groupSize ++;
        cluster[it].push_back(leftIT -> url);
        set<record>::iterator copied = leftIT;
        leftIT ++;
        keeper.erase(copied);
    }
}
```

Приведем построенные гистограммы для размера групп при разных N(допустимое расхождения в колве бит).

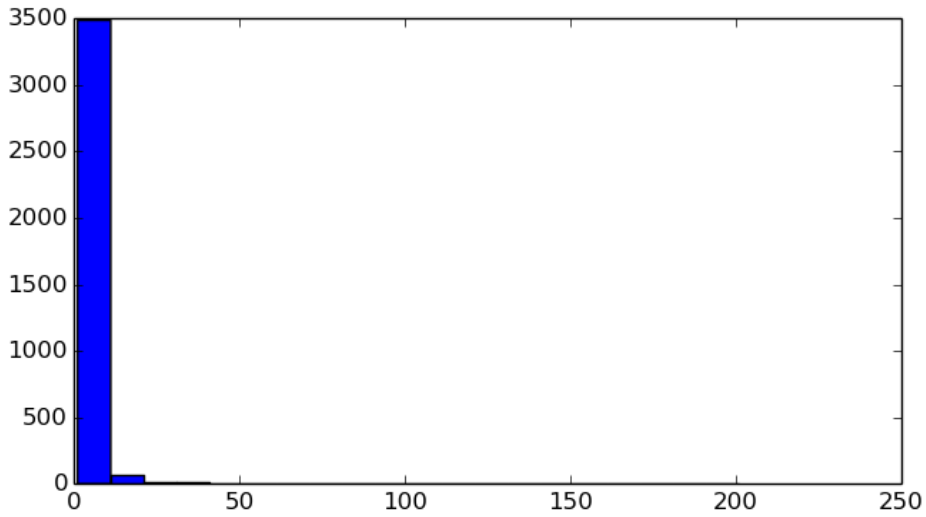
- **N = 5**



1) **Size = 153: http://simple.wikipedia.org/wiki/Bowling_Green,_Florida**
2) **Size = 60: http://simple.wikipedia.org/wiki/September_20**
3) **Size = 51: http://simple.wikipedia.org/wiki/July_16**
4) **Size = 48: http://simple.wikipedia.org/wiki/December_9**
5) **Size = 43: http://simple.wikipedia.org/wiki/July_5**
6) **Size = 23: http://simple.wikipedia.org/wiki/Arabic_Wikipedia**

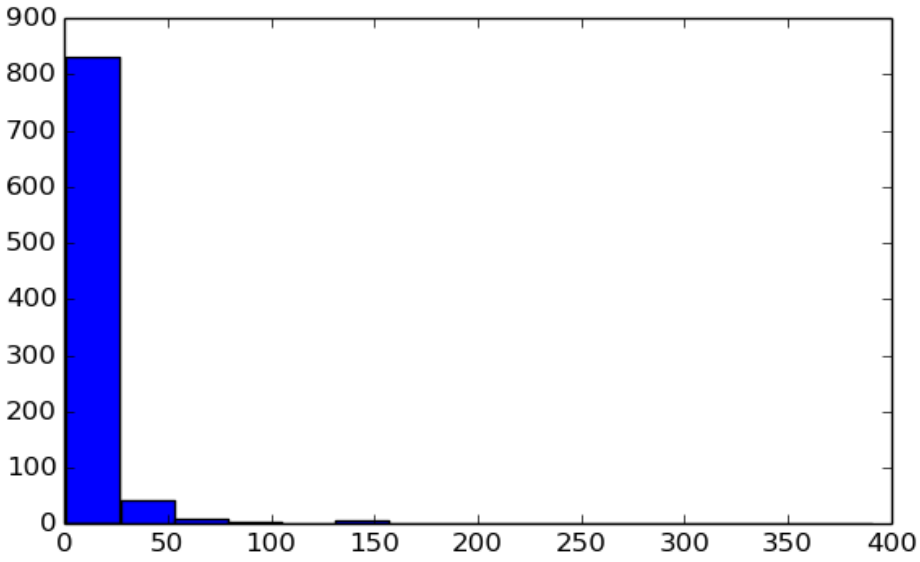
7) Size = 21: http://simple.wikipedia.org/wiki/April_11
8) Size = 19: http://simple.wikipedia.org/wiki/February_19
9) Size = 17: http://simple.wikipedia.org/wiki/Escambia_County,_Florida
10) Size = 15: http://simple.wikipedia.org/wiki/October_16

- **N = 10**



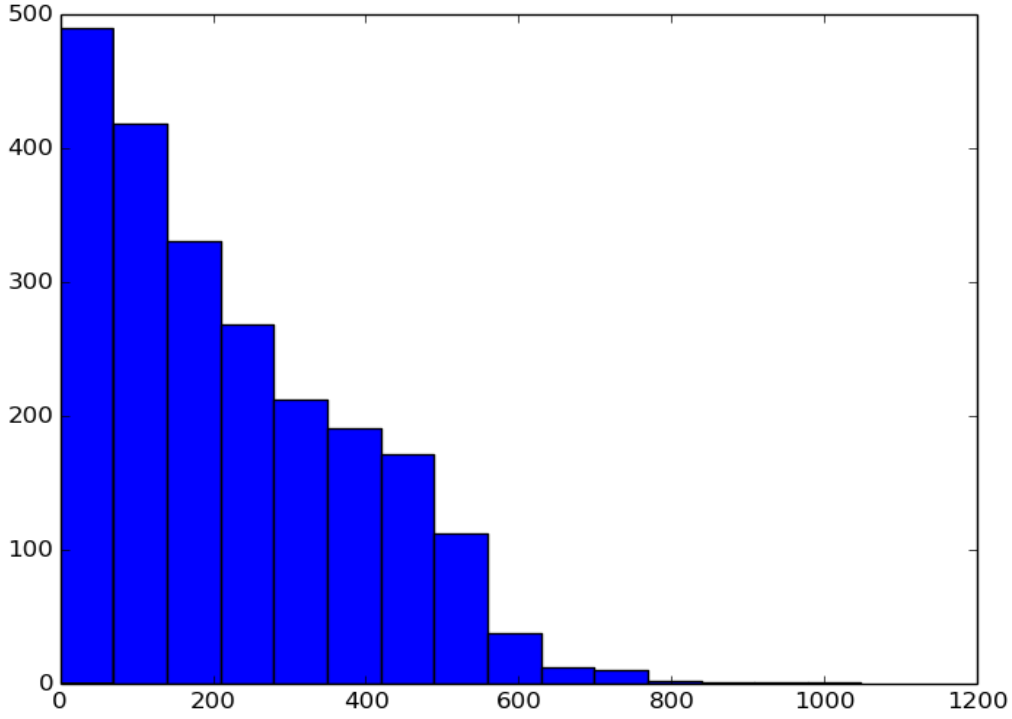
1) Size = 201: http://simple.wikipedia.org/wiki/November_23
2) Size = 165: http://simple.wikipedia.org/wiki/Bowling_Green,_Florida
3) Size = 99: http://simple.wikipedia.org/wiki/July_16
4) Size = 82: http://simple.wikipedia.org/wiki/1894
5) Size = 73: http://simple.wikipedia.org/wiki/Arabic_Wikipedia
6) Size = 71: http://simple.wikipedia.org/wiki/Apprentice
7) Size = 65: http://simple.wikipedia.org/wiki/Bel_and_the_Dragon
8) Size = 58: http://simple.wikipedia.org/wiki/Camera
9) Size = 53: http://simple.wikipedia.org/wiki/Arcadia,_Florida
10) Size = 49: http://simple.wikipedia.org/wiki/1764

- **N = 15:**



1) Size = 391: http://simple.wikipedia.org/wiki/Bel_and_the_Dragon
2) Size = 338: http://simple.wikipedia.org/wiki/Ape
3) Size = 325: http://simple.wikipedia.org/wiki/Bible_translations
4) Size = 255: http://simple.wikipedia.org/wiki/Algebraic_structure
5) Size = 237: http://simple.wikipedia.org/wiki/Geometric
6) Size = 234: http://simple.wikipedia.org/wiki/Chitin
7) Size = 212: http://simple.wikipedia.org/wiki/Consonant
8) Size = 186: http://simple.wikipedia.org/wiki/Fact
9) Size = 168: http://simple.wikipedia.org/wiki/Bowling_Green,_Florida
10) Size = 153: http://simple.wikipedia.org/wiki/January_13

Гистограмма расстояния между проверяемыми документами
(после логарифмирования ординаты, вследствии слишком больших значений)



Вывод:

Как видим, при большем N мы получаем большие размеры групп(при меньшем их количестве) что вполне естественно. При N порядка 15 имеем более чем в 10 раз меньше групп нежели страниц при анализе. Тогда как при 5 уменьшается едва ли на треть. Для последнего случая рандомизированный выбор главного представителя группы и поиск от него дает недостаточно хорошие результаты.

Для построения симхэша лучше использовать методы которые равновероятно распределяеют информацию между всеми битами, в противном случае такое их количество не имеет никакого смысла. Был сделан вывод о плохом соответствии простейшего алгоритма сим-хэширования для данной задачи, в то время как метод разделяющих гиперплоскостей подходит достаточно неплохо. Для слов можно использовать обычный полиномиальный хэш с вариативным основанием.

Исходя из распределения разниц размеров проверяемых документов можно сделать вывод что эта величина имеет логнормальное распределение(ну или экспоненциальное) (по крайне мере без логарифмирования координат получались слишком большие числа, иначе же результат представлен на графике выше и говорит сам за себя).