Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Курсовой проект по курсу «Дискретный анализ»

Студент: А.Д. Волков
Преподаватель: С. А. Сорокин
Группа: М8О-306Б

Дата: 03.01.2025
Оценка:
Подпись:

Москва, 2025

Курсовой проект

Задача: Реализуйте систему, которая на основе базы вопросов и тегов к ним, буден предлагать варианты тегов, которые подходят к новым вопросам.
Формат запуска программы в режиме обучения:

```
./prog learn --input <input file> --output <stats file>
```

—**input** – входной файл с вопросами
—**output** – выходной файл с рассчитанной статистикой
Формат запуска программы в режиме классификации:

```
./prog classify --stats <stats file> --input <input file> --output <output file>
```

–**stats** – файл со статистикой полученной на предыдущем этапе
–**input** – входной файл с вопросами
–**output** – выходной файл с тегами к вопросам
Формат входных файлов при обучении:

```
<Количество строк в вопросе [n]>
<Тег 1>,<Тег 2>,...,<Тег m>
<Заголовок вопроса>
<Текст вопроса [n строк]>
```

Формат входных файлов при запросах:

```
<Количество строк в вопросе [n]>
<Заголовок вопроса>
<Текст вопроса [n строк]>
```

Формат выходного файла: для каждого запроса в отдельной строке выводится предполагаемый набор тегов, через запятую.

1 Описание

Требуется реализовать наивный байесовский классификатор. Наивный байесовский классификатор — вероятностный классификатор на основе формулы Байеса со строгим (наивным) предположением о независимости признаков между собой при заданном классе, что сильно упрощает задачу классификации из-за оценки одномерных вероятностных плотностей вместо одной многомерной.
В данном случае, одномерная вероятностная плотность — это оценка вероятности каждого признака отдельно при условии их независимости, а многомерная — оценка вероятности комбинации всех признаков, что вытекает из случая их зависимости. Именно по этой причине данный классификатор называется наивным, поскольку позволяет сильно упростить вычисления и повысить эффективность алгоритма. Сама формула Байеса выглядит следующим образом:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

где:
$P(A|B)$ - апостериорная вероятность события A при условии выполнения события B;
$P(B|A)$ — условная вероятность события B при условии выполнения события A;
$P(A)$ и $P(B)$ — априорные вероятности событий A и B соответственно.
А в контексте машинного обучения формула Байеса приобретает следующий вид:

$$P(y_k|X) = \frac{P(y_k)P(X|y_k)}{P(X)}$$

где:
$P(y_k|X)$ – апостериорная вероятность принадлежности образца к классу y_k с учётом его признаков X;
$P(X|y_k)$ – правдоподобие, то есть вероятность признаков X при заданном классе y_k;
$P(y_k)$ – априорная вероятность принадлежности случайно выбранного наблюдения к классу y_k;
$P(X)$ — априорная вероятность признаков X.
Если объект описывается не одним, а несколькими признаками $X_1, X_2, ..., X_n$, то формула принимает вид:

$$P(y_k|X_1, X_2, ..., X_n) = \frac{P(y_k)\prod_{i=1}^{n} P(X_i|y_k)}{P(X_1, X_2, ..., X_n)}$$

На практике числитель данной формулы представляет наибольший интерес, поскольку знаменатель зависит только от признаков, а не от класса, и поэтому часто он опускается при сравнении вероятностей разных классов. В конечном счёте правило классификации будет пропорционально выбору класса с максимальной апостериорной вероятностью:

$$y_k \propto argmax_{y_k} P(y_k) \prod_{i=1}^{n} P(X_i|y_k)$$

Для оценки параметров модели, то есть вероятностей $P(y_k)$ и $P(X_i|y_k)$, обычно применяется метод максимального правдоподобия, который в данном случае основан на частотах встречаемости классов и признаков в обучающей выборке. Также, во избежание больших погрешностей вероятности, каждую составляющую формулы мы логарифмируем и нормализуем, так как у нас мультиклассовый классификатор. Нормализуем с помощью формулы $softmax$:

$$softmax(x_i) = \frac{e^{x_i}}{\Sigma_{j=1}^{n} e^{x_j}}$$

Также, так как в формуле появляется логарифм, по свойствам логарифма вместо произведений и делений появляются сложения и вычитания, что дает меньшую погрешность при вычислении вероятностей. [1]

2 Исходный код

Сначала мы считываем данные тренировочной выборки. В вектор target мы заносим теги для каждого вопроса, а в вектор train_sample мы заносим сами вопросы. Далее мы тренируем модель по считанным данным.
Разберемся как работает метод fit у объекта класса TNaiveBayes. Для начала мы сохраняем количество вопросов в тренировочном файле. Это пригодится нам для восстановления данных модели при предсказании. Далее мы заполняем вектор class_labels для хранения всех уникальных тэгов. Затем мы инициализируем словари для подсчета слов с определенным тэгом и априорными вероятностями тэгов. Далее мы подсчитываем статистики для каждого слова, подобно как мы это делали в предыдущей работе, только во время подсчета статистик мы также сохраняем информацию о частоте встреч слов с определенным тэгом и о частоте самих тэгов. Далее мы сохраняем количество встреченных тэгов, это тоже нам пригодится для восстановления модели из файла. Также мы логарифмируем априорную вероятность и подсчитываем ее для каждого тэга. Затем мы сохраняем нужные статистики в выходной файл и заканчиваем работу программы в режиме обучения.
В режиме классификации мы считываем файл со статистиками и просчитываем необходимые для классификации данные. Затем считываем входной файл, в котором уже даны необходимые вопросы для классификации. В методе predict мы выполняем подсчет вероятностей тэгов для каждого предложения. Расчет производится так: если слова не было в тренировочной выборке или оно не встречалось в текущем тэге, то мы высчитываем вероятность по формуле с логарифмами, а именно $log(1) - log(frequency(class) + n)$, если слово встречалось, то по такой формуле: $log(frequency(class, w_i) + 1) - log(frequency(class) + n)$. Все эти вероятности мы теперь должны сложить и прибавить к ним логарифмированную априорную для текущего тэга. Стоит сказать, что все эти значения формально не являются вероятностями, а лишь значениями логарифмов, но после дальнейшего применения функции softmax все эти значения станут вероятностями. Далее мы сортируем полученные вероятности по убыванию и отдаем только те тэги, вероятности которых превышают определенный порог, который подбирается империческим путем. Все это выводится в выходной файл и программа завершает свою работу.

```
1  #include <bits/stdc++.h>
2
3  const double PREDICTION_EDGE = 0.1;
4
5  std::vector<std::string> WordsSplit(std::string sentence)
6  {
7      std::stringstream ss(sentence);
8      std::vector<std::string> words;
9      const char* const delimeters = "!?., ";
10     std::string line;
```

```
11      while (std::getline(ss, line))
12      {
13          char *token = std::strtok(line.data(), delimeters);
14          while (token != nullptr)
15          {
16              words.push_back(token);
17              token = std::strtok(nullptr, delimeters);
18          }
19      }
20      return words;
21  }
22
23  std::string ToLower(std::string input)
24  {
25      for (size_t i = 0; i < input.size(); ++i)
26      {
27          input[i] = std::tolower(static_cast<unsigned char>(input[i]));
28      }
29      return input;
30  }
31
32  long double max_val_vec(std::vector<std::pair<std::string, long double>>& vec)
33  {
34      long double max_value = std::numeric_limits<long double>::lowest();
35
36      for (const auto& p : vec) {
37          if (p.second > max_value) {
38              max_value = p.second;
39          }
40      }
41      return max_value;
42  }
43
44  std::vector<std::pair<std::string, long double>> softmax(std::vector<std::pair<std::
        string, long double>> vec)
45  {
46      std::vector<std::pair<std::string, long double>> result;
47      long double max_val = max_val_vec(vec);
48      long double exp_sum = 0;
49      std::vector<long double> exp_values;
50      for (size_t i = 0; i < vec.size(); ++i)
51      {
52          long double exp_val = std::exp(vec[i].second - max_val);
53          exp_values.push_back(exp_val);
54          exp_sum += exp_val;
55      }
56
57      for (size_t i = 0; i < vec.size(); ++i)
58      {
```

```
59          result.push_back({vec[i].first, exp_values[i] / exp_sum});
60      }
61      return result;
62  }
63
64  class TWordStats
65  {
66  public:
67      std::unordered_map<std::string, int> frequencies; // frequencies of word for each
            class
68  };
69
70  class TNaiveBayes
71  {
72  private:
73      std::unordered_map<std::string, TWordStats> word_stats;
74      std::unordered_map<std::string, long double> prior_probabilities;
75      std::size_t dict_size;
76      std::unordered_map<std::string, int> freqs;
77      std::vector<std::string> class_labels;
78      int questions_quantity;
79      std::unordered_map<std::string, int> tags_quantity;
80
81      std::unordered_map<std::string, int> sum_freqs()
82      {
83          std::unordered_map<std::string, int> counts;
84          for (auto& pair : this->word_stats)
85          {
86              for (const auto& class_label : class_labels)
87              {
88                  counts[class_label] += pair.second.frequencies[class_label];
89              }
90          }
91          return counts;
92      }
93
94      long double probability_calculator(std::vector<std::string>& words, const std::
            string& class_label)
95      {
96          long double probability = 0;
97          for (const auto& word : words)
98          {
99              if (this->word_stats.count(word) == 0 || this->word_stats[word].frequencies
                    [class_label] == 0)
100             {
101                 // Laplas smoothing
102                 probability += log(1.0) - log(freqs[class_label] + dict_size);
103             }
104             else
```

```
105             {
106                 probability += log(this->word_stats[word].frequencies[class_label] + 1)
                        - log(freqs[class_label] + dict_size);
107             }
108         }
109         probability += prior_probabilities[class_label];
110         return probability;
111     }
112 
113 public:
114     void fit(std::vector<std::string>& X, std::vector<std::vector<std::string>>& y)
115     {
116         questions_quantity = X.size();
117         std::set<std::string> unique_classes;
118         for (const auto& tags : y)
119         {
120             for (const auto& tag : tags)
121             {
122                 unique_classes.insert(tag);
123             }
124         }
125         class_labels.assign(unique_classes.begin(), unique_classes.end());
126 
127         for (const auto& class_label : class_labels)
128         {
129             prior_probabilities[class_label] = 0;
130             freqs[class_label] = 0;
131         }
132 
133         for (size_t i = 0; i < X.size(); ++i)
134         {
135             std::vector<std::string> words = WordsSplit(ToLower(X[i]));
136             for (const auto& class_label : y[i])
137             {
138                 for (const auto& word : words)
139                 {
140                     if (this->word_stats.count(word))
141                     {
142                         word_stats[word].frequencies[class_label]++;
143                     }
144                     else
145                     {
146                         TWordStats stats;
147                         stats.frequencies[class_label] = 1;
148                         word_stats[word] = stats;
149                     }
150                     freqs[class_label]++;
151                 }
152                 prior_probabilities[class_label] += 1;
```

```
153             }
154         }
155         for (auto& class_label : class_labels)
156         {
157             tags_quantity[class_label] = prior_probabilities[class_label];
158             prior_probabilities[class_label] = log(prior_probabilities[class_label]) -
                    log(X.size());
159         }
160         this->dict_size = word_stats.size();
161     }
162 
163     std::vector<std::string> predict(std::string X)
164     {
165         std::vector<std::string> words = WordsSplit(ToLower(X));
166         std::vector<std::pair<std::string, long double>> prediction;
167         for (const auto& class_label : class_labels)
168         {
169             long double probability = this->probability_calculator(words, class_label);
170             prediction.push_back({class_label, probability});
171         }
172         prediction = softmax(prediction);
173         std::sort(prediction.begin(), prediction.end(), [](const std::pair<std::string,
                long double>& a, const std::pair<std::string, long double>& b)
174         {
175             return a.second > b.second;
176         });
177         std::vector<std::string> result;
178         for (size_t i = 0; i < prediction.size(); i++)
179         {
180             if (prediction[i].second > PREDICTION_EDGE)
181             {
182                 result.push_back(prediction[i].first);
183             }
184         }
185         return result;
186     }
187 
188     void save_stats(const std::string& filename)
189     {
190         std::ofstream ofs(filename);
191         ofs << questions_quantity << "\n";
192         ofs << dict_size << "\n";
193         ofs << class_labels.size() << "\n";
194         for (const auto& class_label : class_labels)
195         {
196             ofs << class_label << " " << tags_quantity[class_label] << "\n";
197         }
198         for (auto& pair : word_stats)
199         {
```

```
200             ofs << pair.first;
201             for (const auto& class_label : class_labels)
202             {
203                 ofs << " " << pair.second.frequencies[class_label];
204             }
205             ofs << "\n";
206         }
207     }
208 
209     void load_stats(const std::string& filename)
210     {
211         std::ifstream ifs(filename);
212         ifs >> questions_quantity;
213         ifs >> dict_size;
214         class_labels.clear();
215         prior_probabilities.clear();
216         tags_quantity.clear();
217         std::string class_label;
218         size_t tags;
219         ifs >> tags;
220         for (size_t _ = 0; _ < tags; _++)
221         {
222             std::string class_label;
223             ifs >> class_label;
224             class_labels.push_back(class_label);
225             ifs >> tags_quantity[class_label];
226             prior_probabilities[class_label] = log(tags_quantity[class_label]) - log(
                    questions_quantity);
227         }
228         word_stats.clear();
229         std::string word;
230         while (ifs >> word)
231         {
232             TWordStats stats;
233             for (const auto& classes : class_labels)
234             {
235                 int freq;
236                 ifs >> freq;
237                 stats.frequencies[classes] = freq;
238             }
239             word_stats[word] = stats;
240         }
241         freqs = sum_freqs();
242     }
243 };
244 
245 int main(int argc, char* argv[])
246 {
247     if (argc < 5)
```

```
248     {
249         std::cerr << "./a.out learn --input <input file> --output <stats file>\n"
250                   << "./a.out classify --stats <stats file> --input <input file> --
                          output <output file>\n";
251         return 1;
252     }
253 
254     std::string command = argv[1];
255     TNaiveBayes classifier;
256 
257     if (command == "learn")
258     {
259         std::string input_file = argv[3];
260         std::string output_file = argv[5];
261 
262         std::vector<std::string> train_sample;
263         std::vector<std::vector<std::string>> target;
264 
265         std::ifstream infile(input_file);
266 
267         while (infile)
268         {
269             int num_lines;
270             infile >> num_lines;
271             if (infile.eof())
272             {
273                 break;
274             }
275             infile.ignore();
276 
277             std::string tags_line;
278             std::getline(infile, tags_line);
279             if (tags_line.empty())
280             {
281                 break;
282             }
283             std::vector<std::string> tags = WordsSplit(tags_line);
284 
285             std::string question_title;
286             std::getline(infile, question_title);
287 
288             std::string question_text;
289             for (int i = 0; i < num_lines; ++i)
290             {
291                 std::string line;
292                 std::getline(infile, line);
293                 question_text += line + "\n";
294             }
295 
```

```
296             train_sample.push_back(question_title + "\n" + question_text);
297             target.push_back(tags);
298         }
299 
300         classifier.fit(train_sample, target);
301         classifier.save_stats(output_file); // statistics saving
302     } else if (command == "classify")
303     {
304         std::string stats_file = argv[3];
305         std::string input_file = argv[5];
306         std::string output_file = argv[7];
307 
308         classifier.load_stats(stats_file); // statistics loading
309 
310         std::ifstream infile(input_file);
311         std::ofstream outfile(output_file);
312 
313         while (infile)
314         {
315             if (infile.eof())
316             {
317                 break;
318             }
319             int num_lines;
320             infile >> num_lines;
321             infile.ignore();
322 
323             std::string question_title;
324             std::getline(infile, question_title);
325 
326             std::string question_text;
327             for (int i = 0; i < num_lines; ++i)
328             {
329                 std::string line;
330                 std::getline(infile, line);
331                 question_text += line;
332             }
333             std::vector<std::string> prediction = classifier.predict(question_title + "
                \n" + question_text);
334             if (prediction.size() != 0)
335             {
336                 outfile << prediction[0];
337             }
338             for (size_t i = 1; i < prediction.size(); ++i)
339             {
340                 outfile << ", " << prediction[i];
341             }
342             outfile << "\n";
343         }
```

```
    } else
    {
        std::cerr << "Unknown command: " << command << "\n";
        return 1;
    }
}
```

main.cpp	
std::vector<std::string> WordsSplit()	Функция разбиения документа на слова
std::string ToLower()	Функция перевода строки в нижний регистр
long double max_val_vec()	Функция для нахождения максимального элемента в векторе
std::vector<> softmax()	Функция для подсчета формулы softmax
std::unordered_map<> TNaiveBayes::sum_freqs()	Метод для суммирования всех частот в каждом классе
long double TNaiveBayes::probability_calculator()	Метод для подсчета вероятности принадлежности тестовых документов к определенному классу
void TNaiveBayes::fit()	Метод для обучения модели
std::vector<std::string> predict()	Метод для предсказания тэга отдельно взятого вопроса
void save_stats()	Метод для сохранения статистики в файл
void load_stats()	Метод для загрузки статистики из файла
int main()	Точка входа программы

```
class TWordStats
{
public:
    std::unordered_map<std::string, int> frequencies;
};
class TNaiveBayes
{
private:
    std::unordered_map<std::string, TWordStats> word_stats;
    std::unordered_map<std::string, long double> prior_probabilities;
    std::size_t dict_size;
    std::unordered_map<std::string, int> freqs;
    std::vector<std::string> class_labels;
    int questions_quantity;
```

```
15      std::unordered_map<std::string, int> tags_quantity;
16      std::unordered_map<std::string, int> sum_freqs();
17      long double probability_calculator(std::vector<std::string>& words, const std::
            string& class_label);
18  public:
19      void fit(std::vector<std::string>& X, std::vector<std::vector<std::string>>& y);
20      std::vector<std::string> predict(std::string X);
21      void save_stats(const std::string& filename);
22      void load_stats(const std::string& filename);
23  };
```

3 Консоль

```
lexasy@lexasy$ g++ nonbinary.cpp
lexasy@lexasy$ ./a.out learn --input learn.txt --output output.txt
lexasy@lexasy$ ./a.out classify --stats output.txt --input test.txt --output
result.txt
lexasy@lexasy$ cat result.txt
technology,innovation,future
fitness
culture
education
entrepreneurship,business,startups
creativity,expression,art
fitness,sports,competition
```

4 Тест производительности

Тест производительности представляет из себя замер метрик классификации, таких как полнота и точность. Проверим на датасете который мы использовали в консоли. В том датасете есть 15 вопросов в тренировочной выборке и 7 вопросов в тестовой выборке, что является довольно большим датасетом. Посмотрим какие метрики получили наши предсказания:

```
lexasy@lexasy$ python3 presision_recall.py
AVG PRESISION: 0.42857142857142855
AVG RECALL: 0.5
```

Как мы видим метрики получились не лучшие. Но на самом деле, для модели, которая практически ничего не учитывает, кроме частоты слова, и то, слово во множественном числе и в единственном числе это разные слова, то это еще даже довольнотаки неплохой результат. Плюс стоит учесть что это практически реальный текст, который неподготовлен под работу наивного классификатора. Попробуем взять датасет полегче. Теперь в датасете будет всего 7 тренировочных вопросов и 3 тестовых, посмотрим на то какие будут метрики:

```
lexasy@lexasy$ python3 presision_recall.py
AVG PRESISION: 0.7777777777777778
AVG RECALL: 0.6666666666666666
```

Видим уже удовлетворительные метрики. В принципе, ничего удивительного, что на больших данных мы получаем результат хуже, ведь даже несмотря на логарифмирование все-равно с каждым новым вопросом погрешность увеличивается, что дает знать о себе. Также стоит сказать, что метрики будут зависеть от выбранного порога вероятности, в данном случае я выбрал порог 0.1, так как при таком пороге я получал мало случаев, когда программа вообще не сделала предсказаний и метрики при таком пороге получились более менее адекватными.

5 Выводы

Выполнив курсовую работу по курсу Дискретный Анализ я научился реализовывать модель наивного байесовского классификатора. Также я подтянул знания по классификаторам из курса машинного обучения и по методу максимального правдоподобия из курса математической статистики. Но реализованная мною модель точно не работает в практических задачах, так как она игнорирует множество признаков. Но скорее всего, данный курсовой проект позволит столкнуться с меньшим количеством трудностей при знакомстве с реальным текстовым классификатором.

Список литературы

[1] https://habr.com/ru/articles/802435/ - 2024