1. Обзор проблематики.

Каждый день мир узнает имена новых писателей, блогеров, публицистов. В месте с этим в Интернет загружается огромное количество текстовой информации. Поэтому процесс поиска источников информации (вне зависимости от типа электронного ресурса) и выбор из них нужных затрудняется. Появляется необходимость в том, чтобы сделать объёмы информации более доступными для восприятия. Для этого можно разбить текстовые документы на тематические группы. Тогда можно выбирать из них релевантные для пользователя, а мало-релевантные группы просто отбрасывать.

Проблема проекта – неупорядоченность текстовой информации в Интернете.

Цель проекта – разработать программное решение, объединяющее заданные текстовые документы в группы на основе их семантической схожести.

Группировку, описанную в цели проекта, можно осуществить с помощью кластерного анализа. Кластерный анализ – это процесс распределения заданных объектов в однородные группы(кластеры). В нашем случае объектами кластерного анализа будут текстовые документы.

1. Выбор инструментов.

В качестве языка программирования я выбрал Python, ведь он предоставляет доступ к огромному количеству полезнейших библиотек, таких как NumPy, Beautiful Soup, Pandas. В качестве среды разработки я выбрал Spyder – дело личного вкуса.

1. Описание логики работы программы.

Предисловие. Я буду только описывать логику и принцип работы программы, упоминая используемые библиотеки, чтобы не нагружать документацию кодом. Ссылку на код готовой программы я оставлю в конце документации.

Компьютер – не человек, выбрать схожие слова он может только сравнив каждый символ слов. Поэтому первым делом программа обрабатывает заданные тексты таким образом, чтобы иметь возможность в дальнейшем удобно с ними работать. Каждый текст проходит процесс нормализации:

1)Текстовый документ представляется в виде набора обособленных слов.

Использована стандартная библиотека регулярных выражений «re» Python.

2)Слова, не влияющие на смысл текста, исключаются. Не влияют на смысл местоимения, предлоги и некоторые другие языковые конструкции. Они называются «стоп-словами».

Использована библиотека «stop-words».

3)Слова переводятся в нижний регистр.

4)Проводится стэмминг – выделение основы слов.

Использована библиотека «pymorphy2»

Программа в итоге работает не со всеми словами, а с оставшимися терминами. Термины непосредственно влияют на смысл текста, кроме того. Множество терминов составляет пространство.

Каждый документ теперь программа представляет как точку (радиус вектор) в этом пространстве, при этом за координаты точки документа принимает количество вхождений каждого из терминов в этот документ.

Представив все документы в виде векторов, ей можно было бы вычислить схожесть между всеми ними, используя одну из мер расстояния (расстояние между точками/косинус между радиус-векторами). Однако на данном этапе точность таких измерений мала.

Поэтому она повышается с помощью оценивания важности каждого термина. Для этого используется популярная статистическую мера TF-IDF, состоящую из двух компонентов:

TF(t, d) – (term frequency – частота слова) – вычисляется как , где n – количество вхождений термина t в документ d. Данный множитель оценивает важность t в пределах d.

IDF(t, D) – (inverse document frequency – обратная частота документа) – инверсия частоты, с которой термин t встречается в документах коллекции. Вычисляется как, где |D| - количество документов, а в знаменателе – количество документов, в которых встречается t

Совместив эти два компонента, получаем формулу tf-idf для термина t, находящегося в документе d, принадлежащему множеству документов D:

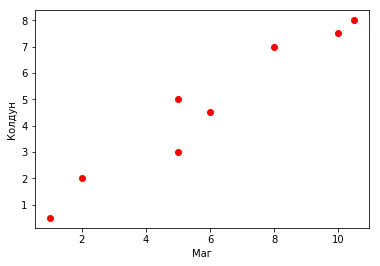
tf-idf(t, d, D) = TF(t, d) \* IDF(t, D).

Взвесив все слова, программа опять же могла вычислять схожесть между документами. Но точность всё еще может быть увеличена. Для этого нужно решить еще одну незатронутую проблему – проблему синонимии и омонимии. Максимально снизив её влияние, программа сможет наиболее точно вычислить схожесть между текстами и сгруппировать их, проведя кластерный анализ.

Главной идеей в борьбе с проблемой является то, что сравнивать нужно не сами слова, а их значения. Хорошо коррелируемые слова являются взаимозаменяемыми, а значит, имеют одинаковое значение. Программа будет объединять их!

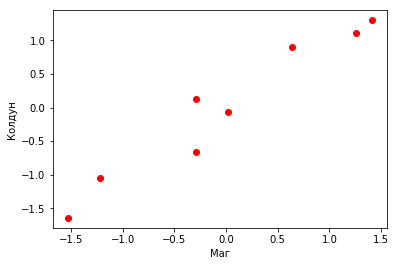
Каким образом происходит объединение, объясню на примере. Представим, что имеется некая матрица tf-idf для документов T1..6.

|  |  |  |
| --- | --- | --- |
|  | маг | колдун |
| Т1 | 1 | 0.5 |
| Т2 | 2 | 2 |
| Т3 | 5 | 3 |
| Т4 | 8 | 7 |
| Т5 | 10 | 7.5 |
| Т6 | 10.5 | 8 |
| Т7 | 5 | 5 |
| Т8 | 6 | 4.5 |

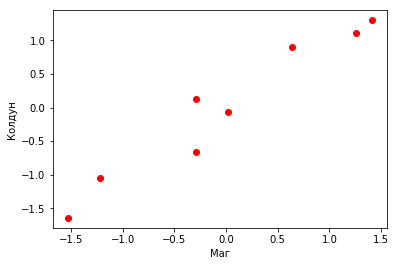


На построенной диаграмме видно, что оба слова неплохо коррелируются, ведь они взаимозаменяемы в употреблении и имеют схожий вес. Следовательно, вместо того чтобы рассматривать два эти слова как отдельные, можно объединить в одну переменную. Используем для этого метод главных компонент:

Проведем стандартную практику, центрируем и стандартизируем наши данные:



Нужно через начало отсчёта провести такую прямую, чтобы сумма квадратов расстояний от начала отсчёта до проекций точек на неё была максимальной (а вместе с ней и дисперсия).:



PC1, (волшебник)

Полученная прямая называется главной компонентой, PC1. Она «от руки» изображена на рисунке выше. Значениями проекций точек на эту прямую и будут являться значения tf-idf новой переменной «волшебник», которая объединяет два синонима – «маг» и «колдун».

Это работает, потому что мы, зная значение проекции точки, можем примерно восстановить её исходные координаты, и, следовательно, значения tf-idf для объединенных слов. «Волшебник» совмещает в себе «мага» и «колдуна», поэтому значениями проекций будет являться, на самом деле, результат линейной комбинации. Её можно посчитать, найдя собственный вектор. Я не буду углубляться в математику, однако приведу лишь небольшой пример. Допустим, нам нужно сравнить расположение двух точек – с проекциями 2 и 0.2. Очевидно, что из-за большой корреляции обе точки будут находиться вблизи прямой – мы можем лишь установить, что 2 находится дальше от центра отсчёта, а 0.2 находится почти у точки (0;0).

Восполнить потерянную информацию о изначальном расположении точки можно, используя, помимо PC1, PC2. Логично, что PC2 - прямая, перпендикулярную к PC1, ведь проекции точек на неё как раз описывают то самое недостающее отклонение от PC1.

Однако из-за малой дисперсии объем потерянной информации слишком мал и им можно пренебречь, используя информацию PC1.

В данном примере, объём информации, которую объясняет PC1, равняется 0.98 от общего и лишь 0.02 объёма объясняет PC2.

Вернемся к программе. Чтобы решить проблему синонимии и омонимии, она применяет метод главных компонент к уже полученной матрице tf-idf (используется библиотека Sklearn).

Далее формируется редуцированная матрица и программа переходит к кластерному анализу.

В качестве алгоритма кластеризации я выбрал самый популярный – «К-средних». Опишу вкратце его классическую реализацию:

1. Начальное положение центроидов кластеров выбирается случайно.
2. На каждой итерации пересчитывается центр масс каждого кластера(в моём случае, используя простое Евклидово расстояние), полученного на предыдущем шаге, после объекты снова разбиваются на кластеры в соответствии с тем, какой из новых центров оказался ближе.
3. Завершается, когда изменение расстояния внутри кластеров не происходит.

Конечно, у данного алгоритма есть один большой недостаток: из-за того, что начальное положение центроидов случайно, при каждом запуске будут образовываться разные кластеры. Помимо этого, минусом является необходимость в самостоятельном выборе количества кластеров.

Правильно оценить его можно тоже только эмпирически. Существует несколько популярных методов оценивания. Я использую самый простой – «локтевой» метод. Его суть заключается в том, чтобы запустить «К-средних» в цикле, при этом каждую итерацию увеличивать количество кластеров на 1 и записывать сумму квадратов расстояний внутри них. По завершению цикла нужно построить диаграмму и сопоставить полученные значения на каждой из итераций. На диаграмма нужно найти «локтевую» точку – после неё сумма квадратов расстояний перестает резко падать (потому что с этого момента кластеры начинают дробиться на более мелкие)

Вся описанная выше логика находится в файле project.py в github-репозитории - . Кроме того, все упомянутые ниже файлы могут быть найдены там же.

1. Испытание кластеризатора.

Представим, что у нас имеется реальная задача – нужно кластеризировать детские сказки с сайта detskie-skazki.com/russkie-narodnye-skazki/. Я создал новую программу example.py , в которой провёл парсинг сказок. Имея теперь названия и тексты документов, сформировал конечную матрицу, вызвав функцию generate\_matrix\_using\_pca. Далее провёл 15 раз кластеризацию полученных данных и выяснил с помощью «локтевого» метода, что наилучшее количество кластеров -

В результате испытаний получил набор кластеров, описанных в файле clusters.txt , проанализировав которые становятся ясно, что программа работает успешно.

4.Вывод.

Из испытания программы становится ясно, что она хорошо справляется со своей задачей, а значит, цель выполнена, и я смог сделать шаг к решению поставленной проблемы.