МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

Федеральное государственное бюджетное образовательное учреждение

высшего профессионального образования

«Московский Авиационный Институт»

РЕФЕРАТ

по дисциплине: «Фундаментальная информатика и вычислительная система»

на тему: «Семантический анализ: анализ тональностей»

Выполнил студент группы М80-107Б-18

Шичко Алексей Игоревич

Проверила преподаватель:

Аспирант кафедры 806

Ридли Александра Николаевна

Москва

2018

Содержание

[1. Введение 3](#__RefHeading___Toc247_2798732080)

[2. План работы 4](#__RefHeading___Toc301_2798732080)

[3. Выполнение работы 5](#__RefHeading___Toc303_2798732080)

[3.1 Получение данных из steam 5](#__RefHeading___Toc305_2798732080)

[3.1.1 Нахождение страниц с играми 5](#__RefHeading___Toc416_2798732080)

[3.1.2 Выуживание комментариев 6](#__RefHeading___Toc418_2798732080)

[3.1.3 Нахождения текста комментария и информации о пользователе 7](#__RefHeading___Toc420_2798732080)

[3.2 Обработка полученных данных 8](#__RefHeading___Toc422_2798732080)

[3.2.1 Разбиение комментариев на фичи, обработка артефактов. 8](#__RefHeading___Toc424_2798732080)

[3.2.2 Подсчёт различных значений фичей 9](#__RefHeading___Toc511_2447044075)

[3.3 Обучение модели, проверка 11](#__RefHeading___Toc513_2447044075)

[4. Заключение 11](#__RefHeading___Toc515_2447044075)

[4.1 Термины 12](#__RefHeading___Toc577_3991888621)

[5. Библиографический список 13](#__RefHeading___Toc579_3991888621)

# 1. Введение

Актуальность анализа текста на сегодняшний день обусловлена несколькими факторами:

1 - для многих исследований надо проанализировать большое количество данных, и делать это всё руками, без помощью автомата будет очень громоздким и трудоёмким процессом.

2 – работа с клиентами или поддержания определённого состояния общения в комьюнити требует большое количество человеческого факторами

3 – улучшение сервиса, автоматический сбор и анализ данных, получение фидбека даже если человек явно ничего не произносил

Как раз NLP совместно Machine\Deep Learning позволяют избавится от проблем с поиском большого числа консультантов, секретарей, диспетчеров, операторов и т.п.

Возьмём для чистоты следующий пример:

Интернет магазин подарочных вещей имеет неплохой зароботок, но руководитель проекта хочет получать ещё больше. Где можно достать дополнительный зароботок? У магазина сидят два – три оператора каждые сутки и помогают клиентам. Как вариант – заменить трёх операторов на одного программиста, который напишет софт для автоматического анализа запросов пользователей и поиска по сайту. Т.е. если пользователь пишет “Что подарить бабушке?” система может сделать поиск по тегам, комментариям, рейтингу и т.д. Для полноты системы программист может ввести систему отзывов для поддержания своей модели “в форме”.

Очевидно что компания явно выигрывает от снижения штата и автоматизации монотонного процесса.

Заметим, что это только единичный пример в конкретной отрасли, такого рода требования можно встретить повсюду, и каждая компания пытается найти свою выгоду.

# 2. План работы

Программу для обработки я пишу на интерпретируемом языке Python. Причины данного выбора: прост в освоении, достаточно библиотек подходящих к самым разным задачам.

Анализ данных буду проводить с помощью машинного обучение (Machine Learning – ML). Для того чтобы показать основы будут использоваться комментарии с игровой платформы Steam ( Steam площадка где можно разместить и купить игру, так же оставить отзывы об игре и поделиться впечатлениями )

Для построения исходных данных есть два подхода: 1 – достать много игр и комментарий к ним, затем передать на конвейер для модели и обрабатывать; как исход, много слов, много оценок и, можно сказать, готовый датасет. 2 – ограничить количество комментариев, затем провести аугментацию ( каждое слово можно разбить на морфемы, отнести к определённой лексической группе и разными способами добавить признаков к одному слову )

В данном исследовании я выбрал первый способ, т.к. он более прост в реализации неопытному программисту.

О том как получить комментарии из игр в большом количестве – я использовал библиотеку для python’a – scrapy. Данная библиотека упрощает переход с одной страницы на другую и получение из неё определённых классов и тэгов ( об этом позже ). После настройки скрапи я перехожу по популярным тегам игр → по каждому из тегов выбираю перехожу во все игры на всех страницах тега → в каждой игре обрабатываю комментарии

После того как работа с получением и обработкой данных законченна я создаю датасет из комментариев как разряженную матрицу и передаю её на “съедение” модели опорных векторов с линейным ядром. О модели и как всё это создаётся позже будет в соответству

# 3. Выполнение работы

## 3.1 Получение данных из steam

### 3.1.1 Нахождение страниц с играми

Стояла задача получить данные любого рода, чтобы поиграться с ними и попробовать классифицировать. Изначально планировалось обратиться к API твиттера, но мою заявку не подтвердили и никакого доступа к нему не было. Тут я решил что игры, как искусство, ничуть не уступает каким-то политическим событиям и сериалам и подумал это будет хорошей идеей.

Настраиваем скрапи на определённый таймаут, чтобы он не слишком часто делал запросы к сайту ( чтобы не получить перма бан по айпи за дудос ). Включаем кжширование сайтов, чтобы не было слишком частых запросов.

В скрапи весь интерфейс построен на так называемых “пауках”. В скрипте паука я прописываю страницу старта и правила переходов. В правилах переходов есть несколько опций:

1. Имя паука – для того чтобы его можно было вызвать из командной строки
2. Выбирать по какому селектору\* будем выбирать линк по которому перейдём
3. Указываем функцию, которая будет обрабатывать ответ сервера
4. Если надо, пишем регулярные выражения по которым наш паук будет переходить\не будет переходить
5. Задаём разрешённые сайты для переходов ( обойдёмся без редиректов )

Что бы всё это прошло успешно, нужно собственными ручками пощупать html содержимое страницы и посмотреть какие теги ведут меня к моей цели, и какого формата линки мне могут встретиться.

В данном случае мой стартовый url – <https://store.steampowered.com/games/>,

разрешённый домен – store.steampowered.com

В качестве link extractor’a используем LxmlLinkExtractor который нам поставляет скрапи ( это наиболее распространённая форма + она встроенна в скрапи и ненадо выдумывать велосипед ). Далее предстоит сделать выбор селектора по html тэгам: xpath или css селектор.

Кратко по ним:

1. xpath – более гибкий и предоставляет более широкий функционал по выбору различных тегов
2. css селектор – простой. структура тэг.класс#айди

Порывшись в html структуре стартовой страницы видим следующие признаки:

* **Популярные теги** имеют тэг div с классом content\_popular\_tags с линком внутри их
* Внутри страницы с конкретным тегом видим что **стрелочка с линком на следующую страницу** имеет тэг a с классом pagebtn
* На каждой из страниц с тэгами видим что **игры которые нам подходят** представлены в большой коробке с тэгом div и айди search\_result\_container

Видим что ничего сложного нет, поэтому используем css селектор для выбора подходящего линка. После обращения к каждой игре получаем html страницы с игрой (которых к слову 11.5к)

### 3.1.2 Выуживание комментариев

Казалось бы, зная что такое css селекторы и xpath легко получить любую информацию со страницы.

Я долго не понимал, почему я вижу комментарии на странице игры, а мой паук не видит. В чём подвох? Ответ: для того чтобы не нагружать пользовательский трафик, стим подгружает комменты только когда ты доскролишь до них, иначе их просто нет на странице. Делается это путём AJAX запроса, в ответ на него возвращается json объект, с двумя полями: success и html.

Узнав про такую систему ( всё чаще и чаще используется на самых разных страницах ), я полез узнавать где этот запрос можно проследить. В инспекторе на странице с игрой я нашёл нужный мне запрос, вот его пример: [https://store.steampowered.com/appreviews/%s?start\_offset=0&day\_range=30&start\_date=-1&end\_date=-1&date\_range\_type=all&filter=summary&language=russian&l=russian&review\_type=all&purchase\_type=all&review\_beta\_enabled=1](https://store.steampowered.com/appreviews/%25s?start_offset=0&day_range=30&start_date=-1&end_date=-1&date_range_type=all&filter=summary&language=russian&l=russian&review_type=all&purchase_type=all&review_beta_enabled=1) ( в реале он немного больше т.к. в нём указанно кол-во возвращаемых комментариев, но для симулирования изощрятся не надо ), в этой строчке %s символизирует id игры, к которой я хочу получить комменты.

Таким образом, переходя на страницу с игрой, я из возвращаемого объекта нахожу айди игры, и по этому айди конструирую следующий запрос уже к комментариям.

На выходе получаю json из которого я беру html значения и перевожу его в селектор для дальнейшего извлечения

### 3.1.3 Нахождения текста комментария и информации о пользователе

Повторяем цикл – смотрим структуру сайта → пытаемся провести аналогию у себя в коде.

В подгружаемых комментариях есть много блоков – полезные, последние, случайные и т.п., из которых мне нужны только первые. Эти “полезные” комментарии находятся в div’e с классом review\_box, но загвоздка в том, что ненужные мне комментарии находятся в теге с похожим классом. Тут на помощь приходит xpath селектор. В нём легко указать регулярное выражение, которому должен соответствовать класс, я составил следующее “\Areview\_box\s\*\Z” ( здесь \A означет начало строки, review\_box – текст который должен присутствовать в названии класса, \s\* - любое количество пробелов, \Z – конец строки ). По данному селектору нашёл все полезные обзоры на игру. Далее…

В каждом обзоре, с помощью регулярных выражений для выуживания чисел, с помощью внутренних функций питона для работы со строками, с помощью xpath и css селекторов и, в конце концов, с божьей помощью, я собираю всю нужную информацию о комментаторах и их отзывах.

Дабы не потерять всё что я тут напарсил, использую инструмент для сохранения данных в csv таблицу ( удобно потом в любой программе открыть и посмотреть ) одна легко устранимая проблема - \n в комментарии, его заменяю на {n}, переводя в одну строчку.

На выходе получаем таблицу csv с нужными данными, и больше не возвращаемся к парсингу steam’a, пусть отдыхает

( одна из причин по которой я начал сохранять данные в файл – слишком долгое время получение комментариев раз за разом )

P.S. На данный момент я дополнительно написал файл utils.py который позволяет быстро писать файлы как html так и txt формата. Причина – обычно это занимает много строк кода: проверка существуют ли родительские дирректории, проверка существует ли файл, выбор записи текста ( w ) или последовательности битов ( wb ). А так занимает одну строчку - ut.write\_html(self.dest + "comments.html", data['html'])

## 3.2 Обработка полученных данных

После того как протестировали нормальное сохранение полученных данных, создаём скрипт main.py который будет управлять всем выполнением программы, и стыковкой его модулей.

Запуская main.py с флагом -crawl мы заново проходимся по всем играм и создаём новую таблицу

### 3.2.1 Разбиение комментариев на фичи, обработка артефактов.

Уже зная как полностью реализуется обработка буду всё рассказывать вариант, который существует на данный момент.

Пока считываем готовые комментарии из файла, сразу создаём объекты класса Comment, для более организованного доступа к ним.

Для начала подсчитываем общее кол-во положительных и отрицательных комментариев, это нам понадобиться для ускорения подсчёта в будущем ( любые запаршенные данные сохраняем в отдельный файл для доступа к ним из скрипта + можно взглянуть на данные ).

Далее по каждому из комментов проходимся, создавая вектор. Попутно удаляем ненужные слова, очищаем артефакты, подсчитываем общее количество слов. Как это проходит?

**Векторы.** В данном случае я решил взять униграммы и биграммы из слов, т.к. по общему мнению они показывают коэффицент результат/вложенные силы. Униграмма – слово, биграмма – словосочетание.

**Очищение от артефактов.** Приведу пример для ясности: если какой-то пользователь напишет комментарий “Хорошая игра.Всем советую” функция определит одним из признаков строку “игра.всем”. Понятно что слова через точку должны разделяться, если стоит запятая в конце слова, она должна убираться ( “приехал,” и “приехал” будут разными фичами). Также надо некоторые люди пишут английскими, китайскими ( вообще самые разные встречаются ) ставят много смайликов, иногда сарказм иногда невместно. Всё это непомерно увеличивает результирующее количество уникальных униграмм и биграмм. Попытаясь написать небольшую функцию для обработки всех этих случаев у меня ничего не получилось, пришлось делать отдельную функцию для этого.

Вызвать её можно путём ut.clear\_some\_sht(feature), возвращает строку. Если на выходе получается массив строк, метод возвращает одну строку, где фичи представлены через ||||. (Функция что-то взяла от алгоритма маркова и пузырькового метода сортировки, затем с помощью регулярных выражений отслеживаются различные случаи встречания разделителей. С помощью кода каждой из буквы отслеживаются ненужные фичи).

После разделения на фичи, **подсчитываем количество** каждой из них и добавляем в общий счёт всех слов ( сколько раз встречается в положительных комментариях, в отрицательных, сколько положительных\отрицательных комментариев содержит это слово ( для дальнейших вычислений ) )

### 3.2.2 Подсчёт различных значений фичей

На данном этапе комментарии содержат много не нужных предлогов, междометий, союзов, которые не несут смысловой нагрузки. Чтобы избавиться от них, в каждом комментарии для каждой фичи **считаем tf-idf** (Term Frequency – Inverse Data Frequency, временная частота – обратная частота данных), если это значение ниже определённой планки, убираем это слово из списка фичей.

Теперь удалять уже нечего, создаём массив Target Names – он будет указателем на какую позицию вставлять значение каждой фичи в общем случае. Создаю его путём инициализации set’a ( множества ), а после всех дополнений к нему сортирую его ( позже объясню зачем ) и перевожу в список.

Прогоняя код на данном этапе, можно заметить что список уникальных фичей имеет размер 2 миллиона, а искать индекс каждого слова в этом списке занимает безумно много времени. Так как список отсортирован можем найти приблизительные индексы для каждой фичи. Для этого создаём индекс начала вхождения первой буквы в таргет неймс и конец. Допустим после того как я всё определил программа будет знать, что фичи на “г” имеют индексы с 61019 номера по 71700.

Последний этап в обработке конкретно фичей – подсчитываем значение delta tf-idf. Отвечает за то, какую окраску фича имеет среди документов. Формула у него следующая cnt[feature] – сколько раз фича встречается в этом комменте, |N|, |P| - количество негативных и положительных комментов ( зафиксировали эти числа в самом начале, пока обрабатывали все комментарии скопом ) P\_f, N\_f – сколько раз эта фича встречается в положительных и отрицательных комментариях соответственно ( тоже подсчитывали пока создавали вектора признаков ).

Заполняем матрицу. Создаём разряженную матрицу с помощью библиотеки scipy, где есть функция sparse.lil\_matrix использует немного другое представления для матриц, что позволяет бомбой создавать гораздо бОльшие матрицы чем обычно.

Проходимся по каждому комментарию ( индекс комментария будет i – строка в матрице ), по каждой фиче. Индекс фичи ищем с помощью подсчитанных раньше точек старта и конца ( индекс фичи – j ). Придаём значение матрице в точке data[i,j] значение delta tf-idf соответствующей фичи в конкретном комменте.

На выходе получается готовая, заполненная матрица.

P.S. На данном моменте была большая проблема отслеживания прогресса обработки данных, т.к. при быстрых операциях система тратила много ресурсов на вывод. Реализовал класс Watcher в файле utils.py, цель в том, что с помощью псевдо-псевдо-графики класс отображает этап выполнения обработки списка комментариев.

P.P.S. Также реализованно подкачка уже известных данных. Например: если я хочу подсчитать заново значение delta tf-idf, но у меня уже существует файл delta\_frac.json, программа будет погружать значения из него используя более простую формулу self.cnt[feature]\*Comments.delta\_frac[feature]

P.P.S. Написан небольшой файл settings.py откуда берутся разные ключевые параметры, чтобы менять их не ищя по всему коду ( например нижний порог tf-idf )

## 3.3 Обучение модели, проверка

Данной задачи используем классификатор основанный на методе опорных векторов ( SVM ). Суть в том, что если данные имеют размерность n, то алгоритм увеличивает размерность на 1 и гиперплоскостью разделяет на два класса. ( можно взять в пример когда шарики расположены на столе, их подбрасывают в воздух и в моменте где они находятся на разной высоте разделяют их плоскостью ).

Оценку точности модели проводим с помощью кросс валидации, указывая количество проверок. Принцип работы: данные разбиваются на несколько блоков ( допустим 5 ), 4 блока используются для обучения, на 5 тестируется модель, и так 5 раз. В итоге берём среднее между значениями.

После того как модель обучена, сохраняем её как битовую последовательность чтобы было удобнее пользоваться в дальнейшем.

P.S. Последовательно дополняя различные функции к программе, добавляем разные ключи в main:

* crawl – парсим стим
* process – обрабатываем данные создаём матрицу
* bm ( build model ) - обучаем модель

# 4. Заключение

Обработка неструктурированных данных, вводимыми пользователями разных возрастов, национальностей, соц положения – очень трудоёмкий процесс.

В ходе написания данной программы я узнал про много новых инструментов и попрактиковался в работе с ними. Регулярные выражения стали мне молотком, css\xpath селекторы стали мне отвёрткой. json и csv помогал сохранить данные в ходе работы и посмотреть что они из себя представляют.

При написании первой версии она представляла из себя хиленькую программу. Когда данных было мало я не понимал что даже милисекунды в работе программы могут обернуться большой потерей при переходе на более крупные масштабы.

Задача могла упроститься в несколько раз, но твиттер отказался предоставлять мне своё айпи.

Это наиболее полезный опыт при написании программ. Я остро почувствовал, что не свободно ориентируюсь в питоне, можно было лучше. Каждую десятую строчку приходилось гуглить, т.к. до этого не занимался подобными вещами. Приучился использовать интерпретатор в коммандной оболочке чтобы проверять некоторые конкретные функции и как они ведут себя на заданных данных.

При обрабатывании текста было много проблем с нормализацией данных. Можно улучшить модель путём устранения грамматических ошибок, использования тонального словаря, аугментации, лемматизации, стемметизацией, комбинации слов с тегами (напр. я-метоимение), добавления дополнительных морфологических признаков, использование 2,3,4 буквенных грамм. Для ускорения вычислений можно использовать openCL или CUDA ( распределение вычислений на видеокарту ).

Не зная с чего начать и что из себя представляет анализ текста, я освоился в этой области, понял недостатки своего кода и теперь понятно в каком направлении можно двигаться дальше.

## 4.1 Термины

стим – ( steam ) площадка для торговли играми

питон – интерпретируемый язык программирования python

скрапи – библиотека для питона scrapy

таймаут – ( timeout )какое-то время когда программа бездействует

бан, забанить – ( ban ) ограничение доступа к ресурсу

айпи – ( IP ) сетевой адресс узла

дудос – ( Ddos ) хакерская атака с целью довести систему до отказа обычным пользователям

линк - ( link ) ссылка на странцу

айди - ( id ) уникальный идентификатор

коммент - ( comment ) комментарий

фича - ( feature ) признак

# 5. Библиографический список

Помогло вникнуть в механику анализа текста:

* <https://habr.com/post/263171/>
* <https://habr.com/post/149605/>

<http://nlpx.net/archives/57> – tf-idf и связанное с ним

<http://www.long-short.pro/post/kross-validatsiya-cross-validation-304> – как работает кросс валидация

<https://toster.ru/q/21037> – полезные советы по улучшению модели

<http://datascientist.one/support-vector-machines/> - как работает метод опорных векторов

<https://machinelearningmastery.com/save-load-machine-learning-models-python-scikit-learn/> - как сохранять модель

[https://stackoverflow.com](https://stackoverflow.com/)/ - все остальные вопросы