Министерство образования и науки Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего образования
«КАЗАНСКИЙ (ПРИВОЛЖСКИЙ) ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

ИНСТИТУТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

КАФЕДРА СИСТЕМНОГО АНАЛИЗА И ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

Направление: 02.04.02 — Фундаментальная информатика и информационные технологии
Профиль: Математические основы и программное обеспечение информационной безопасности и защиты информации

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

Разработка метода получения векторного представления слов, отсутствующих в обучающей выборке

Студент 2 курса
группы 09-635
"___"_____ 20__ г. (Боробов П.А.)

Научный руководитель
к.ф.-м.н., ассистент КСАИТ
"___"_____ 20__ г. (Разинков Е.В.)

Заведующий кафедрой
д.т.н., профессор
"___"_____ 20__ г. (Латыпов Р.Х.)

Казань – 2018

Оглавление

Введение

Область обработки естественных языков (англ. Natural Language Proccessing, далее NLP) активно развивается несколько десятков лет. Задачи решаемые этой областью имеют широкое применение в реальном мире, например, после появления различных интернет-сервисов, которые предоставляют возможность динамически создавать или менять контент интернет-страниц, текстовой информации стало очень много и ее ручная обработка стала занимать невероятное количество времени. NLP позволяет автоматизировать обработку текста с целью выявления некоторой структурированной информации, которая может быть использована в каких-либо бизнес-целях. В данной работе была рассмотрена задача определения тональности текста. Эта задача не нова и актуальна по сей день. Об этом говорят проводимые каждый год конференции (Sentiment Analysis Symposium [19], SemEval [18]). Что же такого интересного в этой задаче, почему она привлекает такой интерес, и почему решение этой задачи является важным для области анализа текста на естественном языке? Для начала стоит понять в чем заключается эта задача и уже потом посмотреть на ее применение. Смысл задачи заключается в том, чтобы по одной фразе, предложению или абзацу, предсказать содержащуюся в них эмоцию, сказать какой характер носит текст: позитивный или негативный, или усложняя эту задачу производить оценку по шкале (например, от 1 до 5 звезд). Получается, данный метод может быть применен по отношению к любому тексту. Существует много интернет-сервисов, где данный метод применяется для оценки пользовательского мнения о товарах. Хорошим примером является Google Product Search, Bing Shopping [12]. В этих сервисах совместно с определением эмоционального окраса текста используются техники извлечения атрибутов объекта и определяется мнение не всего объекта, а его атрибутах или характеристиках. Понятное дело, что задача не была бы такой актуальной, если бы ее применение было возможно только в интернет-сервисах. Так в Gollup Poll [12] исследовали корреляцию эмоционального окраса сообщений в твиттере и показате-

лем доверия потребителя, в результате было выявлено, что изменение некоторых экономических параметров может влиять на мнение людей в социальных сетях. Ключевым применением, конечно, является предсказание трендов на рынках биржевых товаров [12], которое дало бы преимущество в игре на таком рынке. Данную задачу можно решать как методами основанными на словарях, так и с использованием методов машинного обучения. В работе будут рассмотрены алгоритмы, основанные на машинном обучении. Задача может быть сформулирована, как задача двухклассовой или многоклассовой классификации. Для ее постановки необходимы некоторые текстовые признаки и предсказываемая метка этого текста, на их основе может быть построена выборка, которая будет использована при обучении моделей. Модели описанные в этой работе используют некоторое машинное представление текста, такой способ представления текста подразумевает использование чисел или вектора из чисел при представлении конкретного слова. Данный вектор может быть представлен различными способами. Например, наиболее интуитивный способ, представить слово в виде вектора состоящего из нулей и единицы в той позиции, где это слово расположено в словаре. В данной работе были рассмотрены различные способы представления слов. Семантическое векторное представление слов - это отдельная область в NLP. Она занимается разработкой алгоритмов, получающих векторные представления слов способных распозновать семантику, например, $sim(v(good), v(bad)) \approx 1$. Подробнее о свойствах такого представления слов будет описано в разделе 1.2. Важно отметить, что решаемая задача, позволяет представлять вектор более компактно и, как показывают результаты из статей [4] [10] [15] [5] позволяют улучшать результат решения целевых задач, например, задачи определения тональности текстов. В области векторного представления остается много нераскрытых вопросов. Одним из таких вопросов является векторное представление слов, отсутствующих в словаре. Решаемая в этой диссертации задача только недавно получила широкое распространение, но уже сейчас можно наблюдать непреодолимый интерес исследователей в области обработки естественного языка. Об

этом могут говорить последние конференции ориентированные на эту область, такие как: EMNLP, CoNLL, ACL и т.п. В этих конференциях всегда специальным образом выделялись исследования связанные с возможностью получения векторных представлений по тем словам, которые могли не участвовать в процессе обучения. Данная проблема может быть поставлена как расширение основной задачи по векторному представлению слов, поэтому многие исследователи, занимающиеся этим направлением, являются также авторами статей, решающих и эту проблему. На текущий момент существует несколько работ решающих эту задачу. Одной из таких работ является [5], где авторы статьи моделируют векторное представление слова, как усредненную поэлементную сумму составных частей слова, представленных в векторной форме. Также существуют и более тривиальные подходы для решения этой задачи, например, предложенная в статье модель [16], где проверяется гипотеза, что при помощи информации о символах можно восстановить векторное представление слова по уже составленному словарю. Оба подхода будут рассмотрены в данной работе и будут приведены результаты валидации по каждому алгоритму. В данной работе были использованы различные подходы для решения этой задачи. Например, описанный в разделе 2.2.1 алгоритм, восстанавливает векторные представление лучше, чем тот алгоритм, на котором были основаны разработки описанные в главе 2, но при этом такой результат все равно сильно уступал текущим state-of-the-art работам в области векторного представления слов. Также в этой работе возникли проблемы по оценке качества полученных векторных представлений подробнее об этом будет рассказано в разделе 5.2.1. Был разработан и реализован еще один метод решающих данную задачу, который был основан на двух работах [5] [15]. Такой алгоритм получил state-of-the-art результат на некоторых валидационных выборках и был получен state-of-the-art результат при решении задачи языкового моделирования английского языка.

1. Литература

1.1. Определение тональности текста

Современные алгоритмы определения эмоционального окраса текста содержат в себе различные подзадачи из разных областей. Сюда входят семантическое векторное представление слов, определение частей речи и логическое представление текста.

1.1.1. Лингвистическая структура предложения

Важно понимать, что любое предложение на естественном языке представляет собой некоторую структуру, которая может быть по разному интерпретирована. Существует много работ в данной области, которые пытаются организовать последовательности слов в такие структуры. Логическое представление – это такое представление, при котором выражается некоторая логическая взаимосвязь слов, определяется их порядок по смыслу текста. Это некоторое представление текста в виде дерева, где выражается нисходящая взаимосвязь слов. Ключевым в этом представлении является ее структура в виде дерева, которая может быть использована в рекурсивных моделях. Такая форма представления текста позволяет подсказать алгоритмам информацию о структуре предложения и о его смысле, что непременно дает улучшение результата в решаемых задачах. Так, например, была улучшена точность алгоритма, использующего рекурентные нейронные сети [17], где впервые была использована структура предложения в виде дерева и архитектура сети, и где слова были представленны векторами. В этом разделе будут описаны state-of-the-art работы последних лет [10][17]. Также в этих алгоритмах были раскрыты варианты использования различных представлений текста, которые позволяют использовать последние наработки в области глубинного обучения. Описанные в секциях (1.1.2, 1.1.3, 1.1.4) алгоритмы решают эту задачу с совершенно различными моделями, каждая из которых интересна излагаемой

идеей, каждый из представленных алгоритмов раскрывает некоторый аспект задачи и ее подводные камни, эти модели интересны еще и тем, что здесь происходит качественный переход в области NLP от настраиваемых вручную признаков в сторону автоматически определяемых моделью, то есть являющимися обучаемыми признаками.

1.1.2. Рекурсивные нейронные модели

Рекурсивные нейронные модели (Recursive Neural Models) – модели, использующие векторное представление фраз, которые потом используются для классификации. [17]

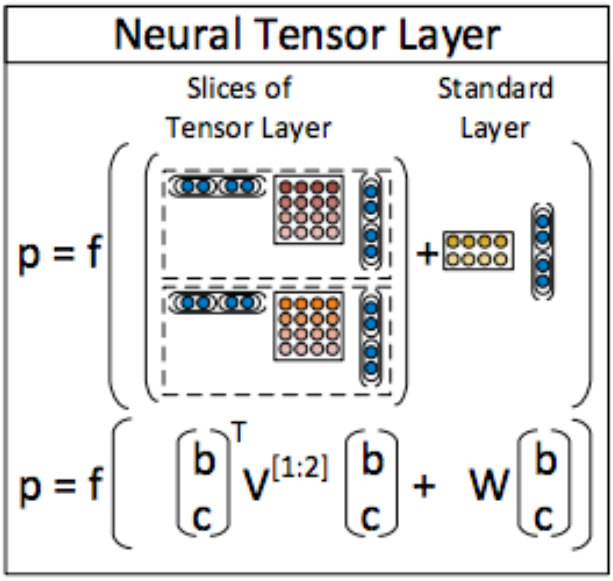

Рис. 1: Представление одного слоя в RNTN [17]

Входящий текст представляется в логической форме в виде дерева, где каждый узел является композицией двух дочерних узлов. Слова представлены векторами размерности $d = 300$, здесь же определена функция $g(w_1, w_2) = p_k$, где $|w_i| = d,\ i = 1, 2$ и $|p_k| = d$. На вход модели дается некоторое заранее заданное представление слов во всем словаре, которая обучается совместно с параметрами модели. Функция активации по вектору в конечном узле по заданой метке класса определена

следующим образом:

$$y^a = softmax(W_s a), \tag{1}$$

где $W_s \in \mathbb{R}^{c \times d}$ матрица весов, a - метка класса. Основные отличия таких моделей лежат в вычислении скрытых векторов p_k или, другими словами, в определении функции g [14] [17] [6]. Используемая в статье [17] модель, достигла наилучшего результата на момент выпуска статьи на выборках SST-1, SST-2, MR. Результаты представлены в таблице 1.

MR	SST1	SST2
80.5	45.7	85.4

Таблица 1: RNN результаты на различных выборках

Минусом используемой модели является некоторый предварительный этап преобразования сырого текста в его логическую форму. Это накладывает дополнительные ограничения на использование данного алгоритма в прикладных задачах.

1.1.3. Сверточные нейронные сети в классификации предложений

Одна из последних моделей в значительной мере отличающихся от рекурсивных моделей. Здесь заложена идея векторного представления слов и матричного представления документов. Между документом и графическим изображением проводится некоторая параллель, где указывается, что между отдельными элементами этой матрицы могут быть связи, то есть фразы в начале и конце предложения связаны между собой не через цепочку слов, а напрямую, чего нельзя получить в древовидном представлении текста. Подобными идеями задавались при анализе изображений и реализации модели классификации изображений. В работе [8] использовались сверточные нейронные сети (Convolutional Neural Network, далее CNN) для решения этой задачи.

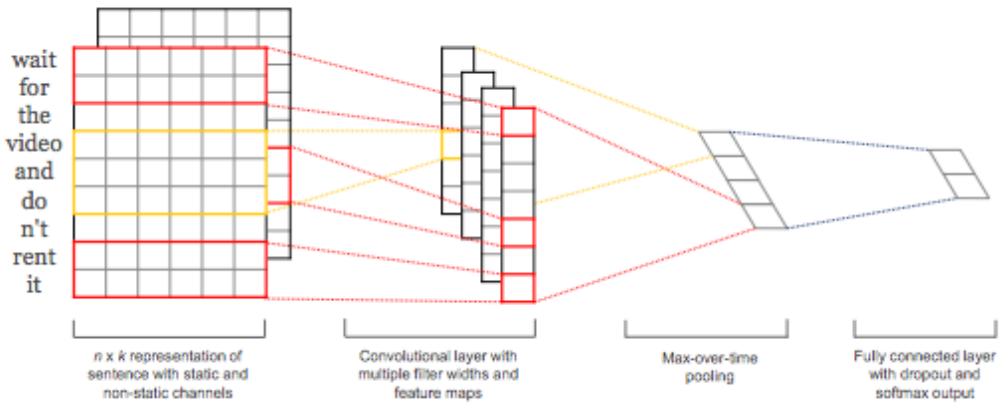

Рис. 2: Архитектура нейронной сети [10]

Все дальнейшее описание модели взято из статьи [10] Пусть у нас есть некоторое векторное представление слов $x_i \in \mathbb{R}^d$, $i = 1..s$, а s – количество слов в предложении. Представим документ в виде вектора, состоящего из слов, в векторной форме фиксированной размерности, в том порядке, в котором слова находились в исходном предложении. К полученной матрице применим операцию свертки с нелинейной функцией активации (ReLU). Размерность фильтра будет определяться размерностью вектора d, а высота вектора h будет варьируемой и является гиперпараметром нейронной сети.

$$X_{1:s} = \begin{pmatrix} x_1^T \\ x_2^T \\ \vdots \\ x_s^T \end{pmatrix} \tag{2}$$

Пусть (2) матрица, где x_i – векторное представление слова на i-ой, тогда функция свертки, для фильтра размерностью $w \in \mathbb{R}^{hk}$, где k определяет количество таких фильтров, определена следующим образом:

$$o_i = f(w \cdot x_{i:i+h-1} + b), \tag{3}$$

где $b \in \mathbb{R}$ – смещение, а f – нелинейная функция (ReLU). После этого

получается некоторая карта признаков по всем регионам:

$$c = [c_1, c_2, ..., c_{s-h+1}]. \tag{4}$$

После получения карты признаков c выбирается максимальный признак $\bar{c} = max(c)$, такое поведение обсуловлено тем, что длина документов может быть разной, и количество регионов в зависимости от длины документов может изменяться, но архитектура нейронной сети не должна меняться от этого. После проведения этих действий следует полносвязный слой с регуляризацией dropout, который на момент обучения выбрасывает часть переменных. Выход полносвязного слоя подается на вход функции $Softmax$. $Softmax$ – функция принимающая на вход n-мерный вектор и с n-мерным выходом, где сумма компонент равна 1 и каждая компонента неотрицательна. Авторы приводят несколько вариантов такой модели, в которой меняется количество входных каналов, меняется способ их инициализации: предобученные векторы статичны, пред обученные векторы нестатичны, случайные векторы. В зависимости от такой инициализации результаты значительно отличаются. Такая модель показала высокий результат (таблица 2).

Модель	SST-1	SST-2	MR
CNN-static	42.2	83.5	80.5
CNN-random	44.8	85.6	75.9
CNN-non-static	46.7	87.0	81.3
CNN-multichannel	44.6	87.1	80.8

Таблица 2: Результаты разных моделей CNN

Здесь приведена архитектура нейронной сети без детального описания всех тонкостей и всех гиперпараметров модели. В статье [22] проводится валидация данной модели. Производится поиск и валидация лучших значений гиперпараметров модели, при которых достигается высокая точность классификации.

1.1.4. LSTM в виде дерева

В статье [20] приводится архитектура рекурсивных нейронных сетей с памятью (RNN-LSTM) решающая задачу определения эмоционального окраса предложения. Данный вид нейронной сети является развитием идеи из алгоритма описанного в разделе 1.1.2. В данном алгоритме также анализируются структура предложения в виде дерева, но ключевым отличием является тот факт, что в такой сети закономерности запоминаются на всей последовательности слов и не отображены только в каких-то конкретных узлах. Одним из преимуществ этой работы по сравнению с работой [17] является количество обучаемых параметров, авторами статьи удалось сократить количество параметров и улучшить результат.

Данная работа не представляет из себя типовое использование LSTM-сетей. Здесь произведена модификация LSTM-ячеек. Обычно у такой ячейки с индексом j имеется входящий и выходящий фильтр i_j, o_j, запоминающий фильтр c_j и скрытое состояние h_j. Отличие Tree-LSTM от обычного LSTM в том, что у Tree-LSTM обновление фильтров зависит от всех дочерних узлов этой ячейки. Также данный тип ячейки содержит забывающие фильтры для каждого дочернего узла.

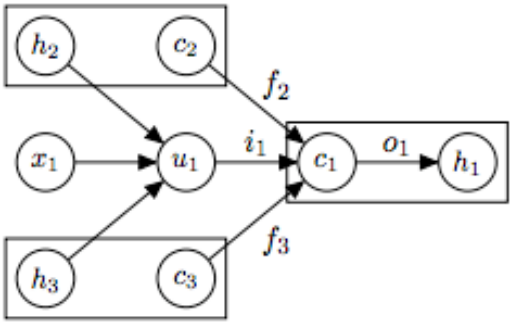

Рис. 3: Tree-LSTM с двумя дочерними узлами [20]

На вход алгоритму подаются заранее подготовленная выборка, состоящая из предложений с представлением в виде дерева. Сначала каждое дочернее слово подается на вход дочерним узлам Tree-LSTM, дальше их скрытые состояния h_j, подается на вход родительскому узлу вме-

сте с родительским словом в структуре предложения. Таким образом производятся вычисления по всем ячейкам этой сети. Количество ячеек является гиперпараметром модели и требует валидации при обучении алгоритма. Данная работа показала на момент написания статьи state-of-the-art. Результаты обучения на различных выборках представлены в таблице 3

Модель	SST-1	SST-2	MR
Constituency Tree-LSTM, tuned	51.0	88.0	81.7
Constituency Tree-LSTM, fixed	49.7	87.5	-

Таблица 3: Результаты разных моделей Tree-LSTM [20]

1.2. Семантическое векторное представление

1.2.1. NLP и задача семантического векторного представления слов

Обычная форма векторного представления слов представляет собой one-hot вектор его размерность определена размером словаря и он состоит полностью из нулей кроме одной единственной позиции, при таком представлении можно точно определить слово и последовательность слов, представляя таким образом документ в виде матрицы, можно заметить, что такое представление сильно избыточно и не является вектором в математическом смысле. То есть математические операции над такими векторами не представляют из себя никакой семантики. Проблему векторного представления слов, отображающих семантику, решают множества алгоритмов: word2vec [4], GloVe [15], fastText [5] и т.д. Это некоторое векторное представление слова, размерность которого может быть задана при обучении модели. Такое представление более компактно и отвечает свойству семантической схожести, то есть два близких по смыслу слова в векторной форме обладают большим скалярным произведением. Применение математических операций в действительности может дать какой-то осмысленный результат, например,

$v(King) - v(Man) + v(Woman)$, где $v(w)$ - функция преобразования слова в векторное представление, результат этого выражения будет похож, согласно определенной метрике, на слово $v(Queen)$. Такое векторное представление в области обработки естественного языка (англ. Natural Language Processing) достаточно сильно распространено и показывает значительное улучшение результатов в решении не только задачи классификации текста, но и во многих других.

1.2.2. Описание задачи для слов отсутствующих в обучающей выборке

В данной работе для разработки алгоритма необходимо поставить задачу и определить требования по валидации получаемых векторных представлений слов. Задачу можно сформулировать следующим образом: для любого произвольного текста, необходимо получить векторные представления слов в этом тексте, даже для тех слов, которые отсутствовали при обучении. Полученные векторные представления слов, должны соответствовать определенным характеристикам схожести. Также полученные векторные представления должны улучшать результат какой-либо целевой задачи, использующие векторное представление слов, в зависимости от выбранного способа инициализации embedding слоев в искусственных нейронных сетях и с расширяемым словарем. Можно сказать, что для оценки качества получаемых векторных представлений должна проводиться валидация на выборках, оценивающих схожесть двух векторных представлений слов, и валидацию на целевой задаче. Способы валидации схожести и целевая задача подробнее описана в главе 3.

1.2.3. Модель Mimick

Авторы статьи [16] предлагают решать проблему отсутствующих векторов, как проблему генерации вектора, независимо от того, как они были получены. Они предполагают, что все вектора для слов могут быть вычислены согласно некоторому алгоритму. Обучая модель

на существующем словаре таких векторов, они предполагают, что можно получить вектор для любого произвольного слова по его символам. Для некоторого языка L, словаря $V \subseteq L$ и пред обученных векторов $W \in \mathbb{R}^{|V| \times d}$, где d - размерность векторов и каждый вектор $\{w_i\}_{i=1}^{|V|}$ размечен вектором e_i. Функция $f : L \to \mathbb{R}^{|V| \times d}$, приближающая значение функции к существующему вектору: $f(w_i) \approx e_i$. Таким образом появляется возможность получать вектора для слов $w_k^* \in L \setminus V$ следующим образом: $f(w_k^*) = e_k^*$. В статье предполагается использовать нейронную сеть глубокого обучения, как функцию f. Архитектура нейронной сети представляет из себя двунаправленную LSTM-сеть (рис. 4).

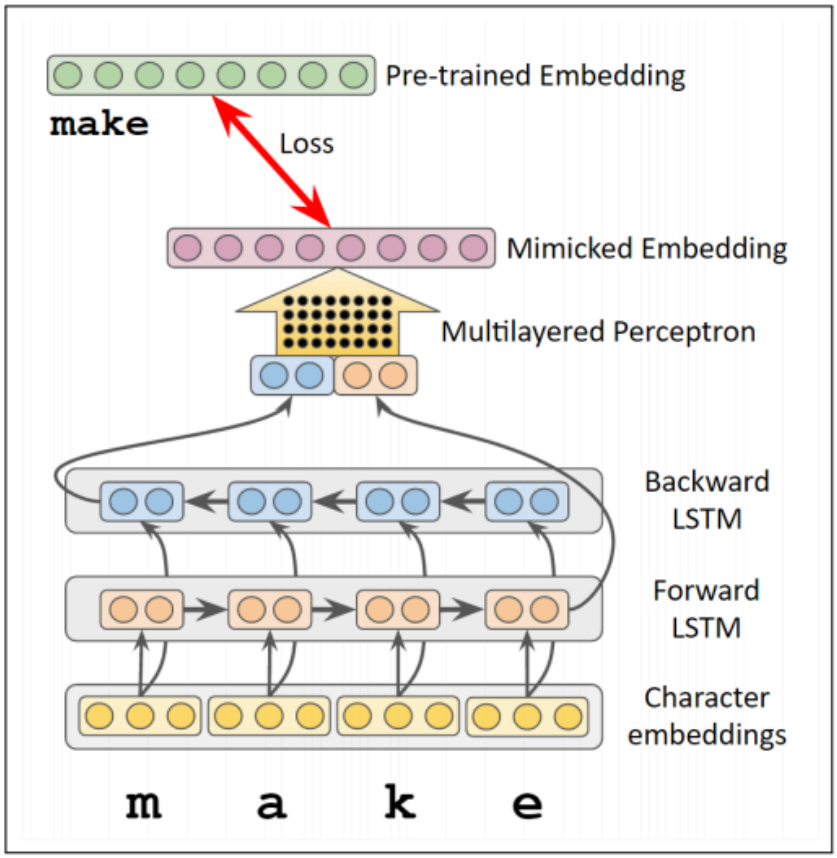

Рис. 4: Архитектура модели [16]

Где входом будет являются символы, генерируемого слова. Последовательность символов $w = \{c_i\}_1^n$ подаются на вход к слоям BiLSTM (forward, backward), как векторные представления символов $\{e_i^{(c)}\}_1^n$. Скры-

тые состояния для направленных вперед и назад слоев h_f^n и h_b^0 заключительные скрытые состояния соединяются и векторное представление слова будет представлено следующей формулой:

$$f(w) = O_T \cdot g(T_h \cdot (h_f^n \oplus h_b^0) + b_h) + b_T, \tag{5}$$

где T_h, b_h, O_T, b_T - параметры полносвязного слоя, а функция g - нелинейная функция активации, $\oplus$ - конкатенация двух векторов. Функция ошибки представляет из себя средне-квадратичную ошибку:

$$L = \| \ f(w_k) - e_k \ \|^2$$

1.2.4. Модель Enriching

В данном разделе будет описан метод представленный в статье [5]. Метод описанный в статье предлагает решение проблемы векторного представления слов, отсутствующих в обучающей выборке. Векторные представления слов, получаемые этим алгоритмом, показывают state-of-the-art результат на валидационных задачах, таких как: схожесть слов, схожесть редких слов (англ. WordSim, RareWords). Основная идея метода - получать векторное представления слова, как линейную комбинацию векторных представлений символьных n-грамм. Таким образом пытаются выяснить, как отдельные части слова влияют на общее представление вектора, чтобы в последствии для слов, отсутствующих в выборке, была возможность получить его векторное представление, которое могло отвечать некоторым семантическим свойствам, как, например, слова получаемые алгоритмами wor2dvec, GloVe и т.п. Основой этого алгоритма стала модель Skip-gramm предложенная в статье [4]. В этой статье было проверено предположение о распределении слов в зависимости, от его контекстных слов [9]. Целевая функция имеет вид:

$$\sum_{t=1}^{T} \sum_{c \in C_t} p(w_c | w_t), \tag{6}$$

где $p(w_c|w_t)$ - вероятность появления слова t в контексте c слова. В случае алгоритма word2vec используется функция softmax:

$$p(w_c|w_t) = \frac{e^{s(w_t,w_c)}}{\sum_{j=1}^{|W|} e^{s(w_t,w_j)}}. \tag{7}$$

Из-за такого способа представления функции вероятности, получается, что существует только одно предсказываемое контекстное слово. Поэтому решают несколько независимых задач, на каждое контекстное слово. Используя технику negative sampling, в нашу целевую функцию добавляются еще слова, которые не принадлежат контексту. В конечном виде целевая функция выглядит следующим образом:

$$\sum_{t=1}^{T} [\sum_{c \in C_t} \log(1 + e^{-s(w_t,w_c)}) + \sum_{n \in N_{t,c}} \log(1 + e^{s(w_t,n)})]. \tag{8}$$

Такая функция параметризирована относительно оценивающей функцией s, где в статье [4] было скалярное произведение: $s(w, v) = w^T v$. В описываемом методе использовалась другая оценивающая функция $s(\cdot, \cdot)$. С целью получить информацию о внутренней структуре слова, была предложена функция оценки на основе мешка с символьными n-граммами. Например, слово "какой" при $n = 3$ было бы представлено как: <ка, как, ако, кой, ой>. Символы $<$, $>$ были добавлены с целью обозначить начало и конец слова. Допустим имеется словарь n-грамм размера G. Для некоторого слова w можно определить множество $\zeta_w \subset \{1, .., G\}$ - множество символьных n-грамм для слова w. Пусть z_g - векторные представления для всех n-грамм размера g. Таким образом наше слово теперь представимо в виде множества таких n-грамм:

$$s(w_t, w_c) = \sum_{g \in \zeta_{w_t}} z_g^T w_c. \tag{9}$$

Авторы также приводят решение ряда проблем с требованиям к памяти, но к основной модели они относятся несущественно и носят исключительно оптимизационный характер при реализации алгоритма. Таким образом целевая функция статьи [5], является модифицирован-

ной в форме оценочной функции формулы (8) с оценочной функцией из формулы (19).

1.2.5. Модель Glove

В данном разделе будет описан алгоритм метод получения векторного представления слов, для фиксированного словаря из статьи [15]. Авторы статьи предлагают более детально рассмотреть каким именно образом используется статистика совместной встречаемости в современных алгоритмах, решающих эту задачу. Авторы предлагают новый метод для генерации векторного представления на основе статистики совместной встречаемости собранной со всей обучающей выборки. Дальнейшее описание алгоритма сохранило нотацию авторов статьи [15]. Пусть матрица совместной встречаемости пар слов будет обозначена X, X_{ij} - количество раз слово j встретилось в контексте слова i. Пусть $X_i = \sum_k X_{ik}$ - количество слов, которые были в контексте слова i. Теперь необходимо обозначить вероятность появления слова j в контексте слова i: $P_{ij} = P(j|i) = \frac{X_{ij}}{X_i}$. Для иллюстрации моделируемой вероятности, авторы статьи приводят пример с термодинамическими фазами. Пусть $i = ice$, $j = steam$, $k = solid$ можно ожидать, что отношение $\frac{P_{ik}}{P_{jk}}$ будет высоким. И таким же образом, если слово k будет иметь большее отношение к слову j, то отношение будет меньше. На основе текущего предположения авторы статьи предлагают использовать вероятность основанную на статистике совместной встречаемости. Авторы предлагают обобщенную модель:

$$F(w_i, w_j, \widetilde{w_k}) = \frac{P_{ik}}{P_{jk}}, \tag{10}$$

где $w \in \mathbb{R}^d$ - векторное представление слова i, а $\widetilde{w} \in \mathbb{R}^d$ - векторное представление контекстного слова. При текущем определении функции $F(\cdot, \cdot, \cdot)$ невозможно определить какую либо конкретную функцию, которая решала бы тождество 10. Для решения необходимо ввести ограничения эту функцию. Для этого необходимо ввести ограничение на $\frac{P_{ik}}{P_{jk}}$. Так как векторное пространство линейно, то векторная разница -

это более интуитивный способ отобразить это свойство свойство.

$$F(w_i - w_j, \widetilde{w_k}) = \frac{P_{ik}}{P_{jk}}. \tag{11}$$

Так как F может быть абсолютно всем чем угодно, например, искусственной нейронной сетью, то при такой формулировке можно потерять линейную структуру. Чтобы предотвратить эту проблему, необходимо взять скалярное произведение от аргументов функции:

$$F((w_i - w_j)^T \widetilde{w_k}) = \frac{P_{ik}}{P_{jk}}, \tag{12}$$

что предотвращает смешивание элементов вектора в произвольной модели. Чтобы отобразить симметрию относительно слова и контекстного слова авторы определяют следующее свойство:

$$F((w_i - w_j)^T \widetilde{w_k}) = \frac{F(w_i^T \widetilde{w_k})}{F(w_j^T \widetilde{w_k})}, \tag{13}$$

где $F(w_i^T \widetilde{w_k}) = P_{ik} = \frac{X_{ik}}{X_i}$. Таким образом можно выявить, что $F = exp$. Тогда верно и следующее:

$$w_i^T \widetilde{w_k} = \log P_{ik} = \log X_{ik} - \log X_i. \tag{14}$$

Так как $\log X_i$ не зависит от слова k можно заменить его на дополнительное смещение b_i, $\widetilde{b_k}$. Основной проблемой такого подхода является тот факт, что при прямом использовании статистика совместной встречаемости оценивается одинаково для всех пар слов. То есть более редкие пары достаточно зашумлены и несут меньше информации. Авторы предлагаю следующую функцию ошибки:

$$L = \sum_{i,j=1}^{V} f(X_{ij}) \cdot (w_i^T \widetilde{w_j} + b_i + \widetilde{b_j} - \log(X_{ij})), \tag{15}$$

где X_{ij} - показатель встречаемости слова j в контексте слова i, f - взвешивающая статистику встречаемости функци, V - размер словаря. Функция f должна отвечать некоторым свойствам:

1. $f(0) = 0$. Если f - непрерывная функция, то лучше рассматривать тождество, как сходящийся предел $\lim_{x \to 0} f(x) \log x^2$,

2. $f(x)$ неубывающая, чтобы не происходило перевзвешивания совместной встречаемости редких пар слов,

3. $f(x)$ должно быть достаточно низким на больших значениях x.

Существует множество функций, которые отвечают этим свойствам, но авторы статьи привели следующую:

$$f(x) = \begin{cases} (\frac{x}{x_{max}})^{\alpha} & if \quad x < x_{max}, \\ 1 & otherwise. \end{cases} \tag{16}$$

2. Разработки

2.1. Определение тональности

Реализация модели из статьи [10] производилась на языке Python и фреймфорке по созданию искусственных нейронных сетей tensorflow [21]. Часть кода по предобработке текста: разбиение на слова и замена специальных символов была позаимствована из исходного кода предоставленная авторами статьи для сохранения равных условий. Нами была реализована архитектура нейронной сети из этой статьи и эта реализация была взята за некоторый основной алгоритм для дальнейших улучшений.

В процессе проведения экспериментов не удалось добиться точных результатов из статьи [22], но полученный результат обучения было принято взять за базовый, то есть все дальнейшие результаты оценивались относительно его, для этого было проведено обучение с использованием кросс-валидации, чтобы удостовериться в воспроизводимости результатов нашей реализации. В рамках работы не было цели реализовать алгоритм, который в точности воспроизводит описанную в статье [10] модель. Также для проведения дополнительной проверки были обучены существующие реализации на фреймворке Torch [3], результаты обучения и проверки описаны в главе 5.

2.1.1. Распределение эмоционального окраса в предложениях

В процессе создания нейронной сети были выдвинуты теории о словах, после которых мнение, содержащееся в левой части предложения не имеет смысла из-за второй части. Например, предложения по структуре похожие "X, но Y". В предложениях подобного типа могли встречаться абсолютно разные слова между X и Y, но их свойства сохранялись. Такой тип предложений легко воспринимался в модели RNTN [17], и было важным понять, а верно ли это для CNN, для этого был проведен эксперимент, где по каждому документу из выборки скользило окно определенного размера, каждое окно было подано на предсказа-

ние метки класса в обученную модель. После этого строились графики в зависимости от расположения контекста (рис. 5, рис. 6).

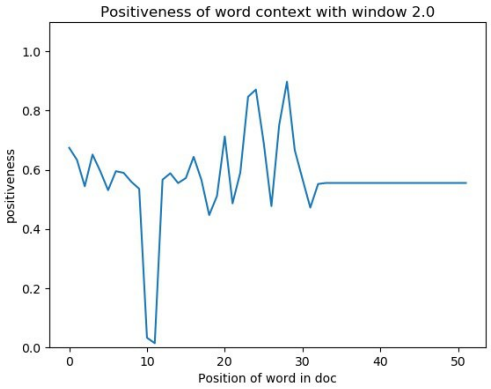

Рис. 5: Полярность документа с контекстом длины 2

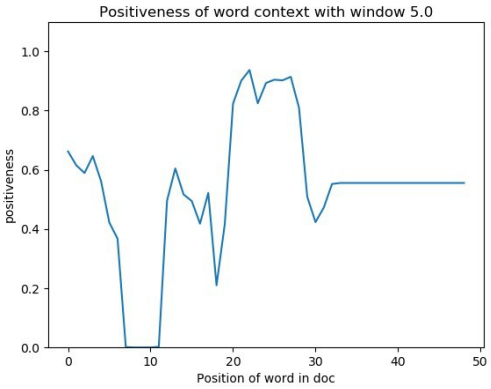

Рис. 6: Полярность документа с контекстом длины 5

Выявление подобных слов позволяло увеличить размер выборки в два раза за счет переворачивания частей предложения, чтобы из ”X, но Y”, предложение менялось на ”Y, но X” с заменой метки класса такого документа. Такой вид модификации мог быть использован для искусственного увеличения объема выборки. Об экспериментах и других способах синтетического увеличения датасета будет сказано далее в разделе 2.1.2 и главе 5

2.1.2. Увеличение выборки: замена синонимов

В процессе реализации и последующего анализа результатов было выявлено, что структура нейронной сети уже достаточно сложна и ее углубление будет приводить лишь к переобучению. Одна из возможных причин такого поведения нейросетей – недостаточный объем выборки. В процессе мозгового штурма были выдвинуты аналогии из области компьютерного зрения, решающие подобную проблему. Одной из таких аналогий стало применение техник искусственного увеличения объемов данных (англ. data augmentation). Эти техники направлены на искусственное увеличение датасета за счет элементарных преобразований

над изображениями, такими как: зеркальное отображение, поворот на 90 градусов и т.п. В случае же текста было принято решение на первом этапе заменять прилагательные и наречия на их синонимы. Синонимы были получены при помощи WordNet [13]. Это некоторый граф слов и их описаний, он заранее обучен и имеется в открытом доступе. С использованием этой техиники можно было получить выборку достаточно большого размера, но по некоторым экземплярам обучающей выборки не удавалось найти синонимы для используемых частей речи. Чтобы сеть могла достаточно хорошо обучиться и не переобучалась на конкретных экземплярах, присутствующих в увеличенной выборке, было введено ограничение на количество генерируемых документов по одному экземпляру. В этой работе были рассмотрены несколько вариантов такой генерации с рассмотрением процесса обучения. Такая модификация не затрагивала метку класса. И поэтому такое искусственное увеличение объема выборки не влияло на первоначальную сбалансированность выборки, то есть соотношение документов разных классов было с тем же отношением, что и до искусственного увеличения.

2.1.3. Увеличение выборки: замена слова на усредненное описание

В результате поиска современных статей о векторном представлении слов и текста, был найдем один алгоритм, который мог представить весь документ в виде одного вектора [1]. Этот алгоритм был использован в качестве замены какого-то слова в предложении на описание этого слова представленное алгоритмом [1]. Данная модификация не изменяла метки класса и также нейтральна к изменению сбалансированности выборки, то есть свойства выборки должны сохраняться. Также данный алгоритмах идет в двух модификациях, по которым были проведены эксперименты. Результаты экспериментов приведены в разделе 5.1.2.

2.1.4. Углубление архитектуры нейронной сети

Все действия, описанные в разделе 2.1.2, были направлены на то, чтобы попытаться искусственно увеличить выборку, для поиска более сложных взаимосвязей, которые могут быть распознаны более глубокими сетями. На примере работ из области компьютерного зрения, можно выявить, что более глубокие сети могут находить более сложные образы. Как следствие такие сети получают более высокий показатель точности. Но совместно с глубиной у нас увеличивается и количество обучаемых параметров, что требует большой обучающей выборки. Также важным знанием в этой области является и то, что каждый сверточный уровень представляет из себя некоторое абстрактное представление и добавление дополнительных слоев носит не случайный характер. Делается предположение, что каждый следующий слой является некоторой комбинацией простых признаков предыдущего слоя, что позволяет в конечном итоге распознавать очень сложные объекты. Подобные рассуждения были использованы в текущей работе, первый слой находил сочетания слов, а следующий сверточный слой находил отношения между сочетаниями слов.

2.2. Векторное представление слов

2.2.1. Модель Mimick и наша модификация

Суть разработки алгоритма лежит в предположении, что для поставленной задачи необходима дополнительна информация, например, о векторном представлении контекстного слова. Было выдвинуто предположение, что добавление контекстного слова при обучении модели mimick, позволит получить более общую модель. Была модифицирована функция f из формулы (5) следующим образом:

$$f(w, e_j) = O_T \cdot g(T_h \cdot (h_f^n \oplus h_b^0 \oplus e_j) + b_h) + b_T, \tag{17}$$

где e_j - векторное представление j-го слова, контекст слова w.

2.2.2. Наша модификация GloVe + SISG

Основой разработанного алгоритма является алгоритм GloVe. Его описание может быть найдено в главе 1. В алгоритме из раздела 1.2.4 было предложено обобщенное представление отношения между словом и его контекстом. Так и в алгоритме GloVe существует возможность замены части целевой функции, отвечающее за произведение векторов $w_i^T \widetilde{w_j}$. Необходимо модифицировать функцию ошибки из формулы (15) схожим образом:

$$L = \sum_{i,j=1}^{V} f(X_{ij}) \cdot (s\left(w_i, \widetilde{w}_i\right) + b_i + \widetilde{b_j} - \log\left(X_{ij}\right)). \tag{18}$$

Функция $s(\cdot, \cdot)$ была адаптирована из формулы 19, но с некоторыми изменениями относительно количества используемых размеров n-грамм. Фактически в функции оценки используется только один фактический размер символьных n-грамм. Пусть G_w - все векторные представления n-грамм слова w для заданного n, тогда оценивающую функцию можно записать в следующем виде:

$$s(w_t, w_c) = \sum_{g \in G_{w_t}} g^T w_c. \tag{19}$$

Предполагается, что за счет статистики совместной встречаемости контекста слова, можно получить качественные представления для символьных n-грамм, с помощью которых могут быть получены слова, отсутствующие в обучающей выборке.

Язык	Токены	|V|
CS	1m	46k
DE	1m	36k
EN	1m	17k
ES	1m	27k
FR	1m	25k
RU	1m	62k

Таблица 4: Статистика набора данных DATA-1M [2]

3. Способы валидации

3.1. Валидация на моделировании английского языка

В области векторного представления слов использование способа валидации на целевой задачи достаточно распространено. Такой способ валидации также демонстрирует некоторый способ применения техники transfer learning - использование результатов алгоритмов из другой задачи в решении целевой задачи. Сгенерированные вектора являются результатом решения нашей задачи, поэтому ключевым критерием оценки качества является оценка на задаче по обработке естественного языка. Авторы статьи [5] проводят такую валидацию на задаче языкового моделирования английского языка. Авторы статьи предлагают обучить искусственную нейронную LSTM сеть, решающую эту задачу. Качество обученной сети оценивается по метрике perplexity. Для проведения качественной и честной валидации, используется заранее подготовленная и обработанная выборка, где есть фиксированная тестовая и обучающая часть. Авторы статьи [5] позаимствовали выборку из работы [2]. Выборка представляет из себя набор текста разделенный по языкам. Базовые статистики по этому набору данных можно посмотреть в таблице 3.1. В каждом языке были отобраны около миллиона токенов, которые были заранее получены описанным в статье скриптом. Таким образом мы можем исключить отклонения из-за какой-либо обработки данных и другому разбиению текста на токены. Авторы статьи [5] не

приводили результаты на английском языке, поэтому нам необходимо было обучить эту сеть на этой выборке и также обучить сеть с различными инициализациями, подробнее об этом описано в разделе 5.2.2. Многие исследователи в области генерации векторных представлений слов используют и другие целевые задачи при валидации, но в рамках текущей работы они не были рассмотрены.

4. Детали реализации

4.1. Детали реализации методов векторного представления

В данном разделе содержится описание используемых языков программирования, библиотек и алгоритмов, которые позволили реализовать эту работу. Здесь же будут приведены базовые гиперпараметры, которые использовались при обучении искусственных нейронных сетей. Все разработанные алгоритмы и системы реализовываны на языке Python. Для обработки больших объемов текстов использовались скрипты на языке Perl и Bash. Предварительная обработка текста была написана на ящыке программирования Bash, часть обработки текстов для валидации была реализована на языке Python с использованием библиотек: nltk [11], Word Embeddings Benchmarks. Для реализации искусственных нейронных сетей использовался фреймворк глубинного обучения Tensorflow. Также важно отметить использование библиотеки tflearn, позволяющая описывать архитектуру нейронной сети в декларативном стиле. Все обучение производилось на CPU на процессоре Intel Core i7 2.8 Ghz и 16 Gb RAM. Размерность векторов была выбрана на основе максимально доступных данных, на которых была возможность проверить достоверность полученных результатов авторами статей, поэтому размерность вектора равнялась $d = 300$.

4.1.1. Mimick с контекстным вектором

При реализации данной сети возникли вопросы относительно того, каким образом обучать на большом объеме данных, ведь в оригинальной статье сложность обучения равнялась размеру словаря умноженному на сложность сети. И в их программной системе не было использовано алгоритмов оптимизирующих работу с большим объемом данных. Для решения этой задачи были использованы эффективные способы представления матриц с использование библиотек hdf5, dask. Эти библиотеки позволяли хранить большие данные в памяти только тот ку-

сок матрицы, который использовался только в текущий момент. Данное улучшение программной реализации позволило сократить время обучение и в принципе сделало обучение сети возможным. Гиперпараметры архитектуры искусственной нейронной сети описанной в разделе (5.2.1) были от части адаптированы из работ [5] и [16]. Таким образом были получены гиперпараметры: количество эпох обучения (4), количество ячеек в LSTM-слое (50), а также это дало основу для различных предположений относительно гиперпараметров размера выборки для одной итерации алгоритма и параметр learning rate (0.001). Векторные представления слов для валидации были сэмплированы из обучающей выборки. Из обучающей выборки по составленному словарю случайным образом выбиралось контекстное слово и подавалось на вход этой модели, полученные векторные представления записывались в словарь и таким образом был получен финальный словарь, на котором проводилась вся валидация. Также важно отметить, что для слов отсутствующих в обучающей выборке был составлен дополнительный словарь, где контекстное слово было заменено на единичный вектор. Также важно отметить, что это было сделано только для тех выборок, где у анализируемого слова не было контекстного слова. Эта проблема была описана в разделе (5.2.1).

4.1.2. GloVe + SISG

При реализации данной системы основной точкой интереса являлась модификация целевой функции, а не эффективное написание подсчета статистики встречаемости слов. Поэтому была использовала библиотека с открытым исходным кодом (`https://github.com/maciejkula/glove-python/tree/master/glove`), где подсчет данной статистики был реализован на языке C++. Архитектуру сети реализовали на чистой библиотеке Tensorflow с использованием off-heap хранилища для таблицы n-грамм, и разряженной матрицы совместной встречаемости. В целях дополнительной оптимизации программного алгоритма были использованы библиотеки по работе с off-heap хранилищем dask. Также возникали проблемы при обучении самой сети, ведь приходилось об-

новлять векторные представления в достаточно большой таблице векторных представлений символьных n-грамм. Для этого использовалось только частичное обновление параметров только участвующих при вычислении целевой функции и также было использован дополнительный сервис для распределения всей таблицы на несколько вычислительных машин. При обучении такой сети использовались аналогичные гиперпараметры из статьи [15]. Количество эпох при обучении равнялось 50, а размер контекста 5. Все остальные параметры подробно описаны в самой статье. Размер символьных n-грамм также выбирался исходя из лучших результатов из статьи [5].

5. Результаты экспериментов

5.1. Результаты экспериментов разработанных методов определения тональности

В данной главе описаны эксперименты по искусственному увеличению выборки, влияние этого на обучение модели и результаты проводимых экспериментов. Обучение нейронных сетей производилось без использования GPU и обучалось на стационарном компьютере. Обучение модели занимало от 30 минут до 3 часов, в случае использования CNN, и около одной недели при обучении модели Tree-LSTM.

5.1.1. Оценка методик на существующих реализациях

Всего было проведено два типа экспериментов с искусственно увеличенной выборкой:

1. обучение разных моделей;
2. углубление сети.

Весь эксперимент заключался в том, чтобы искусственно увеличить объем обучающей выборки и потом использовать его, для обучения той же самой модели с абсолютно идентичными гипер параметрами. Лучшие результаты можно посмотреть в таблице 5.

Модель	без увеличения	с увеличением	тип проверки
Tree-LSTM	81.7	82.58, 82.46	тестовая выборка
CNN (реализация torch)	81.3	81.8	кросс-валидация
CNN наша реализация	80.5	81	кросс-валидация
CNN 2conv 1x2	-	79	кросс-валидация
CNN 2conv 2x2	-	80	кросс-валидация

Таблица 5: Результаты проведенных экспериментов: последние две строчки с таблице, эксперимент по добавлению еще одного сверточного слоя; реализация harvard `https://github.com/harvardnlp/sent-conv-torch`

Полученная точность больше базового на 0.5, результат хоть и не значительный, но как показали эксперименты, можно брать любой алгоритм и производить его обучение с использованием этой методики, то есть берется абсолютно любой алгоритм, берется выборка, на котором он обучался, производится его искусственное увеличение, и получается результат лучше на 0.5%. Чтобы удостоверится в корректности нашего предположения был проведен еще ряд экспериментов, как на существующих имплементациях алгоритма, так и других алгоритмов, которые вышли после начала текущей работы. Этот эксперимент был повторен на реализации `https://github.com/harvardnlp/sent-conv-torch`, на алгоритме Tree-LSTM [20].

Второй эксперимент был по усложнению архитектуры сети с добавлением еще одного слоя свертки, для получения более сложных шаблонов (описано в разделе 2.1.4). В таблице 5 представлены результаты экспериментов двух сетей, на кросс валидации получен результат

на двух версиях сети, где отличается гиперпараметр размера свертки, один был размерностью 1×2, что предположительно, должно было дать шаблоны находящие сочетания соседних словосочетаний одной длины. И был рассмотрен другой вариант со сверкой 2×2, где находились сочетания не только сочетаний одинаковой длины, но и сочетаний разных длины. Как видно из таблицы, данный эксперимент не принес успеха и базовый результат модели.

5.1.2. Эксперимент: замена слова на его описание

Здесь приведено описание эксперимента по еще одному способу искусственного увеличения выборки, описанного в разделе 2.1.3. Эксперимент производился для одной модели из статьи [10]. На нашей реализации этого алгоритма. Представление документа из статьи [1] идет в двух вариантах: простое усреднение всех векторов-слов и усреднение с вычитание главных компонент из аглоритма SVD [7]. Результаты описаны в таблице 6

Модель	no-SVD	SVD
CNN наша реализация	75	76

Таблица 6: Результат экспериментов по замене слова на его усредненное описание

5.2. Результаты экспериментов разработанных методов векторного представления слов

5.2.1. Результаты модификации модели Mimick

В данном разделе будут описаны проведенные эксперименты по обучению и валидации алгоритма описанного в разделе 2.2.1. При обучении этой сети возникали различные проблемы с валидацией реализации основного алгоритма из статьи [16]. Реализация этого алгоритма была несовместимой с машиной на которой проводились эксперименты и было необходимо использовать виртуализацию для решения этой

проблемы. Результаты валидации на нескольких выборках по задаче векторного представления представлены в таблице 7.

Модель	Rare Words	WordSim353
fastText	0.43	0.73
Mimick	0.27	0.17
Модификация Mimick	0.29	0.21

Таблица 7: Результат экспериментов по получению векторных представлений из различных моделей

При проведении алгоритма выявилась проблема по составлению словаря используемого при валидации. Так как в описанной нами модификации при составлении вектора должно участвовать контекстное векторное представление слова, было необходимо каким-то образом инициализировать это слово из обучающей выборки или же использовать некоторый случайный вектор. Был выбран подход по составлению словаря по обучающей выборке, если же для какого-то слова не существовало контекстного слова или же само слово не присутствовало в обучающей выборке, то производилась случайная инициализация этого вектора. По этим результатам можно сказать, что как и работа [16], так и наша разработка не являются достаточным результатом, чтобы говорить о state-of-the-art результате, но важно отметить, что обучение этих моделей было проведено на основе какого-то словаря векторных представлений и логично предположить, что абсолютное улучшение результата здесь невозможно и важно оценивать не способность получать векторные представления лучше тех, на которых проходило обучение, а смотреть на способность алгоритма восстановить семантику этих векторных представлений. Таким образом можно сказать, что алгоритм [16] и наша реализация, могут достаточно хорошо распознавать семантику по существующим словарям. Также важно отметить, что результат полученный в данной работе превосходит результат [16].

5.2.2. Результаты GloVe + SISG

В данном разделе описаны проведенные эксперименты по обучению моделей из разделов (1.2.4, 2.2.2). При проведении экспериментов возникали некоторые проблемы по обучении сети. Эти проблемы были описаны в деталях реализации. По проведенным экспериментам по обучении искусственных нейронных сетей была проведена валидация на целевой задаче описанной в главе 3. Результаты различных векторных представлений на целевой задаче и на задачах по семантической схожести векторных представлений представлены в таблице 8.

Модель	LM (PPL)	Rare Words	WordSim353
random init	241	-	-
fastText	233	0.47	0.73
GloVe	229	0.478	0.759
наша модификация	225	0.48	0.757

Таблица 8: Результат экспериментов по получению векторных представлений из различных моделей на целевой задаче

Анализируя полученные результаты можно сделать вывод, что разработанный нами алгоритм способен получать получать state-of-the-art результат на целевой задаче и на задаче семантической схожести слов. Также важно отметить, что помимо этого, такая модификация алгоритма позволяет получать векторные представления для слов, отсутствующих в словаре.

Заключение

В данной магистерской диссертации были изучены и реализованы работы последних лет в области задачи определения тональности текста и векторного представления слов. Были разработаны алгоритмы машинного обучения и реализованы в виде программных систем способных определять тональность текста и способных получать векторные представления слов. В рамках диссертационной работы был разработан и реализован алгоритм векторного представления слов для слов, отсутствующих в словаре, были проведены эксперименты по обучению искусственных нейронных сетей для решения этой задачи и проведена валидация полученных векторных представлений.

Список литературы

[1] Arora Sanjeev, Liang Yingyu, Ma Tengyu. A simple but tough-to-beat baseline for sentence embeddings. — 2016.

[2] Botha Jan, Blunsom Phil. Compositional morphology for word representations and language modelling // International Conference on Machine Learning. — 2014. — P. 1899–1907.

[3] Collobert R., Kavukcuoglu K., Farabet C. Torch7: A Matlab-like Environment for Machine Learning // BigLearn, NIPS Workshop. — 2011.

[4] Distributed representations of words and phrases and their compositionality / Tomas Mikolov, Ilya Sutskever, Kai Chen et al. // Advances in neural information processing systems. — 2013. — P. 3111–3119.

[5] Enriching word vectors with subword information / Piotr Bojanowski, Edouard Grave, Armand Joulin, Tomas Mikolov // arXiv preprint arXiv:1607.04606. — 2016.

[6] Goller Christoph, Kuchler Andreas. Learning task-dependent distributed representations by backpropagation through structure // Neural Networks, 1996., IEEE International Conference on / IEEE. — Vol. 1. — 1996. — P. 347–352.

[7] Golub Gene H, Reinsch Christian. Singular value decomposition and least squares solutions // Numerische mathematik. — 1970. — Vol. 14, no. 5. — P. 403–420.

[8] Gradient-based learning applied to document recognition / Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner // Proceedings of the IEEE. — 1998. — Vol. 86, no. 11. — P. 2278–2324.

[9] Harris Zellig S. Distributional structure // Word. — 1954. — Vol. 10, no. 2-3. — P. 146–162.

[10] Kim Yoon. Convolutional neural networks for sentence classification // arXiv preprint arXiv:1408.5882. — 2014.

[11] Loper Edward, Bird Steven. NLTK: The natural language toolkit // Proceedings of the ACL-02 Workshop on Effective tools and methodologies for teaching natural language processing and computational linguistics-Volume 1 / Association for Computational Linguistics. — 2002. — P. 63–70.

[12] McAuley Julian, Leskovec Jure, Jurafsky Dan. Learning attitudes and attributes from multi-aspect reviews // Data Mining (ICDM), 2012 IEEE 12th International Conference on / IEEE. — 2012. — P. 1020–1025.

[13] Miller George A. WordNet: a lexical database for English // Communications of the ACM. — 1995. — Vol. 38, no. 11. — P. 39–41.

[14] Parsing natural scenes and natural language with recursive neural networks / Richard Socher, Cliff C Lin, Chris Manning, Andrew Y Ng // Proceedings of the 28th international conference on machine learning (ICML-11). — 2011. — P. 129–136.

[15] Pennington Jeffrey, Socher Richard, Manning Christopher. Glove: Global vectors for word representation // Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP). — 2014. — P. 1532–1543.

[16] Pinter Yuval, Guthrie Robert, Eisenstein Jacob. Mimicking word embeddings using subword RNNs // arXiv preprint arXiv:1707.06961. — 2017.

[17] Recursive deep models for semantic compositionality over a sentiment treebank / Richard Socher, Alex Perelygin, Jean Y Wu et al. // Proceedings of the conference on empirical methods in natural language processing (EMNLP) / Citeseer. — Vol. 1631. — 2013. — P. 1642.

[18] SemEval-2016 Task 5: Aspect Based Sentiment Analysis / Maria Pontiki, Dimitrios Galanis, Haris Papageorgiou et al. // Proceedings of the 10th International Workshop on Semantic Evaluation. — 2016.

[19] Symposium Sentiment Analysis. Sentiment Analysis Symposium // Sentiment Analysis Symposium. — 2016.

[20] Tai Kai Sheng, Socher Richard, Manning Christopher D. Improved semantic representations from tree-structured long short-term memory networks // arXiv preprint arXiv:1503.00075. — 2015.

[21] Tensorflow: Large-scale machine learning on heterogeneous distributed systems / Martın Abadi, Ashish Agarwal, Paul Barham et al. // arXiv preprint arXiv:1603.04467. — 2016.

[22] Zhang Ye, Wallace Byron. A Sensitivity Analysis of (and Practitioners' Guide to) Convolutional Neural Networks for Sentence Classification // arXiv preprint arXiv:1510.03820. — 2015.

A. Исходный код

A.1. Определение тональности

```
import gzip
import os
import pickle

import numpy as np
import tensorflow as tf
import tflearn
from gensim.models.keyedvectors import \
    KeyedVectors
from nltk.corpus import wordnet as wn
from nltk.tag import pos_tag
from sklearn.decomposition.truncated_svd import \
    TruncatedSVD
from sklearn.model_selection import KFold
from sklearn.utils import shuffle
from tflearn import conv_1d
from tflearn.datasets import imdb
from tflearn.layers.core import input_data, \
    dropout, fully_connected, flatten
from tflearn.layers.estimator import regression
from tflearn.layers.merge_ops import merge

from data_augmentation import wordnet_pos_code
from data_helpers import raw2csv


def load_embeddings(w2v, vocab, new_emb=None):
    """
    :param w2v: Word2Vec gensim instance
    :param new_emb:
    :param vocab:
    :return:
    """
    if new_emb is None:
        new_emb = dict()
    # initial matrix with random uniform TODO: replace 300 with arg
    initW = np.random.uniform(-0.25, 0.25,
                              (len(vocab), 300))
    for word, idx in vocab._mapping.iteritems():
        if word in new_emb:
            initW[idx] = new_emb[word]
            break
```

```
        if word in w2v:
            initW[idx] = w2v[word]
    return initW


def augment_with_avg(df, vocab, w2v):
    """
    :param w2v:
    :param df:
    :param vocab:
    :return:
    """

    def definition_to_word_vector(definition,
                                  with_pca=False):
        result = []
        for token in definition:
            if token in w2v:
                result.append(w2v[token])
        a = np.array(result)
        avg = np.average(a, axis=0)
        if with_pca:
            svd = TruncatedSVD()
            svd.fit(a)
            pc = svd.components_
            return avg - avg.dot(pc.T) * pc
        else:
            return avg

    out_emb = dict()
    out_data, out_labels = [], []
    for _, row in df.iterrows():
        document, label = row.drop(
            ['positive', 'negative']), row[
                           ['positive',
                            'negative']]
        document = document.as_matrix()
        raw_doc = []
        non_unk_words = []
        for i, w_id in enumerate(document):
            word = vocab.reverse(w_id)
            if w_id != 0:
                non_unk_words.append((i, word))
            raw_doc.append(word)
        num_of_words = len(non_unk_words)
        cnt = 0
```

```
        res = [document, document]
        while cnt < num_of_words ** 2:
            cnt += 1
            selected_word = np.random.randint(0,
                                              len(
                                                  non_unk_words))
            orig_idx, aug_word = non_unk_words[
                selected_word]
            aug_tag = None
            for w, t in pos_tag(raw_doc):
                if w == aug_word:
                    aug_tag = wordnet_pos_code(t)
                    break

            new_word = aug_word + '_mavg'
            syns = wn.synsets(aug_word, aug_tag)
            if len(syns) > 0:
                definition = syns[
                    0].definition().split(' ')
                wv, _ = definition_to_word_vector(
                    definition)
                out_emb[new_word] = wv
                vocab.freeze(False)
                vocab.add(new_word)
                copy = document.copy()
                copy[orig_idx] = vocab.get(
                    new_word)
                res[1] = copy
                break
        out_data += res
        out_labels += [label.as_matrix(),
                       label.as_matrix()]

    return np.array(out_data), np.array(
        out_labels), vocab, out_emb
```

```
from itertools import chain, product

import numpy as np
import pandas as pd
from nltk import pos_tag_sents
from nltk import word_tokenize
from nltk.corpus import wordnet as wn
from pip._vendor.distlib._backport import shutil
from tflearn.data_utils import VocabularyProcessor
from sklearn.utils import shuffle

```

```
12 from data_helpers import load_data_and_labels
13
14
15 def synonyms(word, tag):
16     try:
17         synonyms = wn.synsets(word, tag)
18         lemmas = set(
19             chain.from_iterable(
20                 [syn.lemma_names() for syn in
21                  synonyms]))
22         return list(lemmas)
23     except Exception:
24         return []
25
26
27 def wordnet_pos_code(tag):
28     if tag.startswith('NN'):
29         return wn.NOUN
30     elif tag.startswith('VB'):
31         return wn.VERB
32     elif tag.startswith('JJ'):
33         return wn.ADJ
34     elif tag.startswith('RB'):
35         return wn.ADV
36     else:
37         return ''
38
39
40 def my_tokenizer(raw_docs):
41     for raw_doc in raw_docs:
42         splitted = raw_doc.split(' ')
43         yield splitted
44
45
46 def load_docs_as_word_ids(posfile, negfile,
47                           vocab_pkl):
48     """
49     :param posfile:
50     :param negfile:
51     :param vocab_pkl:
52     :return: :type tuple(:type np.ndarray :type np.ndarray :type VocabularyProcessor)
53     """
54     import pandas as pd
55     pos = pd.read_csv(posfile,
56                       header=None, ).as_matrix()
57     neg = pd.read_csv(negfile,
```

```
                    header=None, ).as_matrix()
    data = np.concatenate((pos, neg), axis=0)
    pos_labels = [[0, 1] for _ in pos]
    neg_labels = [[1, 0] for _ in neg]
    labels = np.concatenate(
        [pos_labels, neg_labels], axis=0)
    data, labels = shuffle(data, labels)
    return data, labels, VocabularyProcessor.restore(
        vocab_pkl)


def load_train_mr_data():
    return load_docs_as_word_ids(
        'data/train/rt-polarity.pos',
        'data/train/rt-polarity.neg',
        'data/vocab_original.pkl')


def load_train_imdb_data():
    return load_docs_as_word_ids(
        'imdb_data/train/imdb.pos',
        'imdb_data/train/imdb.neg',
        'imdb_data/vocab_original.pkl')


def augment(load_data_fn, augmentation_path):
    data, labels, vocab = load_data_fn()
    MAX_DOCS = 2
    shutil.rmtree(augmentation_path,
                  ignore_errors=True)
    import os
    os.mkdir(augmentation_path)

    def reverse(documents):
        print "reverse doc"
        for _doc in documents:
            out = []
            for class_id in _doc:
                if class_id == 0:
                    continue
                out.append(
                    vocab.vocabulary_.reverse(
                        class_id))
            yield out

    tagged_sent = pos_tag_sents(reverse(data))
```

```
104     print "tagged sent"
105     doc_num = 0
106     total = len(tagged_sent)
107     for doc, label in zip(tagged_sent, labels):
108         print "proccess document {} of {} with doc len {}".format(
109             doc_num,
110             total,
111             len(doc))
112         doc_word_variants = []
113         total_varian_count = 1
114         for word, tag in doc:
115             if total_varian_count > 2000:
116                 print "exit: too many variants"
117                 break
118             variants = [word]
119             wn_tag = wordnet_pos_code(tag)
120             if wn_tag is wn.ADJ:  # or wn_tag is wn.ADV:
121                 syns = synonyms(word, wn_tag)[:6]
122                 variants += syns
123                 total_varian_count *= len(
124                     variants)
125             doc_word_variants.append(
126                 list(set(variants)))
127         docs = []
128         for new_doc in product(
129                 *doc_word_variants):
130             try:
131                 raw_doc = ' '.join(new_doc)
132                 docs.append(raw_doc)
133             except Exception:
134                 continue
135         docs = shuffle(docs)[:MAX_DOCS]
136         while len(docs) < MAX_DOCS:
137             docs *= 2
138         docs = docs[:MAX_DOCS]
139         # need to unfreeze vocab for dynamic fitting of it
140         vocab.vocabulary_.freeze(False)
141         print "generate {} number of documents from one".format(
142             len(docs) - 1)
143         docs = [doc_ for doc_ in
144                 vocab.fit_transform(docs)]
145         df = pd.DataFrame(docs, dtype=np.uint32)
146         f = augmentation_path + '/rt-polarity.pos' if \
147         label[
148             1] else augmentation_path + '/rt-polarity.neg'
149         df.to_csv(f, index=False, header=False,
```

```
                mode='a+')
        doc_num += 1
    vocab.save(augmentation_path + '/vocab.pkl')
```

```
import os
import pickle
import re

import numpy as np
import pandas as pd
from tflearn.data_utils import VocabularyProcessor


class DataSet(object):
    def __init__(self, folder, filename,
                 extensions=('pos', 'neg')):
        self._folder = folder
        self._filename = filename
        self._extensions = extensions
        data, labels, vocab = self.__preprocess_data()
        self.data = data
        self.labels = labels
        self.vocab_ = vocab

    def save(self, filepath):
        with open(filepath, 'ab+') as f:
            pickle.dump(self, f)

    @staticmethod
    def load(filepath):
        """
        :param filepath: path to pickle file of :type DataSet instance
        :return: instance of :type DataSet
        """
        with open(filepath, 'r') as f:
            return pickle.load(f)

    def word_index(self, word):
        return self.vocab_.vocabulary_.get(word)

    def __preprocess_data(self):
        positive_filename = self._filename + '.' + \
                            self._extensions[0]
        negative_filename = self._filename + '.' + \
                            self._extensions[1]
        positive_filepath = os.path.join(
            self._folder, positive_filename)
```

```
        negative_filepath = os.path.join(
            self._folder, negative_filename)
        return preprocess(positive_filepath,
                          negative_filepath,
                          save_vocab=False)


def __generate_dataset_loader(folder, filename):
    def load_from_dump(dump_path):
        if os.path.exists(dump_path):
            dataset = DataSet.load(dump_path)
        else:
            dataset = DataSet(folder, filename)

            dataset.save(dump_path)
        return dataset

    return load_from_dump


polarity_ds = __generate_dataset_loader('data',
                                        'rt-polarity')


def clean_str(string):
    """
    Tokenization/string cleaning for all datasets except for SST.
    Original taken from https://github.com/yoonkim/CNN_sentence/blob/master/process_data.py
    """
    string = re.sub(r"[^A-Za-z0-9(),!?\'\`]", " ",
                    string)
    string = re.sub(r"\'s", " \'s", string)
    string = re.sub(r"\'ve", " \'ve", string)
    string = re.sub(r"n\'t", " n\'t", string)
    string = re.sub(r"\'re", " \'re", string)
    string = re.sub(r"\'d", " \'d", string)
    string = re.sub(r"\'ll", " \'ll", string)
    string = re.sub(r",", " , ", string)
    string = re.sub(r"!", " ! ", string)
    string = re.sub(r"\(", " \( ", string)
    string = re.sub(r"\)", " \) ", string)
    string = re.sub(r"\?", " \? ", string)
    string = re.sub(r"\s{2,}", " ", string)
    return string.strip().lower()


```

```
def load_data_and_labels(positive_data_file,
                         negative_data_file):
    """
    Loads MR polarity data from files, splits the data into words and generates labels.
    Returns split sentences and labels.
    """
    # Load data from files
    positive_examples = list(
        open(positive_data_file, "r").readlines())
    positive_examples = [s.strip() for s in
                         positive_examples]
    negative_examples = list(
        open(negative_data_file, "r").readlines())
    negative_examples = [s.strip() for s in
                         negative_examples]
    # Split by words
    x_text = positive_examples + negative_examples
    x_text = [clean_str(sent) for sent in x_text]
    # Generate labels
    positive_labels = [[0, 1] for _ in
                       positive_examples]
    negative_labels = [[1, 0] for _ in
                       negative_examples]
    y = np.concatenate(
        [positive_labels, negative_labels], 0)
    return x_text, y


def space_split_tokenizer(iterator):
    for iterate in iterator:
        yield iterate.split(' ')


def build_vocab(data):
    max_doc_length = max(
        map(len, space_split_tokenizer(data)))
    vocab_processor = VocabularyProcessor(
        max_doc_length,
        tokenizer_fn=space_split_tokenizer)
    return vocab_processor


def preprocess(postive_path, negative_path,
               save_vocab=True,
               vocab_path='dataset_vocob.pkl'):
    data, labels = load_data_and_labels(
```

```
        postive_path, negative_path)
    vocab = build_vocab(data)
    docs = np.array(
        [w for w in vocab.fit_transform(data)])
    if save_vocab:
        vocab.save(vocab_path)
    return docs, labels, vocab


def raw2csv(
        file_path='data/rt-polarity_data_labels.csv'):
    data, labels, v = preprocess(
        'data/rt-polarity.pos',
        'data/rt-polarity.neg',
        vocab_path='data/vocab_original.pkl')
    frame = pd.DataFrame(
        np.concatenate([data, labels], axis=1))
    columns = [str(i) for i in
               xrange(len(frame.columns))]
    columns[-1] = 'positive'
    columns[-2] = 'negative'
    frame.columns = columns
    frame.to_csv(file_path, header=False,
                 index=False)
    return frame, v


def save_separate_files(tt, type):
    tt[tt['positive'] == 1].drop(
        ['positive', 'negative'], axis=1).to_csv(
        "data/{}/rt-polarity.pos".format(type),
        index=False, header=False, mode='w+')
    tt[tt['positive'] == 0].drop(
        ['positive', 'negative'], axis=1).to_csv(
        "data/{}/rt-polarity.neg".format(type),
        index=False, header=False, mode='w+')
```

```
import gzip
import math
import os
import pickle
import time

import numpy as np
import pandas as pd
import tensorflow as tf
import tflearn
```

```
from gensim.models.keyedvectors import \
    KeyedVectors
from gensim.models.word2vec import Word2Vec, \
    LineSentence
from sklearn.model_selection import KFold
from sklearn.utils import shuffle
from tflearn import conv_1d
from tflearn.datasets import imdb
from tflearn.layers.conv import global_max_pool, \
    conv_2d
from tflearn.layers.core import input_data, \
    dropout, fully_connected, flatten
from tflearn.layers.estimator import regression
from tflearn.layers.merge_ops import merge
from data_augmentation import \
    load_docs_as_word_ids, my_tokenizer, augment
from data_helpers import raw2csv, \
    save_separate_files


def load_embeddings(filename, vocab):
    """
    :param filename:
    :type vocab: tensorflow.contrib.learn.python.learn.preprocessing.categorical_vocabulary.CategoricalVocabulary
    :param vocab:
    :return:
    """
    w2v = KeyedVectors.load_word2vec_format(
        filename, binary=True)
    # initial matrix with random uniform TODO: replace 300 with arg
    initW = np.random.uniform(-0.25, 0.25,
                              (len(vocab), 300))
    for word, idx in vocab._mapping.iteritems():
        if word in w2v:
            initW[idx] = w2v[word]
    return initW


def load_imdb_vocab(path="imdb.dict.pkl"):
    path = imdb.get_dataset_file(
        path, "imdb.dict.pkl",
        "http://www.iro.umontreal.ca/~lisa/deep/data/imdb.dict.pkl.gz")
    if path.endswith(".gz"):
        f = gzip.open(path, 'rb')
    else:
        f = open(path, 'rb')
```

```
57
58      vocab = pickle.load(f)
59      return vocab
60
61
62  def get_network(
63          vocab_size,
64          max_document_length,
65          filter_sizes=(2, 3, 4, 5),
66          num_filters=100,
67          dropout_keep_prob=0.5,
68          n_classes=2,
69          embedding_dim=300
70  ):
71      pass
72
73
74  def train_embed(embedding_dim, vocab):
75      model = Word2Vec(
76          LineSentence('data/rt-data.txt'),
77          embedding_dim, window=10, min_count=1)
78      initW = np.random.uniform(-0.25, 0.25, (
79      len(vocab), embedding_dim))
80      for word, idx in vocab._mapping.iteritems():
81          if word in model:
82              initW[idx] = model[word]
83      return initW
84
85
86  def learn():
87      from data_augmentation import my_tokenizer
88      folds = 10
89      k_fold = KFold(folds, shuffle=True)
90      file_path = 'data/rt-polarity_data_labels.csv'
91      df, _ = raw2csv(file_path)
92      for train_indicies, test_indicies in k_fold.split(
93              df):
94          # save_separate_files(df.loc[train_indicies, :], 'train')
95          # save_separate_files(df.loc[test_indicies, :], 'test')
96          # augment()
97          X_train, y_train, vocab = load_docs_as_word_ids(
98              'data/train/rt-polarity.pos',
99              'data/train/rt-polarity.neg',
100             'data/augmentation/vocab.pkl')
101         X_test, y_test, _ = load_docs_as_word_ids(
102             'data/test/rt-polarity.pos',
```

```
        'data/test/rt-polarity.neg',
        'data/augmentation/vocab.pkl')
    X_train, y_train = shuffle(X_train,
                               y_train)
    X_test, y_test = shuffle(X_test, y_test)
    max_document_length = X_train.shape[1]
    vocab = vocab.vocabulary_

    print "loaded vocab, size: {}".format(
        len(vocab))
    vocab_size = len(vocab)

    print "Loaded data"
    print "Max document length {}".format(
        max_document_length)

    dropout_keep_prob = 0.5
    batch_size = 64
    n_classes = 2
    embedding_dim = 300
    filter_sizes = [2, 3, 4]
    num_filters = 100
    # snapshot_every = 200
    folds = 3
    # max_document_length = 280
    multichannel = False
    nonstatic = True

    def get_run_name():
        mult = "M"
        stat = "S"
        nonstat = "NS"
        params = "dkp{}_filters{}_nb_filters{}_{}" \
            .format(dropout_keep_prob,
                    "_".join(
                        map(lambda b: str(b),
                            filter_sizes)),
                    num_filters, time.time())
        name = ""
        if multichannel:
            name += mult
        elif nonstatic:
            name += nonstat
        else:
            name += stat
        name += "_"
```

```
149         return name + params
150
151     print "loading embeddings"
152     # pretrained = train_embed(embedding_dim, vocab)
153     pretrained = load_embeddings(
154         'GoogleNews-vectors-negative300.bin',
155         vocab)
156     # pretrained = None
157     # for train_indices, test_indices in ten_fold.split(X, y):
158     tf.reset_default_graph()
159     network = input_data(
160         shape=[None, max_document_length],
161         name='input')
162     emb = tflearn.embedding(network,
163                             input_dim=vocab_size,
164                             output_dim=embedding_dim)
165     branches = []
166     for filter_size in filter_sizes:
167         conv = conv_1d(emb, num_filters,
168                        filter_size,
169                        padding='valid',
170                        activation='relu',
171                        regularizer="L2",
172                        name="conv-{}".format(
173                            filter_size))
174         branches.append(conv)
175     branches = map(
176         lambda b: tf.expand_dims(b, 2),
177         branches)
178     network = merge(branches, axis=1,
179                     mode='concat')
180     # network = conv_2d(network, 32, [1, 2], padding='valid', activation='relu', name='2conv2x2')
181     network = tf.reduce_max(network, axis=1)
182     network = flatten(network)
183     network = dropout(network,
184                       dropout_keep_prob)
185     network = fully_connected(network,
186                               n_classes,
187                               activation='softmax')
188     network = regression(network,
189                          optimizer='adam',
190                          learning_rate=0.0001,
191                          loss='categorical_crossentropy',
192                          name='target')
193     run_name = 'mr_std_conf'
194     checkpoint_path = 'results/' + run_name
```

```
        os.makedirs(checkpoint_path)
        model = tflearn.DNN(network,
                            tensorboard_verbose=3,
                            checkpoint_path=checkpoint_path,
                            tensorboard_dir='logs')
        if nonstatic or multichannel:
            model.set_weights(emb.W, pretrained)

        model.fit(X_train, y_train,
                  n_epoch=20,
                  validation_set=(X_test, y_test),
                  show_metric=True,
                  batch_size=batch_size,
                  snapshot_step=100,
                  snapshot_epoch=False,
                  validation_batch_size=512,
                  run_id=run_name)
```

A.2. Векторное представление слов

```
from __future__ import division

import datetime
import pickle
import sys

import numpy as np
import tensorflow as tf
from nltk import ngrams

import config
import data_helper


class GloveModule:
    def __init__(self, coocur_file, ngram_table_file):
        print("Load vocab and cooc matrix...")
        self.vocab, cooccurrence_matrix, self.ngram_vocab, self.max_ngram_cnt = \
            data_helper.get_wiki_corpus_and_dump(coocur_file, ngram_table_file)

        self.cooccurrence_matrix = list(cooccurrence_matrix.items())
        vocab_size = len(self.vocab)
        ngram_table_size = len(self.ngram_vocab)
        print("Done loading vocab and cooc.")
        print("Creating tensorflow graph...")
        self.graph = tf.Graph()
```

```
27
28          with self.graph.as_default():
29              with self.graph.device("/cpu:0"):
30                  count_max = tf.constant([config.COUNT_MAX], dtype=tf.float32)  # tf          -
31                  alpha = tf.constant([config.SCALING_FACTOR], dtype=tf.float32)  # tf          -
32                  # word_ids
33                  self.focal_input_w_id = tf.placeholder(tf.int32, shape=[config.BATCH_SIZE])
34                  self.context_input_w_id = tf.placeholder(tf.int32, shape=[config.BATCH_SIZE])
35                  # ngram_ids
36                  self.focal_input = tf.placeholder(tf.int32, shape=[config.BATCH_SIZE, self.max_ngram_cnt])
37                  self.context_input = tf.placeholder(tf.int32, shape=[config.BATCH_SIZE, self.max_ngram_cnt])
38
39                  self.cooccurrence_count = tf.placeholder(tf.float32, shape=[config.BATCH_SIZE])
40                  self.ngram_embeddings = tf.Variable(
41                      tf.random_uniform([ngram_table_size, config.EMBEDDING_SIZE], 1.0, -1.0)
42                  )
43
44                  focal_word_biases = tf.Variable(
45                      tf.random_uniform([vocab_size], 1.0, -1.0)
46                  )
47
48                  context_word_biases = tf.Variable(
49                      tf.random_uniform([vocab_size], 1.0, -1.0)
50                  )
51
52                  focal_embedding = tf.nn.embedding_lookup([self.ngram_embeddings], self.focal_input_w_id)
53                  context_embedding = tf.nn.embedding_lookup([self.ngram_embeddings], self.context_input_w_id)
54                  focal_bias = tf.nn.embedding_lookup([focal_word_biases], self.focal_input)
55                  context_bias = tf.nn.embedding_lookup([context_word_biases], self.context_input)
56
57                  weighting_factor = tf.minimum(
58                      1.0,
59                      tf.pow(
60                          tf.div(self.cooccurrence_count, count_max),
61                          alpha
62                      )
63                  )
64                  # s(w_i, w_j) = \sum_{z \in G_j}{w_i^T*z}
65                  context_word_embedding = tf.reduce_sum(context_embedding, axis=0)
66                  embedding_product = tf.reduce_sum(tf.multiply(focal_embedding, context_word_embedding), axis=
67
68                  log_cooccurrences = tf.log(tf.to_float(self.cooccurrence_count))
69
70                  distance_expr = tf.square(tf.add_n([
71                      embedding_product,
72                      focal_bias,
```

```
                context_bias,
                tf.negative(log_cooccurrences)
            ]))

            single_losses = tf.multiply(weighting_factor, distance_expr)
            self.total_loss = tf.reduce_sum(single_losses)
            # with self.graph.device("/cpu:0"):
            tf.summary.scalar('loss', self.total_loss)
            # self.optimizer = tf.train.AdamOptimizer(config.LEARNING_RATE).minimize(self.total_loss)
            self.optimizer = tf.train.AdagradOptimizer(config.LEARNING_RATE).minimize(self.total_loss)
        print("Done creating")

    def get_ngram_ids(self, word_id):
        result = np.zeros(self.max_ngram_cnt)
        for i, subword in enumerate(ngrams('<' + self.vocab[word_id] + '>', config.NGRAM_SIZE)):
            result[i] = self.ngram_vocab[subword]
        return np.array(result)

    def get_batch(self, idx):
        ww, counts = zip(*self.cooccurrence_matrix[idx:idx + config.BATCH_SIZE])
        i_s, j_s = zip(*ww)
        nis = []
        for w_id in i_s:
            nis.append(self.get_ngram_ids(w_id))
        njs = []
        for w_id in j_s:
            njs.append(self.get_ngram_ids(w_id))
        return i_s, j_s, np.array(nis), np.array(njs), counts

    def begin(self):
        print("Prepare cooccurrence matrix...")

        # i_indices, j_indices, counts = zip(*self.cooccurrence_matrix)
        print("Done preparing")
        print("Get batches...")
        # self.batches = list(self.batchify(i_indices, j_indices, cooc))
        print("Done get batches.")
        print("Begin training: {}".format(datetime.datetime.now().time()))
        print("==================")
        sys.stdout.flush()
        with tf.Session(graph=self.graph) as session:
            merged = tf.summary.merge_all()
            batch_writer = tf.summary.FileWriter('./logs/batch')
            tf.initialize_all_variables().run()
            for epoch in range(config.NUM_EPOCHS):
                # shuffle(self.batches)
```

```
119                 print("Batches shuffled")
120                 print("----------------")
121                 sys.stdout.flush()
122                 accumulated_loss = 0
123                 total = len(self.cooccurrence_matrix)
124                 num_batches = total / config.BATCH_SIZE
125                 batch_index = 0
126                 for i in range(0, total, config.BATCH_SIZE):
127                     batch_index += 1
128                     i_s, j_s, nis, njs, counts = self.get_batch(i)
129 
130                     if len(counts) != config.BATCH_SIZE:
131                         continue
132                     feed_dict = {
133                         self.focal_input_w_id: i_s,
134                         self.context_input_w_id: j_s,
135                         self.focal_input: nis,
136                         self.context_input: njs,
137                         self.cooccurrence_count: counts
138                     }
139                     summaries, _, total_loss_, = session.run(
140                         [merged, self.optimizer, self.total_loss], feed_dict=feed_dict)
141                     accumulated_loss += total_loss_
142                     if (batch_index + 1) % config.REPORT_BATCH_SIZE == 0:
143                         print("Epoch: {0}/{1}".format(epoch + 1, config.NUM_EPOCHS))
144                         print("Batch: {0}/{1}".format(batch_index + 1, num_batches))
145                         print("Average loss: {}".format(accumulated_loss / config.REPORT_BATCH_SIZE))
146                         print("----------------")
147                         batch_writer.add_summary(summaries, epoch * num_batches + batch_index)
148                         sys.stdout.flush()
149                         accumulated_loss = 0
150                 print("Epoch finished: {}".format(datetime.datetime.now().time()))
151                 print("==================")
152 
153                 sys.stdout.flush()
154             batch_writer.close()
155             final_embeddings = self.ngram_embeddings.eval()
156             print("End: {}".format(datetime.datetime.now().time()))
157             final_dict = {}
158             for word, idx in self.vocab.items():
159                 w_emb = np.zeros((1, config.EMBEDDING_SIZE), dtype=np.float32)
160                 for ngram_id in self.get_ngram_ids(idx):
161                     w_emb += final_embeddings[ngram_id]
162                 final_dict[word] = w_emb
163             final_table = {}
164             for ngram_id in range(final_embeddings.shape[0]):
```

```
                final_table[ngram_table[ngram_id]] = final_embeddings[ngram_id]
        return final_dict, final_table
```

```
import os
import pickle
import glove
from nltk import ngrams
import argparse
import config
from GloVe import GloveModule

def get_wiki_corpus_and_dump(coocur_file, ngram_table_file):
    corpus = glove.Corpus.load(coocur_file)
    vocab = corpus.dictionary
    coocur = corpus.matrix
    max_ngram_cnt = -1
    if os.path.exists(ngram_table_file):
        ngram_table, max_ngram_cnt = pickle.load(open(ngram_table_file))
    else:
        unique_ngrams = set()
        for _, word in vocab.items():
            cur_ngrams = ngrams('<' + word + '>', config.NGRAM_SIZE)
            cur_ngram_cnt = len(cur_ngrams)
            if cur_ngram_cnt > max_ngram_cnt:
                max_ngram_cnt = cur_ngram_cnt
            unique_ngrams += set(cur_ngrams)
        ngram_table = {ngram: n_id for n_id, ngram in enumerate(unique_ngrams)}
        pickle.dump((ngram_table, max_ngram_cnt), open(ngram_table_file, 'wb+'))

    return coocur, vocab, ngram_table, max_ngram_cnt









if __name__ == '__main__':


    parser = argparse.ArgumentParser()
    parser.add_argument('--coocur-file')
    parser.add_argument('--nragm-table-file')
    args = parser.parse_args()
    module = GloveModule(args.coocur_file, args.ngram_table_file)
    embeddings, final_table = module.begin()
    with open('data/emb/glove{}.pkl'.format(config.EMBEDDING_SIZE), 'wb+') as f:
        pickle.dump(embeddings, f, protocol=4)
```

```
    with open('data/emb/glove_ngram{}.pkl', 'wb+') as f:
        pickle.dump(final_table, f, protocol=4)
```

```
from __future__ import print_function
import argparse
import pprint
import gensim
import subprocess

from tqdm import tqdm

from glove import Glove
from glove import Corpus


def get_line_no(fp):
    p = subprocess.Popen(['wc', '-l', fp], stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    result, err = p.communicate()
    if p.returncode != 0:
        raise IOError(err)
    return int(result.strip().split()[0])


def read_corpus(filename):
    delchars = [chr(c) for c in range(256)]
    delchars = [x for x in delchars if not x.isalnum()]
    delchars.remove(' ')
    delchars = ''.join(delchars)

    table = str.maketrans(dict.fromkeys(delchars))

    with open(filename, 'r') as datafile:
        for line in tqdm(datafile, total=get_line_no(filename)):
            yield line.lower().translate(table).split(' ')


def read_wikipedia_corpus(filename):
    # We don't want to do a dictionary construction pass.
    corpus = gensim.corpora.WikiCorpus(filename, dictionary={})

    for text in corpus.get_texts():
        yield text


if __name__ == '__main__':

    # Set up command line parameters.
```

```
45      parser = argparse.ArgumentParser(description='Fit a GloVe model.')
46
47      parser.add_argument('--create', '-c', action='store',
48                          default=None,
49                          help=('The filename of the corpus to pre-process. '
50                              'The pre-processed corpus will be saved '
51                              'and will be ready for training.'))
52      parser.add_argument('-wiki', '-w', action='store_true',
53                          default=False,
54                          help=('Assume the corpus input file is in the '
55                              'Wikipedia dump format'))
56      parser.add_argument('--train', '-t', action='store',
57                          default=0,
58                          help=('Train the GloVe model with this number of epochs.'
59                              'If not supplied, '
60                              'We\'ll attempt to load a trained model'))
61      parser.add_argument('--parallelism', '-p', action='store',
62                          default=1,
63                          help=('Number of parallel threads to use for training'))
64      parser.add_argument('--query', '-q', action='store',
65                          default='',
66                          help='Get closes words to this word.')
67      args = parser.parse_args()
68
69      if args.create:
70          # Build the corpus dictionary and the cooccurrence matrix.
71          print('Pre-processing corpus')
72
73          if args.wiki:
74              print('Using wikipedia corpus')
75              get_data = read_wikipedia_corpus
76          else:
77              get_data = read_corpus
78
79          corpus_model = Corpus()
80          corpus_model.fit(get_data(args.create), window=10)
81          corpus_model.save('corpus.model')
82
83          print('Dict size: %s' % len(corpus_model.dictionary))
84          print('Collocations: %s' % corpus_model.matrix.nnz)
85
86      if args.train:
87          # Train the GloVe model and save it to disk.
88
89          if not args.create:
90              # Try to load a corpus from disk.
```

```
            print('Reading corpus statistics')
            corpus_model = Corpus.load('corpus.model')

            print('Dict size: %s' % len(corpus_model.dictionary))
            print('Collocations: %s' % corpus_model.matrix.nnz)

        print('Training the GloVe model')

        glove = Glove(no_components=100, learning_rate=0.05)
        glove.fit(corpus_model.matrix, epochs=int(args.train),
                  no_threads=args.parallelism, verbose=True)
        glove.add_dictionary(corpus_model.dictionary)

        glove.save('glove.model')

    if args.query:
        # Finally, query the model for most similar words.
        if not args.train:
            print('Loading pre-trained GloVe model')
            glove = Glove.load('glove.model')

        print('Querying for %s' % args.query)
        pprint.pprint(glove.most_similar(args.query, number=10))
```

```
import subprocess
from collections import Counter
from time import clock
from typing import List, Dict

import numpy as np
import pandas as pd
from tqdm import tqdm


def get_line_no(fp):
    p = subprocess.Popen(['wc', '-l', fp], stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    result, err = p.communicate()
    if p.returncode != 0:
        raise IOError(err)
    return int(result.strip().split()[0])


def load_data(path: str):
    with open(path, 'r') as f:
        for line in tqdm(f, total=get_line_no(path)):
            yield tokenize(line)

```

```
24
25  def tokenize(sentence_str: str):
26      # TODO: find better solution for this
27      return sentence_str.split(" ")
28
29
30  def generate_contexts(sentence: List[str], context_size):
31      last_index = len(sentence)
32
33      def window(start, end):
34          return sentence[max(start, 0): min(end, last_index)]
35
36      for i, word in enumerate(sentence):
37          yield window(i - context_size, i), word, window(i + 1, i + context_size)
38
39
40  def charseq(word, c2i):
41      chars = []
42      for c in word:
43          if c not in c2i:
44              c2i[c] = len(c2i)
45          chars.append(c2i[c])
46      return chars
47
48
49  def prepare_corpus(
50          path: str,
51          context_size: int = 5,
52          min_count: int = 3,
53          w2v_file: str = '../sa_cnn/GoogleNews-vectors-negative300.bin'
54  ) -> (Dict[str, int], np.ndarray):
55      """
56      :param w2v_file:
57      :param min_count:
58      :param context_size:
59      :param path:
60      """
61      print('prepare data, loading vectors')
62      from gensim.models import KeyedVectors
63      w2v = KeyedVectors.load_word2vec_format(w2v_file, binary=True)
64      # filter rare words and index every word so we get id -> word
65      st = clock()
66      print('constructing vocab')
67      word_counts = Counter()
68      for sentence in load_data(path):
69          word_counts.update(sentence)
```

```
    df = pd.DataFrame(list(word_counts.items()), columns=['word', 'counts'])
    df = df[df['counts'] > min_count]
    df = df.reset_index(drop=True)
    vocab = {idx: w for idx, w in df['word'].to_dict().items() if w in w2v}
    word_to_id = {v: k for k, v in vocab.items()}
    vocab_size = len(vocab)
    print("vocab get time: ", clock() - st)
    print("vocab size", vocab_size)

    all_pairs = []
    char_vocab = {}
    for sentence in load_data(path):
        for l_ctx, word, r_ctx in generate_contexts(sentence, context_size):
            if word in word_to_id:
                charseq(word, char_vocab)
                contexts = l_ctx + r_ctx
                for ctx in contexts:
                    if ctx in word_to_id:
                        all_pairs.append((word_to_id[word], word_to_id[ctx]))
    return all_pairs, vocab, char_vocab, w2v


def load_dataset():
    return prepare_corpus(
        'wiki.txt', context_size=5, min_count=1
    )
```

```
import tensorflow as tf
import tflearn
import numpy as np
from tflearn import input_data, embedding, bidirectional_rnn, BasicLSTMCell, regression
from tflearn.data_utils import pad_sequences
from tflearn.models.dnn import DNN

from data_helper import load_dataset

max_chars = 128
emb_size = 300
char_emb_size = 20
num_lstm_units = 50


# network graph
def get_network(char_vocab_size):
    context_word_vec = input_data([None, 300], name="context_w", dtype=tf.float32)
    char_input = input_data([None, max_chars], name="word_chars", dtype=tf.float32)
    char_embeddings = embedding(
```

```
        char_input,
        input_dim=char_vocab_size,
        output_dim=char_emb_size,
        trainable=True,
        name='char_embeddings'
    )

    net = bidirectional_rnn(char_embeddings,
                            BasicLSTMCell(num_lstm_units),
                            BasicLSTMCell(num_lstm_units))
    bilstm_out_with_context = tflearn.merge([net, context_word_vec], mode='concat')
    net = tflearn.fully_connected(bilstm_out_with_context, 300)
    return regression(net, optimizer='adam', loss='mean_square', metric='R2')


def main():
    # preparing data to fit
    dataset, word_vocab, char_vocab, w2v = load_dataset()
    context_vectors = []
    char_data = []
    targets = []
    print(len(dataset))
    print(char_vocab)
    for (w_id, c_id) in dataset:
        chars = []
        for ch in word_vocab[w_id]:
            chars.append(char_vocab[ch]) if ch in char_vocab else None
        char_data.append(chars)
        context_vectors.append(w2v[word_vocab[c_id]])
        targets.append(w2v[word_vocab[w_id]])
    context_vectors = np.array(context_vectors)
    char_data = pad_sequences(char_data, max_chars, value=0.)
    print(char_data.shape)
    print(context_vectors.shape)
    # network session
    net = get_network(len(char_vocab))
    model = DNN(net, tensorboard_dir='./logs/')

    # fitting the model
    model.fit({
        "context_w": context_vectors,
        "word_chars": char_data
    }, np.array(targets),
        n_epoch=5,
        validation_set=0.1,
        show_metric=True,
```

```
        batch_size=64,
        shuffle=True
    )

    model.save('model.bin')


if __name__ == '__main__':
    main()
```

```
#!/usr/bin/env bash
#
# Copyright (c) 2016-present, Facebook, Inc.
# All rights reserved.
#
# This source code is licensed under the BSD-style license found in the
# LICENSE file in the root directory of this source tree. An additional grant
# of patent rights can be found in the PATENTS file in the same directory.
#

set -e

normalize_text() {
    sed -e "s/'/'/g" -e "s/ /'/g" -e "s/''/ /g" -e "s/' ' /g" -e "s/“/\"/g" -e "s/”/\"/g" \
        -e 's/"/ " /g' -e 's/\./ \. /g' -e 's/<br \/>/ /g' -e 's/, / , /g' -e 's/(/ ( /g' -e 's/)/ ) /g' -e 's/\!/ \! /g' \
        -e 's/\?/ \? /g' -e 's/\;/ /g' -e 's/\:/ /g' -e 's/-/ - /g' -e 's/=/ /g' -e 's/=/ /g' -e 's/*/ /g' -e 's/|/ /g' \
        -e 's/«/ /g' | tr 0-9 " "
}

export LANGUAGE=en_US.UTF-8
export LC_ALL=en_US.UTF-8
export LANG=en_US.UTF-8

NOW=$(date +"%Y%m%d")

ROOT="data/wikimedia/${NOW}"
mkdir -p "${ROOT}"
echo "Saving data in ""$ROOT"
read -r -p "Choose a language (e.g. en, bh, fr, etc.): " choice
LANG="$choice"
echo "Chosen language: ""$LANG"
read -r -p "Continue to download (WARNING: This might be big and can take a long time!)(y/n)? " choice
case "$choice" in
  y|Y ) echo "Starting download...";;
  n|N ) echo "Exiting";exit 1;;
  * ) echo "Invalid answer";exit 1;;
esac
```

```
wget -c "https://dumps.wikimedia.org/""$LANG""wiki/latest/""${LANG}""wiki-latest-pages-articles.xml.bz2"\
  -P "${ROOT}"
echo "Processing ""$ROOT"/"$LANG""wiki-latest-pages-articles.xml.bz2"
bzip2 -c -d "$ROOT"/"$LANG""wiki-latest-pages-articles.xml.bz2" | awk '{print tolower($0);}' | \
  perl wikifil.pl | normalize_text | awk '{if (NF>1) print;}' | tr -s " " | gshuf > "${ROOT}"/wiki."${LANG}".txt
```

```
$/=">";                     # input record separator
while (<>) {
  if (/<text /) {$text=1;}  # remove all but between <text> ... </text>
  if (/#redirect/i) {$text=0;}  # remove #REDIRECT
  if ($text) {
    # Remove any text not normally visible
    if (/<\/text>/) {$text=0;}
    s/<.*>//;               # remove xml tags
    s/&amp;/&/g;            # decode URL encoded chars
    s/&lt;/</g;
    s/&gt;/>/g;
    s/<ref[^<]*<\/ref>//g;  # remove references <ref...> ... </ref>
    s/<[^>]*>//g;           # remove xhtml tags
    s/\[http:[^] ]*/[/g;    # remove normal url, preserve visible text
    s/\|thumb//ig;          # remove images links, preserve caption
    s/\|left//ig;
    s/\|right//ig;
    s/\|\d+px//ig;
    s/\[\[image:[^\[\]]*\|//ig;
    s/\[\[category:([^|\]]*)[^]]*\]\]/[[$1]]/ig;  # show categories without markup
    s/\[\[[a-z\-]*:[^\]]*\]\]//g;  # remove links to other languages
    s/\[\[[^\|\]]*\|/[[/g;  # remove wiki url, preserve visible text
    s/{\{[^}]*}}//g;
    s/{[^}]*}//g;
    s/\[//g;
    s/\]//g;
    s/&[^;]*;/ /g;          # remove URL encoded chars
    $_=" $_ ";
    chop;
    print $_;
  }
}
```