Санкт-Петербургский государственный университет
Кафедра компьютерного моделирования и многопроцессорных систем

Южанин Артур Ирекович

Магистерская диссертация

Вопросно-ответные системы на основе обработки текстов на естественном языке с применением технологий распределенных вычислений

Направление 02.04.02

Прикладная математика и информатика

Магистерская программа

Вычислительные технологии

Научный руководитель,
кандидат физ.-мат. наук, доцент
Корхов В. В.

Санкт-Петербург

2017

Содержание

Введение

В последнее время в мире информационных технологий возникло множество трендов. Одними из самых популярных трендов являются вопросно-ответные системы, голосовые помощники и всевозможные “умные ассистенты”. Как правило, в реализациях подобного рода проектов имеется нечто общее, а именно — способность понимать информацию, поступающую от пользователя, и проводить ее анализ.

Очевидно, что раз подобные ассистенты являются массовыми продуктами, то должны обрабатывать информацию пользователя способом, наиболее простым для последнего. Такими способами для пользователя являются передача голосовых и текстовых команд на естественном языке, носителем которого он является.

Получается, чтобы быть успешной на рынке, система должна уметь обрабатывать сообщения на естественном языке, что является непростой исследовательской и инженерной задачей. Наиболее яркими примерами подобных систем являются Siri от Apple, Cortana от Microsoft, Google now, ask.com, IBM Watson и др.

На данный момент существует множество способов, с помощью которых достигается некоторое понимание пользовательских сообщений машиной. Эти способы грубо можно разделить на три группы:

- подходы, основанные на техниках information retrieval (IR based approach),
- подходы, основанные на обработке естественных языков и баз знаний (knowledge based approach),
- подходы, комбинирующие предыдущие две техники (и некоторые другие, например, deep machine learning).

В настоящий момент наибольшее распространение получили IR based системы, в силу многих причин: хорошее развитие аппарата математической статистики, внедрение mapreduce, и др. Но с другой стороны, более перспективным направлением является подход на основе баз знаний, который и будет далее рассматриваться в данной работе.

Важной подзадачей задачи трансформирования сообщений на естественном языке в сообщения, понимаемые машиной, в системах с базами знаний является задача отображения сообщения в логическую форму в некоторых условных обозначениях. Это непростая задача. Ее основными сложностями являются:

- сложная формализация задачи,
- плохо представленные данные для обучения,
- большие объемы вычислений.

Рассмотрим подробнее последний пункт. К сожалению, из-за особенностей естественных языков, а точнее в основном из-за присутствующих в них неоднозначностей, задача построения системы с открытой предметной областью (open-domain system), с некоторой относительно высокой точностью преобразующую сообщения на естественном языке в интерпретируемые машиной логические выражения, сильно усложняется. Поэтому строят системы, основанные на базах знаний лишь в некоторой предметной области (domain-closed system). Это одна из многих причин большого количества вычислений: модификацию, обучение и тестирование систем необходимо проводить многократно для различных предметных областей.

Наличие подобных сложностей обусловливает актуальность данной работы.

Постановка задачи

Предметом исследования данной работы является изучение и доработка фреймворка Cornell Semantic Parsing Framework, предназначенного для получения логических форм в виде лямбда-выражений из текста на естественном языке (английском).

Цель работы заключается в реализации более эффективного алгоритма машинного обучения на основе уже представленного в данном фреймворке. Эффективность алгоритма обучения достигается за счет горизонтального роста, т.е. распараллеливания на большее количество процессоров. Но кроме этого стоит отметить еще и качественный рост в реализации параллельной версии алгоритма обучения, который достигается благодаря использованию языка программирования Scala и фреймворка Akka. Благодаря акторной модели, реализованной во фреймворке Akka (впервые примененной в языке программирования Erlang), параллельная реализация алгоритма позволяет проводить "бесшовное" распараллеливание на множестве машин. А использование языка функционального программирования Scala позволяет застраховаться от ошибок в случае необходимости доработки фреймворка, благодаря множеству особенностей языка.

Результатом данной работы является доработанная версия Cornell Semantic Parsing Framework с параллельным алгоритмом обучения, предназначенная для запуска на нескольких машинах.

Обзор литературы

Стоит сказать, что задача обработки сообщений на естественном языке не нова, и раньше предпринимались попытки ее решить.

Одной из первых такой систем стала построенная в 1961 году система BASEBALL [1], предметной областью которой был бейсбол. Система могла отвечать на очень небольшое количество простых вопросов и имела очень небольшой лексикон. В ее основе лежал подход паттерн-матчинга (сопоставление с образцом) вопросов.

Минимальную классификацию вопросов может осуществлять система ELIZE [2], написанная в 1966 году. Система представляет собой диалогового робота, который симулирует психотерапевта. Симуляция заключается в том, что ELIZE выделяет слова-маркеры, и генеририрует уточняющие вопросы по шаблону.

Система LUNAR [3] была разработана в результате космической программы NASA в 1971 году. Ее предназначение заключалось в ответах на простые вопросы относительно объектов, собранных в космосе: например, "Сколько имеется камней круглой формы в наборе?". Точность ее ответов на вопросы составляла до 78%. Сисема состоит из нескольких компонентов: расширенной сети переходов общего пользования (general-purpose augmented transition network (ANT) [4]), фреймворка семантического представления синтаксической информации, фреймворка отображения естественного языка в логическую форму, словаря из 4500 слов и 13 000 сущностей в базе знаний.

Другими интересными примерами являются START [5] – первая IR-based система с открытой предметной областью с использованием web, а также SHRDLU [6], Murax [7], Jupiter [8].

Некоторые более современные системы (например, [9, 10, 11]) основа-

ны на статистическом подходе, суть которого заключается в использовании модификаций алгоритмов SVM, наивного Байесовского классификатора, принципа максимума энтропии и др.

В работах [12, 13] применяется сопоставление с образцом. Суть такого сопоставления заключается в нахождении некоторых шаблонов (паттернов) в сообщениях от пользователя и поиске подходящих под этот шаблон ответов в web.

Глава 1. Подходы к построению вопросно-ответных систем

1.1. Подход на основе техник Information Retrieval

Системы, построенные на основе техник и алгоритмов извлечения информации, в своей основе имеют достаточно сложную архитектуру. В этом параграфе рассмотрим наиболее общую архитектуру, присущую таким системам. Отдельные проекты QA систем строятся на модификациях рассматриваемой архитектуры. Более того, существует множество "state-of-art" концепций построения QA систем.

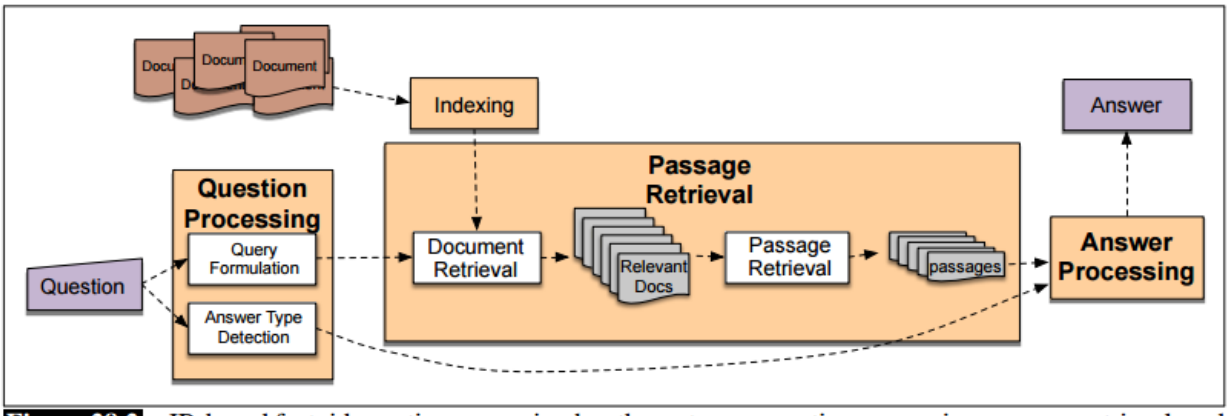

Figure 28.2 IR-based factoid question answering has three stages: question processing, passage retrieval, and answer processing.

Рис. 1: Обобщенный пример IR-based QA системы [21].

Целью этапа обработки вопроса (question processing) является получение некоторой информацию из текстового запроса для ее последующей обработки. Разделяется на два основных этапа: формирование запроса к поисковой системе (query formulation) и определение типа вопроса (answer type detection).

Результатом работы первого этапа являются ключевые слова, которые будут впоследствии обработаны IR системе с целью получения релевантных документов для исходного текстового запроса.

На этапе answer type detection определяется тип вопроса относительно предметной области, и на основе этой информации происходит классификация вопросов. Очевидно, что категорий вопросов может быть множество. На начальном этапе построения такой системы можно вручную задать начальный ограниченный набор категорий вопросов и некоторых соответствующих им сущностей, который с ростом системы необходимо будет расширять. Но обычно строится система классификации вопросов. Такие классификации могут быть построены полуавтоматически и динамически [14] из информационных систем или онтологий вроде WordNet [15].

На этапе формирования запроса необходимо составить список ключевых слов, которые будут отправлены в запросе к поисковой системе [16].

Стадия формирования пассажей (Passage Retrieval) состоит из нескольких частей:

- получение набора документов (web-страниц) из IR системы после отправления запроса,
- сегментирование документов в более короткие связанные текстовые отрывки (пассажи),
- ранжирование пассажей.

Ранжирование пассажей можно производить множеством способов. Можно учитывать следующие признаки, полученные из ответов поисковой системы:

- количество именованных сущностей правильного типа в пассаже,
- близость ключевых слов из исходного запроса друг к другу в текущем отрывке (например, в [17] предпочтение отдают менее длинным отрывкам с большим количеством ключевых слов),

- вычисление длины наибольшей последовательности слов из исходного вопроса в пассаже,

- ранг страницы, назначенный поисковой системой

- и др.

На этапе извлечения ответов (answer extraction) после ранжирования пассажей необходимо произвести тэгирование именованных сущностей. К примеру, если в процессе обработки текствого запроса система распознала тип вопроса как географический, то она должна уметь пользоваться обученным тэггером (named entity tagger), который может найти наименования географических объектов в пассаже.

Задача усложняется, когда пассаж содержит несколько кандидатов.
Вопрос: "Who was Queen Victoria's second son?"
Пассаж: "The Marine biscuit is named after Marie Alexandrovna, the daughter of Czar Alexander II of Russia and wife of Alfred, the second son of Queen Victoria and Prince Albert."
Правильный ответ: "Alfred."

В случае неопределенности и наличия нескольких возможных ответов в пассаже применяется машинное обучение. Возможные в таком случае признаки для обучения:

- соответствие типу вопроса (ответ-кандидат содержит фразу или слово с правильным типом ответа),

- степень соответствия регулярного выражения, полученного из ответа, с ответом-кандидатом,

- наличие и количество ключевых слов из вопроса в ответе-кандидате,

- расстояние между ключевыми словами (этот признак уже упоминался при выборе подходящих пассажей, но можно переиспользовать),
- фактор новизны (отсутствие ответа-кандидата в запросе),
- наличие и длина последовательности ключевых слов из запроса в ответе-кандидате,
- и др.

1.2. Подход на основе обработки естественных языков

Описание категориально-контекстных грамматик (в иностранной литературе — Combinatory Categorial Grammar, CCG) [18] опирается на несколько разделов современной математики. В основном это лямбда-исчисление, комбинаторная логика и логлинейный анализ.

1.2.1. Лямбда-исчисление в категориально-контекстных грамматиках.

Некоторую лингвистическую единицу (чаще всего, предложение) можно представить с помощью типизированного лямбда-исчисления в виде некоторого лямбда-выражения. Например, в качестве типов, используемых в исчислении, можно использовать следующие базовые типы:

- сущности (entities),
- вещественные числа (real values),
- булевы значения (truth values).

Комбинируя представленные типы, можно составлять так называемые стрелочные типы. Примером может являться тип $< e, t >$, который представляет собой функцию, отображающую сущности в вещественные

числа, то есть $e \to t$. В зависимости от конкретной предметной области (домена) тип сущности может быть расширен подтипами.

Лямбда-термы в [19] строятся с помощью некоторого расширения, а именно используются:

- константы
  - $Texas : e$,
  - $6 : r$ и др.;
- кванторы (существования, всеобщности);
- операторы логического связывания (конъюнкция, дизъюнкция, импликация, отрицание):
  - $\exists x.state(x) \wedge borders(x, Texas)$ – соответствует высказыванию: "Существует штат, граничащий со штатом Техас"
- дополнительные кванторы: $count$, $argmax$, $argmin$
  - $argmax(\lambda x.state(x), \lambda x.size(x))$ – соответствует высказыванию: "Наибольший по размеру штат";

Также λ-термы можно строить из других λ-термов без свободных переменных (комбинаторов).

1.2.2. Контекстно-категориальная грамматика, синтаксический тип.

Определение 1 *Формальная грамматика, или просто грамматика, в теории формальных языков — способ описания формального языка, то есть выделения некоторого подмножества из множества всех слов некоторого конечного алфавита.*

Формальные грамматики можно разделить на:

- порождающие – задают правила, используя которые возможно построить любой терм из лексикона;
- распознающие (аналитические) – задают правила, используя которые возможно определить вхождение терма в словарь (лексикон).

Категориальная грамматика — это формальная грамматика распознающего типа. В такой грамматике каждой синтаксической единице приписывается некоторое категориальное значение или тип. Различают два типа значений:

- фраза (группа, phrase):
  - именная группа (noun phrase),
  - глагольная группа (verb phrase),
  - и др.;
- предложение (sentence, S).

К примеру, в простейшем предложении “Utah borders Idaho” выделяются следующие категории:

- $Utah := NP$,
- $Idaho := NP$,
- $borders := (S/NP)\backslash NP$.

В более сложном примере возможно получить еще более сложную структуру. Понятно, что именные группы могут быть неограниченно сложными, то есть иметь тип $A\backslash B$. Где A и B могут быть как примитивами, так и более сложными типами. Также одному терму из лексикона может соответствовать более одного одного синтаксического типа. Помимо некоторого лексикона и синтаксических типов, в CCG необходимо определить

так называемые комбинаторные правила (combinatory rules). Такие правила описывают операции, которые разрешено проводить с синтаксическими типами. Простейшие примеры — это функциональные аппликаторы:

$$A/B \quad B \Rightarrow A$$

$$B \quad A\backslash B \Rightarrow A$$

Тип A/B соответствует строке типа A с отсутствующим типом B справа. Во время аппликации к строке типа A/B справа применяется строка типа B, что в итоге дает дополненную типом B строку типа A. Во втором примере, наоборот, $A\backslash B$ соответствует строке типа A с отсутствующим типом B слева. И аппликация применяется слева к строке типа $A\backslash B$. По сути, аппликация является заменой двух рядом расположенных типов на один сложный тип. Очевидно, что аппликация допустима, если один тип является сложным типом, а тип справа (слева) соответствует правому недостающему типу.

В [18] были введены и другие комбинаторные правила.

Определение 2 *Композиция — приписывание сложного типа цепочке из двух сложных типов.*

$$A/B \quad B/C \Rightarrow A/C$$

$$A/B \quad B\backslash C \Rightarrow A\backslash C$$

$$B/C \quad A\backslash B \Rightarrow A/C$$

$$B/C \quad A/B \Rightarrow A\backslash C$$

Определение 3 *Подъем типа — присвоение единице, рядом с которой находится способное присоединить её выражение, типа, позволяющего ей самой присоединить это выражение.*

$$A/B \quad B \Rightarrow A/B \quad A\backslash(A/B)$$

$$B \quad A/B \Rightarrow A/(A\backslash B) \quad A/B$$

Предыдущие комбинаторные операции были бинарными, эта же операция является унарной.

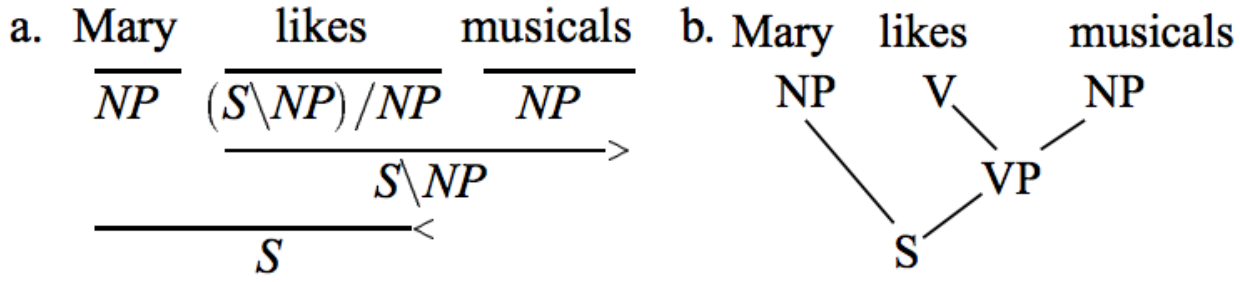

Рис. 2: Простой пример применения аппликации из [20].

1.2.3. Контекстно-категориальная грамматика, семантический тип.

Теперь осталось объединить лямбда-исчисления и комбинаторные правила для синтаксических типов с целью получить семантические типы, используемые в CCG. Рассмотренный ранее пример “Utah borders Idaho” с добавленными семантическими типами будет выглядеть следующим образом.

- $Utah := NP : utah$
- $Idaho := NP : idaho$
- $borders := (S\backslash NP)/NP : \lambda x.\lambda y.borders(x, y)$

Здесь использована нотация “элемент словаря := синтаксическая категория : семантическая категория”.

Запишем рассмотренные ранее комбинаторные правила, расширенные семантическими типами.

Аппликация:

$$A/B : f \quad B : g \Rightarrow A : f(g)$$

$$B : f \quad A\backslash B : g \Rightarrow A : f(g)$$

Композиция:

$$A/B : f \quad B/C : g \Rightarrow A/C : \lambda x.f(g(x))$$

$$B\backslash C : f \quad A\backslash B : g \Rightarrow A\backslash C : \lambda x.f(g(x))$$

Подъем типа:

$$ADJ : x.g(x) \Rightarrow N/N : \lambda f.\lambda x.f(x)^g(x)$$

$$PP : x.g(x) \Rightarrow N\backslash N : \lambda f.\lambda x.f(x)^g(x)$$

$$AP : x.g(x) \Rightarrow S\backslash S : \lambda f.\lambda x.f(x)^g(x)$$

$$PP : x.g(x) \Rightarrow S/S : \lambda f.\lambda x.f(x)^g(x)$$

$$NP : f \Rightarrow S/(S\backslash NP) : \lambda g.g(f)$$

Определив таким образом некоторый набор комбинаторных правил, становится возможным использовать их для вывода семантического типа предложения.

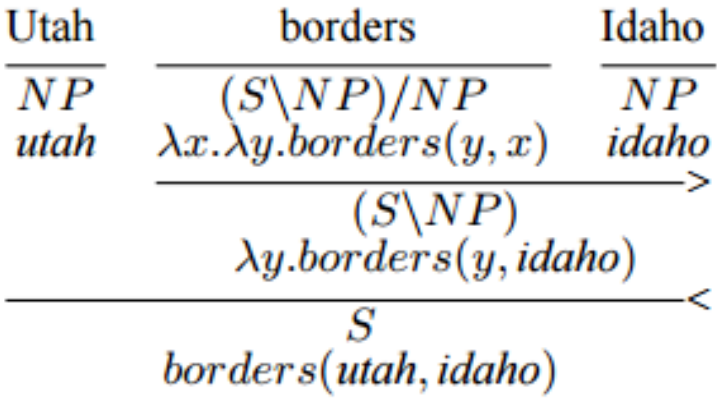

Рис. 3: Пример использования комбинаторных правил в выводе семантического типа предложения.

Стоит отметить, что в различных реализациях CCG используются различные наборы комбинаторных правил. Например, в [20] определены сразу несколько операций подъема типа (forward и backward type raising).

1.2.4. Обощенный лексикон.

Лексикон в CCG — это ключевой компонент, использование которого позволяет сопоставить слова из естественного языка и фразы, построенные из них, с CCG-категориями. Очевидно, что из-за использования различных комбинаторных правил возможны случаи, когда одному слову или фразе ставятся в соответствие более одной CCG-категории. Это усложняет задачу обучения: в процессе обучения необходимо рассматривать все возможные категории для определенного слова или фразы. Однако, несложно заметить некоторые закономерности в построении фраз и предложений на естественном языке. Это наблюдение позволяет ввести так называемое обобщенное представление CCG (factorized representation CCG).

Рассмотрим в качестве примера несколько схожих фраз на естественном языке и их CCG-категории.

Таблица 1: Представление CCG типов

Словосочетание	**Логическая форма**	**Элементы лексикона**
the house dog	$ix.dog(x) \wedge of(x, iy.house(y))$	$house \vdash ADJ : \lambda x.of(x, iy.house(y))$
the dog of the house	$ix.dog(x) \wedge of(x, iy.house(y))$	$house \vdash N : \lambda x.house(x)$
the garden dog	$ix.dog(x) \wedge of(x, iy.garden(y))$	$garden \vdash ADJ : ix.dog(x)of(x, iy.garden(y))$

В таблице (1), представленной выше, кратко описаны нормативы, наиболее важные с точки зрения анализа данных, необходимого для решения задач текущей работы.

Легко заметить, что элементы словаря из первого и второго словосочетаний похожи тем, что используют одну и ту же языковую константу “house”, но отличаются своей структурой. В то время как элементы словаря из первого и третьего словосочетаний похожи своей структурой с точностью до языковой константы.

Очевидно, что для того чтобы представить элементы словаря в более компактном виде, необходимо произвести их декомпозицию на лексемы и шаблоны. Лексемами будем называть языковые константы, а шаблонами будем называть структуры данных, с помощью которых возможно выразить некоторую структуру схожих элементов лексикона, а также вариативность входящих в их состав языковых констант.

Для того чтобы представить рассмотренные ранее словосочетания, можно использовать соответствующие шаблоны

$$\lambda(\omega, \{\nu_i\}_1^n).[\omega \vdash ADJ : \lambda x.of(x, iy.\nu_1(y))],$$

$$\lambda(\omega, \{\nu_i\}_1^n).[\omega \vdash N : \lambda x.\nu_1(x)]$$

и лексемы

$$(garden, \{garden\}),$$

$$(house, \{house\}).$$

В представлении лексем используются элементы естественного языка (первый элемент), а также набор логических констант (перечисление в фигурных скобках).

Помимо того, что таким образом можно достаточно компактно представить довольно большой лексикон, еще одним преимуществом использования обобщенного представления лексикона является то, что, используя декомпозицию его элементов, можно проводить модификации процесса обучения. Такой подход позволяет гибко определять функции призна-

ков: к примеру, становится возможным определять набор дополнительных признаков, которые необходимо будет вычислить, при использовании определенных языковых констант, шаблонов или их комбинаций. Кроме того, возможна частичная или полная шаблонизация элементов словаря.

Элемент лексикона: $house \vdash ADJ : \lambda x.of(x, iy.house(y))$

Таблица 2: Шаблонизация лексикона в CCG

Степень шаблонизации	**Представление лексикона**
Частичная	$(house, \{house\})$, $\lambda(\omega, \{\nu_i\}_1^n).[\omega \vdash ADJ : \lambda x.of(x, iy.\nu_1(y))]$
Частичная	$(house, \{of\})$, $\lambda(\omega, \{\nu_i\}_1^n).[\omega \vdash ADJ : \lambda x.of(x, iy.house(y))]$,
Полная	$(house, \{of, house\})$, $\lambda(\omega, \{\nu_i\}_1^n).[\omega \vdash ADJ : \lambda\nu_1.of(x, iy.\nu_2(y))]$

1.2.5. Обучение.

Целью обучения является обработка подготовленных данных некоторым алгоритмом, на выходе дающим некоторую комбинаторно-категориальную грамматику. Для того, чтобы провести процесс обучения, необходимо ответить на следующие вопросы: какие данные необходимо использовать, и чему должна научиться построенная модель с помощью алгоритма обучения.

Основными двумя компонентами CCG являются некоторый лексикон и некоторые заранее определенные комбинаторные функции. В отличие от комбинаторных функций, лексикон, ставящий словам и фразам из естественного языка некоторые CCG-категории, – это то, чему необходимо обучить алгоритм. Задачу усложняет то, что дерево вывода логической формы предложения неоднозначно. Поэтому необходимо провести обучения таким образом, чтобы выбиралось наиболее подходящее дерево вывода логической формы, а вместе с ним и сама логическая форма. Для этого в

свою очередь необходимо оценить параметры парсинга.

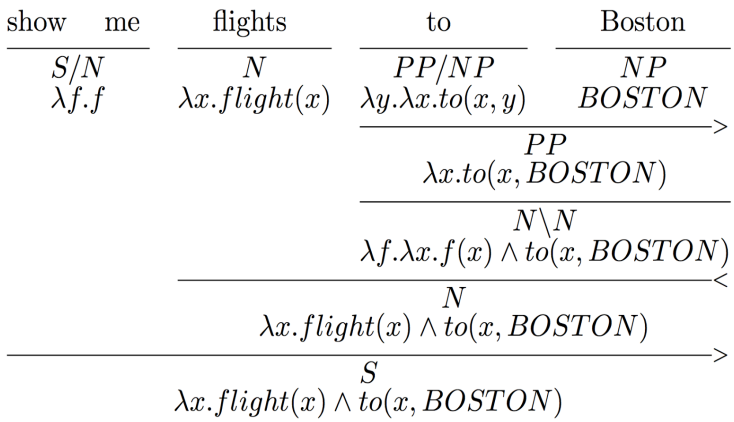

Рис. 4: Пример построения дерева логической формы [19].

Таким образом, отвечая на вопрос, чему должна научиться построенная модель, необходимо ответить на два вопроса:

1. Каковы параметры дерева вывода логической формы (парсинга)?

2. Каков лексикон?

Одним из самых простых вариантов обучения является обучение с учителем, для чего в качестве обучающих данных необходимо использовать размеченный набор данных, то есть предложения и поставленные им в соответствие логические формы.

Во время обучения необходимо рассмотреть все возможные варианты деревьев разбора, полученных из текущего предложения и заданных комбинаторных правил. Например, из вопросительного предложения “What is the largest state that borders Texas?” возможно вывести следующие логические формы, полученные, конечно же, с помощью различных деревьев разбора:

$$argmax(\lambda x.state(x) \wedge borders(x, Texas), \lambda y.size(y)),$$

$$argmax(\lambda x.river(x) \wedge in(x, Texas), \lambda y.size(y)).$$

Полученные логические формы можно интерпретировать как запросы к некоторой базе данных, которые можно выполнить. После получения возможных подобных логических форм происходит сравнение их и ожидаемого ответа из обучающей выборки, а затем и обновление параметров модели. Такой подход позволяет во время обучения выбрать необходимое единственное правильное дерево разбора. В противном случае необходимо во время обучения предоставить все возможные деревья разборов для каждого предложения из обучающей выборки. Описанный выше подход применяется в [19].

Рассмотрим алгоритм обучения персептрона. Пусть дана обучающая выборка $\{(x_i,\ y_i),\ i = 1,\ ...,\ n\}$, необходимо выполнить:

$for\ t = 1...T:$

$\quad for\ i = 1...n:$

$\quad\quad y^* \leftarrow argmax_y(\Theta; \Phi(x_i, y))$

$\quad\quad if\ y_i \neq y^*$

$\quad\quad\quad \Theta \leftarrow \Theta + \Phi(x_i, y_i) - \Phi(x_i, y^*)$

В первой строке задается количество эпох обучения T. В каждой из эпох для каждой пары из обучающей выборки вычисляется предсказание модели, после чего сравнивается фактическое значение y^* с ожидаемым y_i. Если они отличаются на некоторую достаточно большую величину, происходит обновление модели.

Персептрон можно рассмотреть как лог-линейную модель.

$$p(y|x) = \frac{exp(wf(x, y))}{\sum_{y_i} exp(wf(x, y_i))}$$

В таком случае корректировка весов приозводится следующим образом:

$$update_i = f(x_i, y_i) - f(x_i, y), \quad y = argmax_y(wf(x_i, y))$$

Продифереициируем логарифм правдоподобия:

$$update = \sum_i f(x_i, y_i) - E_{p(y|x_i)} f(x_i, y)$$

$$update_i = f(x_i, y_i) - E_{p(y|x_i)} f(x_i, y)$$

Заменим математическое ожидание в правой части на аппроксимацию из алгоритма Витерби, то есть

$$E_{p(y|x_i)} = argmax_y w$$

В итоге получим соотношение для обновления весов персептрона.

Теперь, зная вывод обновления весов персептрона из логлинейной модели, покажем как добавлять скрытые переменные персептрону. Для этого добавим скрытую переменную h в лог-линейную модель.

$$p(y|x) = \sum_h p(y, h|x)$$

$$p(y|x) = \frac{exp(wf(x, h, y))}{\sum_{y_i} exp(wf(x, h_i, y_i))}$$

Продифференцируем маргинал, чтобы максимизировать логарифм правдоподобия:

$$update = \sum_i E_{p(h|y_i, x_i}[f(x_i, h, y_i)] - E_{p(y,h|x_i)} f(x_i, h, y)$$

Итогом является очень похожее на полученное ранее выражение, с той лишь разницей, что в последнем участвуют два математических ожидания. Получив выражение для корректировки весов и рассуждая по аналогии с предыдущим выводом, получаем:

$$update_i = E_{p(h|y_i, x_i}[f(x_i, h, y_i)] - E_{p(y,h|x_i)} f(x_i, h, y)$$

Снова заменим математическое ожидание на аппроксимацию из аглоритма Витерби:

$$update_i = f(x_i, h', y_i) - f(x_i, h^*, y^*)$$

$$y^*, h^* = argmax_{y,h}(wf(x_i, h, y))$$

$$h' = argmax_h(wf(x_i, h, y_i))$$

Теперь становится возможным описать алгоритм обучения персептрона со скрытыми переменными:

$for\ t = 1...T:$

$\quad for\ i = 1...n:$

$\quad\quad y^*, h^* \leftarrow argmax_{y,h}(\Theta; \Phi(x_i, y))$

$\quad\quad if\ y_i \neq y^*$

$\quad\quad\quad h^* \leftarrow argmax_h(\Theta, \Phi(x_i, h, y_i))$

$\quad\quad\quad \Theta \leftarrow \Theta + \Phi(x_i, h', y_i) - \Phi(x_i, h^*, y^*)$

Алгоритм отличается от описанного ранее тем, что для вычисления обновления весов сначала вычисляется наиболее вероятная скрытая переменная.

Использование скрытых переменных в персептроне имеет некоторые особенности, среди которых:

- нет гарантированной сходимости;

  в случае обычного персептрона оптимизация весов рассматривается на многомерной выпуклой функции, в случае же лог-линейной модели выпуклость не гарантируется;

- простота и легкость реализации;

  хорошо работает с правильно выбранными начальными весами;

- возможно реализовать модификации в задаче семантического парсинга;

  например, становится возможным использование множества скрытых переменных, соответствующих множеству типов данных;

1.2.6. Обобщенный алгоритм обучения.

В данной работе используется обобщенный алгоритм обучения, описанный в [19]. Его важной частью является выбор функции проверки достоверности (validation function, функции валидации), принимающей на вход дерево вывода логической формы и возвращающей булевское значение – индикатор достоверности дерева вывода логической формы, а также степень достоверности разбора. Другими словами функция валидации – это некоторая функция : $\nu : y \rightarrow \{t, f\}$, где y – дерево разбора, $t \in \{true,\ false\}$, $f \in R$.

Второй важной частью алгоритма обучения является функция генерации лексикона (Lexical Generation Procedure, Genlex). Генлекс принимает на вход некоторое предложение x, функцию валидации ν, и текущий лексикон Λ, а также некоторый параметр Θ, по которому необходимо произвести оптимизацию. Результатом работы генлекс-функции является набор всевозможных элементов лексикона, который впоследствии фильтруется, и из которого выбираются необходимые для корректировки весов элементы лексикона и деревья разбора.

Схема обобщенного алгоритма выглядит следующим образом. Дан тренировочный набор $(x_i, \nu_i) : i = 1...n$, где x_i – предложение, ν_i – функция валидации.

Задаются начальные веса Θ и начальный лексикон Λ_0, $\Lambda \leftarrow \Lambda_0$.

$for\ t = 1\ ...\ T,\ i = 1\ ...\ n,$

T – количество эпох, n – размер обучающей выборки;

Шаг генерации лексикона.

$\lambda_G \leftarrow GENLEX(x_i, \nu_i, \Lambda, \Theta)$,

x_i – предложение, ν_i – функция валидации, Λ – лексикон, Θ – веса

$\lambda \leftarrow \Lambda \cup \lambda_G$

Пусть Y – наилучшие разборы (k штук), сгенерированные $GEN(x_i, \lambda)$

Выберем некоторые из них для корректировки весов:

$MAXV_i(Y, \Theta) = \{y \mid y \in Y \wedge \nu_i(y) \wedge \forall y' \in Y \wedge \nu_i(y') \Rightarrow \langle \Theta, \Phi_i(y') \rangle \leq \langle \Theta, \Phi_i(y) \rangle\}$

$\lambda_i \leftarrow \cup_{y \in MAXV_i(Y,\Theta)} LEX(y)$

Обновляем лексикон $\Lambda \leftarrow \Lambda \cup \lambda_i$.

Шаг корректировки весов.

Разделяем все разборы на два условных множества “хороших” (удовлетворяющих функции валидации) и “плохих” (неудовлетво - ряющих) разборов.

$G_i \leftarrow MAXV_i(GEN(x_i, \Lambda); \Theta)$ – "хорошие"разборы,

$B_i \leftarrow \{e \mid e \in GEN(x_i, \Lambda) \wedge \neg\nu_i(y)\}$ – "плохие"разборы,

$\Delta_i(y, y') = |\Phi_i(y') - \Phi_i(y)|_1$

$R_i \leftarrow \{g \mid g \in G_i \wedge \exists b \in B_i : \langle \Theta, \Phi_i(g) - \Phi_i(b) \rangle \leq \gamma \Delta_i(g, b)\}$

$E_i \leftarrow \{b \mid b \in B_i \wedge \exists g \in G_i : \langle \Theta, \Phi_i(g) - \Phi_i(b) \rangle \leq \gamma \Delta_i(g, b)\}$

$\Theta \leftarrow \Theta + 1/|R_i| + \sum_{r \in R_i} \Phi_i(r) - \sum_{e \in E_i} \Phi_i(e)$

Выход Θ, Λ.

Глава 2. Программный комплекс для построения вопросно-ответных систем.

2.1. Cornell Semantic Parsing Framework.

Cornell Semantic Parsing Framework (SPF) — фреймворк, предназначенный для обучения и вывода логических форм в виде лямбда-выражений из предложений на естественном языке [22]. Фреймворк реализует CCG [18]. Написан на языке программирования Java и распространяется под лицензией GNU. На данный момент содержит $\approx$ 25 тысяч строк кода. Код и некоторая документация доступны бесплатно в репозитории [23].

В качестве данных для обучения в проекте используется корпус GeoQuery [24]. Каждая директория в репозитории — это отдельный Eclipse проект. Фреймворк собирается системой сборки ant в jar-артефакт.

2.2. Фреймворк Akka.

Akka — набор библиотек с открытым исходным кодом, написанный на языке программирования Scala, предназначенный для создания масштабируемых, устойчивых систем, работающих на JVM [25]. При разработке приложений Акка позволяет сфокусироваться на написании бизнес-логики, а не низкоуровнего кода, для обеспечения надежного поведения, отказоустойчивости и высокой производительности.

Ключевыми моментами приложений, разработанных с помощью Akka, являются:

- ассинхронный параллелизм (достигается в силу того, что Akka реализует модель акторов), основанный на доставке сообщений. Таким образом, не используются разделяемые мутабельные данные и при-

митивы синхронизации, что делает логику приложений прозрачной и понятной;

- взаимодействие акторов друг с другом посредством доставки сообщений, при этом бесшовно, т.е. с помощью одного и того же интерфейса, независимо от того, на разных JVM они работают или нет;
- иерархическая структура акторов, что позволяет прозрачно обрабатывать сбои приложения с помощью супервизии акторов (т.е. контроля родительского актора).

2.2.1. Акторная модель вычислений.

Рассматриваемая модель вычислений была предложена Карлом Хьюиттом в [26]. Главное свойство акторов заключается в том, что с помощью них становится возможным моделировать различные объекты как объекты с некоторым состоянием, обменивающиеся друг с другом сообщениями. В то же время, акторы являются вычислительными субъектами.

В качестве вычислительных субъектов акторы обладают следующими свойствами:

- взаимодействие с другими акторами посредством отправки сообщений вместо вызова методов,
- управление своим собственным состоянием,
- при ответе на сообщение актор может:
  - создавать дочерних акторов,
  - отправлять сообщения другим акторам,
  - завершать выполнение своего потока или потока дочернего актора.

Вместо вызова методов акторы отправляют друг другу сообщения. Но отправка сообщения не передает поток исполнения от отправителя к получателю, так что актор может отправить сообщение, а его поток будет продолжать выполняться. Поэтому актор может выполнять больше работы.

Когда метод возвращает некоторый результат, он освобождает управление своим выполняющимся потоком. С этой точки зрения, акторы ведут себя как обычные объекты в ООП: они реагируют на сообщения и возвращают потом исполнения, когда заканчивают обработку текущего сообщения.

Важное отличие передачи сообщения от вызова метода заключается в том, что сообщения не имеют возвращаемого значения. Посылая сообщение, актор делегирует работу другому актору. Если ожидать возвращаемое значение, то передающий актор должен либо заблокировать свой поток выполнения, либо выполнить работу другого актора в том же потоке. Вместо этого актор-получатель просто возвращет значение в ответном сообщении.

Акторы реагируют на сообщения так же, как объекты реагируют на методы, которые вызываются на них. Еще одно отличие от обычных объектов в многопоточном приложении заключается в том, что вместо многочисленных потоков, вызывающих объект и конкурирующих за его ресурс, акторы обрабатывают сообщения независимо от его отправителей и реагируют на входящие сообщения последовательно, по одному за раз. В то время как актор обрабатывает сообщения, посланные ему последовательно, другие акторы работают одновременно друг с другом, так что акторная система может обрабатывать столько сообщений одновременно, сколько ядер процессора имеется на машине. Так как одним актором всегда обрабатывается не более одного сообщения, состояния актора могут хранится без

синхронизации. Это происходит автоматически, без использования блокировок:

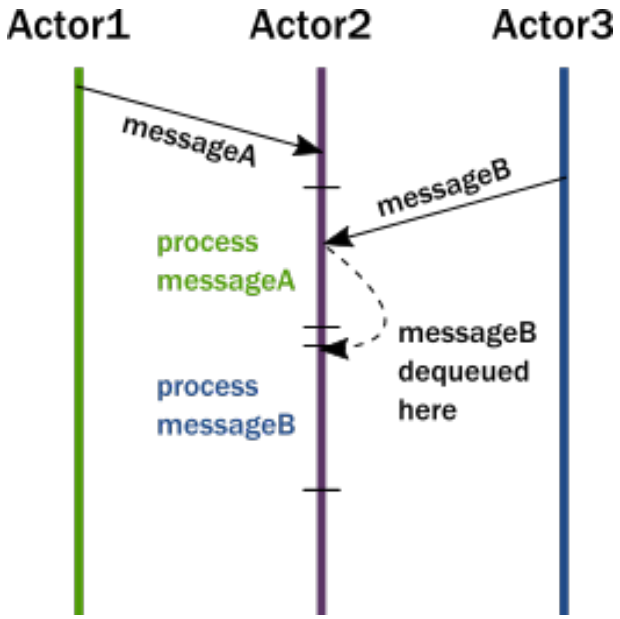

Рис. 5: Схема процесса асинхронного обмена сообщениями сообщениями между акторами [25].

Подводя итог описанию акторной системы, дадим краткое описание процесса асинхронного обмена сообщениями внутри акторной системы.

1. Полученное сообщение добавляется в конец очереди.
2. Если актор не назначен для выполнения задачи, то он помечается как готовый к выполнению.
3. Планировщик принимает актора и начинает исполнять запланированную задачу.
4. Актор получает сообщения из начала очереди.
5. Актор изменяет свое внутренее состояние и отправляет сообщения другим акторам.
6. Актор помечатеся как неназначенный на исполнение задач.

Для реализация подобного рода поведения необходимо задать:

1. почтовый ящик (очередь, куда поступают сообщения),
2. поведение актора (его состоние, внутрение переменные),
3. сообщения (сигналы, похожие на вызовы методов, имеющие некоторые аргументы),
4. среду исполнения (механизм, умеющий оперировать акторами, а точнее вызывать код обработки сообщений),
5. адрес актора.

2.2.2.Библиотеки и модули фреймворка Akka.

Фреймворк Akka состоит из нескольких модулей:

- Actors (core),
- Remoting,
- Cluster,
- Cluster Sharding,
- Cluster Singleton,
- Cluster Publish –Subscribe,
- Persistence,
- Distributed Data,
- Streams,
- HTTP.

Так как в данной работе использовался модуль akka–remote, дадим краткое его описание. Akka–remote позволяет акторам взаимодействовать друг с другом удаленно, т.е. обмениваться сообщениями, находясь на разных машинах. Действительно же достаточно того, чтобы две акторные системы находились в разных JVM, не обязательно при этом на разных машинах. Akka–remote распространяется как jar-артефакт и предоставляет совсем небольшой API, благодаря которому удаленная и локальная отправка сообщений выглядят идентично. Так же, akka–remote используется как фундамент для построения более высокоуровневых akka–cluster, akka–cluster–sharding и др.

Задачи, решаемые akka–remote:

- адресация удаленных акторных систем, располагающихся в других JVM,
- адресация отдельных акторов, располагающихся в удаленных акторных системах,
- сериализация объектов для передачи по сети,
- управление низкоуровневыми сетевыми подключениями (и переподключениями) между удаленными jvm для обнаружения вышедших из строя акторных систем для поддержки прозрачности,
- управление мультиплексированием сообщений от несвязанного набора акторов, использующих одно и то же сетевое подключение, опять же, для поддержки прозрачности.

2.2.3.Пример обработки сообщений актором во вреймворке Akka.

Самый простой способ реализовать и наблюдать обмен сообщениями между акторами — создать несколько экземпляров ActorRef и задать пове-

дение вывода сообщения в консоль. В этом небольшом примере происходит вывод ссылки актора в консоль, который создается через вызов метода акторной системы system.actorOf(), а затем ссылки дочернего актора.

Актор, созданный с помощью вызова system.actorOf(), называется актором верхнего уровня, хотя на практике такие акторы не находятся на вершине всей иерархии, а только на вершине иерархии, определенной пользователем.

Создание актора верхнего уровня возможно с использованием любого актора, с помощью вызова метода context.actorOf(), который имеет точно такую же сигнатуру, что и создающая акторная система. На практике это выглядит следующим образом:

```
class PrintMyActorRefActor extends Actor {
  override def receive: Receive = {
    case "printit" =>
      val secondRef = context.actorOf(Props.empty,
                                      "second-actor")
      println(s"Second: $secondRef")
  }
}

val firstRef = system.actorOf(Props[PrintMyActorRefActor],
                              "first-actor")
println(s"First: $firstRef")
firstRef ! "printit"
```

Глава 3. Разработка распределенного алгоритма обучения SPF.

3.1. Схема параллельного алгоритма обучения.

Кратко напомним схему обобщенного алгоритма обучения. Дан тренировочный набор $(x_i, \nu_i) : i = 1...n$, где x_i – предложение, ν_i – функция валидации.

Задаются начальные веса Θ и начальный лексикон Λ_0, $\Lambda \leftarrow \Lambda_0$.

$for\ t = 1 \dots T,\ i = 1 \dots n,$

T – количество эпох, n – размер обучающей выборки;

Шаг генерации лексикона.

$\lambda \leftarrow \Lambda \cup GENLEX(x_i, \nu_i, \Lambda, \Theta)$

$MAXV_i(Y, \Theta) = \{y \mid y \in Y \wedge \nu_i(y) \wedge \forall y' \in Y \wedge \nu_i(y') \Rightarrow \langle \Theta, \Phi_i(y') \rangle \leq \langle \Theta, \Phi_i(y) \rangle\}$

$\lambda_i \leftarrow \cup_{y \in MAXV_i(Y,\Theta)} LEX(y)$

$\Lambda \leftarrow \Lambda \cup \lambda_i.$

Шаг корректировки весов.

$G_i \leftarrow MAXV_i(GEN(x_i, \Lambda); \Theta)$

$B_i \leftarrow \{e \mid e \in GEN(x_i, \Lambda) \wedge \neg\nu_i(y)\}$

$\Delta_i(y, y') = |\Phi_i(y') - \Phi_i(y)|_1$

$R_i \leftarrow \{g \mid g \in G_i \wedge \exists b \in B_i : \langle \Theta, \Phi_i(g) - \Phi_i(b) \rangle \leq \gamma \Delta_i(g, b)\}$

$E_i \leftarrow \{b \mid b \in B_i \wedge \exists g \in G_i : \langle \Theta, \Phi_i(g) - \Phi_i(b) \rangle \leq \gamma \Delta_i(g, b)\}$

$\Theta \leftarrow \Theta + 1/|R_i| + \sum_{r \in R_i} \Phi_i(r) - \sum_{e \in E_i} \Phi_i(e)$

Выход Θ, Λ.

На шаге генерации лексикона возможно параллельно обрабатывать предложения x_i с функицями валидации ν_i. Для этого надо составить наборы данных и отправить их на машины для параллельной обработки. Затем

данные с машин неоходимо получить и собирать вместе для последующего выбора лучших разборов. При сборе данных, помимо их агрегации, также необходимо произвести их фильтрацию для того, чтобы исключить дубликаты, т.к. в результате генерации лексикона из одного предложения можно получить множество потенциальных элементов лексикона.

В итоге распараллеленный шаг генерации лексикона выглядит следующим образом:

- формирование наборов,
- отправка наборов на машины,
- получение наборов обработанных данных,
- фильтрация (удаление повторящихся данных),
- выбор наилучших разборов и корректировка весов.

Рассмотрим подробнее шаг корректировки весов. В его самом начале происходит разделение разборов на два множества: удовлетворяющих функции валидации и неудовлетворяющих.

Очевидной возможностью для распараллеливания этого шага является разделение разборов на нескольких машинах. Напомним, что на прошлом шаге разборы и лексикон, был отфильтрован. Теперь по аналогии с предыдущим шагом, необходимо сформировать наборы данных, которые будут отправлены для вычисления на другие машины. Достигается это очень легко, так как данные для вычислений не пересекаются между собой. Отправляем отфильтрованные данные λ_i на машины. В результате с каждой машины возвращаем ответ $G_{i,k}, B_{i,k}$, где k - номер машины.

После этого аналогичным способом можно найти и коэффициенты для корректировки весов $|R_i|$ и $|E_i|$ Однако для этого, необходимо провести еще одну итерацию в процессе обучения для получения значений $\Phi_i(g)$

В итоге получаем параллельную версию алгоритма во многом напоминающую модель вычислений mapreduce.

Λ_0, $\Lambda \leftarrow \Lambda_0$.

$for\ t = 1 \ldots T,\ i = 1 \ldots n,$

T – количество эпох, n – размер обучающей выборки;

Шаг генерации лексикона.

$for\ k = 1 \ldots K$, где K - количество машин.

$\lambda \leftarrow \Lambda \cup GENLEX(x_{i,k}, \nu_i, \Lambda, \Theta)$

$MAXV_{i,k}(Y, \Theta) = \{y \mid y \in Y \wedge \nu_i(y) \wedge \forall y' \in Y \wedge \nu_i(y') \Rightarrow$

$\langle \Theta, \Phi_i(y') \rangle \leq \langle \Theta, \Phi_i(y) \rangle\}$

$\lambda_{i,k} \leftarrow \cup_{y \in MAXV_{i,k}(Y,\Theta)} LEX(y)$

$\lambda_i \leftarrow \cap_{k=1}^{K} \lambda_{i,k}$

$\Lambda \leftarrow \Lambda \cup \lambda_i.$

Шаг корректировки весов.

$for\ k = 1 \ldots K$

$G_{i,k} \leftarrow MAXV_i(GEN(x_i, \Lambda); \Theta)$

$B_{i,k} \leftarrow \{e \mid e \in GEN(x_i, \Lambda) \wedge \neg \nu_i(y)\}$

$B_i \leftarrow \cap_{k=1}^{K} B_{i,k}$

$G_i \leftarrow \cap_{k=1}^{K} G_{i,k}$

$\Delta_i(y, y') = |\Phi_i(y') - \Phi_i(y)|_1$

$for\ k = 1 \ldots K$

$R_{i,k} \leftarrow \{g \mid g \in G_{i,k} \wedge \exists b \in B_{i,k} : \langle \Theta, \Phi_i(g) - \Phi_i(b) \rangle \leq \gamma \Delta_i(g, b)\}$

$E_{i,k} \leftarrow \{b \mid b \in B_{i,k} \wedge \exists g \in G_{i,k} : \langle \Theta, \Phi_i(g) - \Phi_i(b) \rangle \leq \gamma \Delta_i(g, b)\}$

$R_i \leftarrow \cap_{k=1}^{K} R_{i,k}$

$E_i \leftarrow \cap_{k=1}^{K} E_{i,k}$

$\Theta \leftarrow \Theta + 1/|R_i| + \sum_{r \in R_i} \Phi_i(r) - \sum_{e \in E_i} \Phi_i(e)$

Выход Θ, Λ.

3.2. Реализация параллельного алгоритма обучения.

Первый этап в реализации параллельного алгоритма — использование другого сборщика проектов. Так как работать предстояло с языком программирования Scala и фреймворком Akka, был выбран sbt. Проект мигрировал с ant на sbt.

Язык программирования Scala предоставляет инструменты для асинхронных вычислений by design. Таким инструментом является класс Future. Future представляет собой удобную абстракицию контейнера асинхронных вычислений. Рассмотрим простой пример.

```
val longComputations = Future {
  computations goes here
}
```

Подождать вычисление объекта класса Future можно с помощью объекта класса Await.

```
val veryLongComputations = Future {
  computations goes here
}

val result = Await.result(veryLongComputations, 10.seconds)
```

Полезной особенностью яызка Scala является for-comprehensions, что является синтаксическим сахаром над вызовом методов map и flatMap. Поэтому вместо

```
val futureA = Future(...)
val futureB = Future(...)
val futureC = Future(...)
```

```
val futureD = futureA.flatMap {a =>
        futureB.flatMap { b =>
            futureC.map { c =>
            ...
            }
        }
    }
val result = Await.result(futureD, 10.seconds)
```

можно использовать более понятный код:

```
val futureD = for {
    futureA <- Future(...)
    futureB <- Future(...)
    futureC <- Future(...)
    result = ...
} yield result

val result = Await.result(futureD, 10.seconds)
```

Использование Future во многом бы упростило реализацию. Например, следующая реализация довольно проста и понятна:

```
class ComputationsSender extends Actor {
  var computations: Future = _

  override def receive: Receive = {
    case Compute => computations = Future( ... )
    case Send => sender ! computations
  }
```

```
}
```

В этом примере создается некоторый кусок вычислений внутри Future, а затем отправляется актору на другой машине. Но, к сожалению, объект класса Future не является сериализуемым.

Для обхода этого недостатка в распределенной реализации алгоритма необходимо посылать сообщения-маркеры, а не куски вычислений. Пример:

```
class ComputationsSender extends Actor {
  case class Computation(data: Data, kind: Kind)
  override def receive: Receive = {
    case Send => sender ! Computations(data, kind)
  }
}
```

3.3. Сериализация в параллельном алгоритме обучения.

В связи с этим, важной частью реализации распределенного алгоритма обучения был выбор сериализатора. К сожалению, стандартный сериализатор java.io.Serializable (интерфейс-маркер) из стандартной библиотеки языка java не подходил для этой задачи в силу его небольшой производительности.

Akka позволяет гибко настраивать сериализаторы. Достаточно добавить в конфигурационный файл:

```
akka {
  actor {
    serializers {
      java = "akka.serialization.JavaSerializer"
      proto = "akka.remote.serialization.ProtobufSerializer"
```

```
        myown = "docs.serialization.MyOwnSerializer"
      }
    }
}
```

Некоторые бенчмарки сравнения java-сериализаторов можно найти в [27]. Отметим, что на основании упомянутого сравнения был выбран сериализатор kryo [28].

3.4. Бенчмарки алгоритма.

Бенчмарки алгоритмов показали следующие результаты:

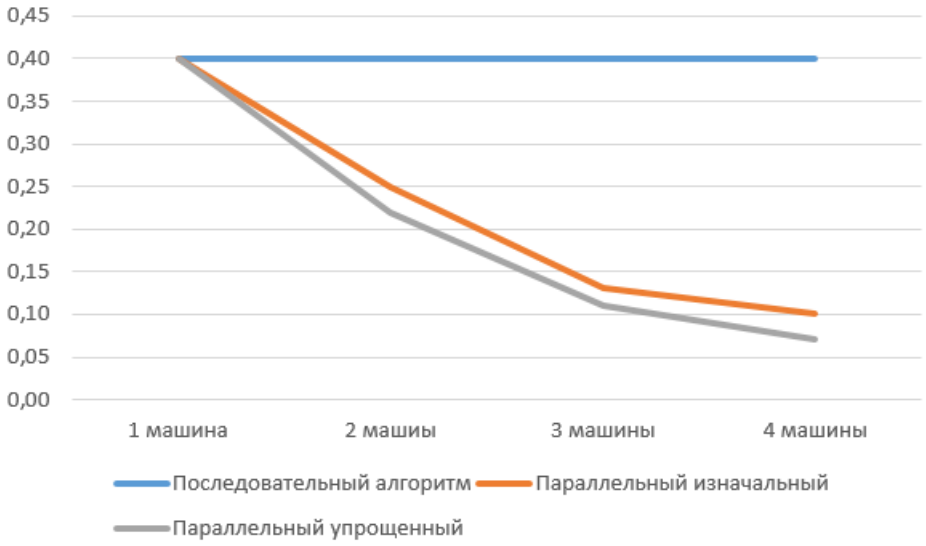

Рис. 6: Тест производительности: время обучения.

После чего было принято решение не считать коэффициенты $|E_{i,k}|$ и $|R_{i,k}|$ распредленным образом. Действительно, для нахождения этих коэффициентов на нескольких машинах, необходимо получить $|B_{i,k}|$ и $|B_{i,k}|$ и перемешав их, заново отправить на машины для дальнейших вычислений, потому что на основе этих множеств находятся необходимые мощности множеств.

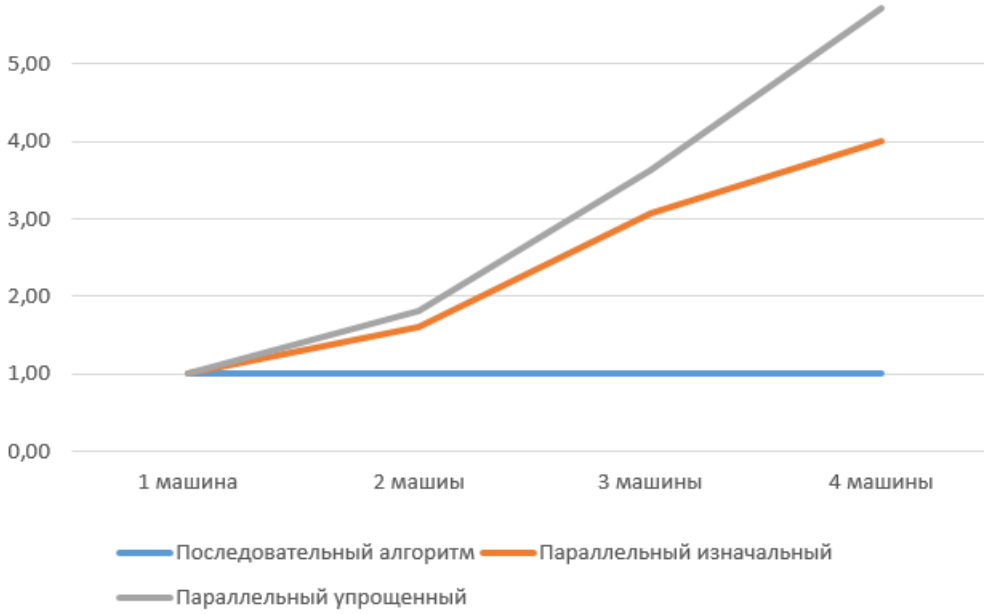

Рис. 7: Тест производительности: ускорение

Действительно, упрощение распредленного алгоритма дает больший прирост производительности. Стоит отметить, что как и ожидается, графики ускоренных алгоритмов находят со временем свои ассимптоты.

Заключение

В рамках данной работы предстояло

- произвести обзор предметной области;
- разработать распределенный существующего алгоритма обучения SPF "на бумаге";
- обеспечить кроме количественоого роста (увеличения производительности алгоритма) качественный рост параллельной реализации;
- произвести тестирование разработанного алгоритма.

Поставленные задачи успешно решены. В данной работе рассмотрены возможности для параллелизации алгоритма обучения SPF. А также представлена реализация распределенного алгоритма. Результом работы является достигнутое ускорение алгоритма при работе на нескольких машинах.

Дальнейшая работа

Дальнейшая работа может производится во множестве направлений. Одним из таких направлений является получение результатов производительности алгоритма обучения на других фреймворках параллельных вычислений, таких как Spark, Hadoop и др., что, кстати, избавит исследователя от проблемы выбора сериализатора (разные сериализаторы могут показывать различные результаты в бенчмарках на различных данных). Но с другой стороны, в силу большого объема кода (25 000 строк кода), задача является очень трудоемкой.

Модификация фреймворка для прохождения тестов Trec 1999 [29], что является довольно большой и трудоемкой задачей в силу того, что фреймворк предназначен в основном для нахождения логических форм предложений и нуждается в доработке для ответов на вопросы.

Интерсным направлением для работы представляется модификация GENLEX-компоненты. А точнее, переписывание ее с Java на Scala или любой другой язык, реализующий парадигму функционального программирования. Это может сократить количество кода в проекте, сделать его более поддерживаемым и более расширяемым.

В силу большой существующей кодовой базы, представленные направления работы совершенно точно представляют собой очень трудоемкие задачи.

Список литературы

1. Bert F. Green, Jr., Alice K. Wolf, Carol Chomsky, and Kenneth Laughery. BASEBALL: AN AUTOMATIC QUESTION-ANSWERER // IRE-AIEE-ACM '61 (Western) C. 219-224

2. J. Weizenbaum. ELIZA—a computer program for the study of natural language communication between man and machine // Communications of the ACM CACM Vol. 9 No 1, 1966 P. 36-45

3. Woods W. A., 1973 Semantics and Quantification in Natural Language Question Answering // Advances In Computers. Vol. 17

4. Woods W. A., William A. Transition Network Grammars for Natural Language Analysis. // Communications of the ACM. Vol. 13 No, 10, P. 591–606

5. Katz, B. Annotating the World Wide Web using Natural Language // Proceedings of the 5th RIAO Conference on Computer Assisted Information Searching on the Internet (RIAO '97)

6. Winograd T. Procedures as a Representation for Data in a Computer Program for Understanding Natural Language // MIT AI Technical Report, 1971. No. 235.

7. Kupiec J. MURAX: a robust linguistic approach for question answering using an on-line encyclopedia Kupiec // ACM SIGIR conference on Research and development in information retrieval, 1993 P. 181-190

8. Zue, S. Seneff, J. Glass, J. Polifroni, C. Pao, T.J. Hazen, and L. Hetherington. JUPITER: A telephone-based conversational interface for weather information. IEEE Transactions on Speech and Audio Processing, 2000, Vol. 8 No, 1.

9. Pundge A. Online learning different approaches and necessity for assessment. // The international conference on recent trends and challenges in science and Technology (RTCST2014), 2014 at padmashri vikhe patil college of Arts, science and commerce.

10. Cai D, Dong Y, Lv D, Zhang G, Miao X. A Web-based Chinese question answering with answer validation. // IEEE International Conference on Natural Language Processing and Knowledge Engineering, pp. 499-502, 2005.

11. Soricut R and Brill E. Automatic question answering using the web. Beyond the factoid. // Journal of Information Retrieval-Special Issue on Web Information Retrieval, 2006, Vol. 9 No. 2, P. 191-206.

12. Ravichandran D., Ittycheriah A, Automatic Derivation of surface text pattern for a maximum Entropy Based question answering system // Work done while the author was an intern at IBM TJ Watson research center during summer 2002.

13. Vanitha Guda. Approaches for question answering systems // International Journal of Engineering science and technology (IJEST), 2011, Vol.3 No.2

14. Xin Li, Dan Roth, 2002. Learning Question Classifiers // COLING 2002

15. WordNet https://wordnet.princeton.edu/

16. Moldovan D., Harabagui S., Paca M., Mihalcea R., Goodrum R., Girju R., Rus V., // TREC-8, 1999

17. Pasca, M. Open-Domain Question Answering from Large Text Collections. // CSLI, 2003.

18. Kwiatkowski T., Zettlemoyer L., Goldwater S., Steedman M. Lexical generalization in CCG grammar induction for semantic parsing // EMNLP '11 Proceedings of the Conference on Empirical Methods in Natural Language Processing, P. 1512-1523

19. Zettlemoyer L.S., Collins M., Learning to Map Sentences to Logical Form: Structured Classification with Probabilistic Categorial Grammars // Proceedings of the Twenty-First Conference on Uncertainty in Artificial Intelligence (UAI2005), 2005. P. 658-666

20. A Very Short Introduction to CCG by Mark Steedman (Draft) http://www.inf.ed.ac.uk/teaching/courses/nlg/readings/ccgintro.pdf

21. Speech and Language Processing. Daniel Jurafsky & James H. Martin. draft http://www.deepsky.com/ merovech/voynich/voynich_manchu_materials/PDFs/

22. Artzi Y. Cornell SPF: Cornell Semantic Parsing Framework https://arxiv.org/pdf/1311.3011.pdf

23. https://github.com/cornell-lic/spf

24. http://www.cs.utexas.edu/users/ml/nldata/geoquery.html

25. http://doc.akka.io

26. Actor Model of Computation: https://arxiv.org/abs/1008.1459

27. https://github.com/eishay/jvm-serializers/wiki

28. https://github.com/romix/akka-kryo-serialization

29. http://trec.nist.gov/data/qa/t8_qadata.html