ГЛАВА 3

Создание первого сквозного пайплайна

Часть I мы начали с разговора о том, как на основе требований к продукту выбрать подходы к моделированию. Затем мы перешли к этапу планирования и поговорили о том, как искать релевантные ресурсы и использовать их при составлении начального плана разработки. Наконец, мы узнали, что создание исходного прототипа работоспособной системы — лучший способ продвинуться в работе. Именно этому этапу будет посвящена глава 3.

Первая версия приложения не должна быть идеальной. Ее задача — предоставить все элементы пайплайна, чтобы мы могли понять, какие из них следует улучшать в первую очередь. Создание полного прототипа — простейший способ выявить то узкое место, о котором говорила Моника Рогати в разделе «Моника Рогати: как выбирать и приоритизировать МО-проекты» на с. 40.

Давайте начнем с создания простейшего пайплайна для выдачи предсказаний на основе входных данных.

Простейший «каркас» приложения

Мы уже говорили, что большинство МО-моделей включает в себя два пайплайна: пайплайн обучения и пайплайн инференса. Пайплайн обучения позволяет получить высококачественную модель, а пайплайн инференса обеспечивает поставку результатов пользователям. Подробное описание различий между этими пайплайнами см. в разделе «Начинайте с простого пайплайна» на с. 61.

При создании первого прототипа приложения мы сосредоточимся на поставке результатов пользователям. Это означает, что мы начнем разработку с пайплайна инференса. Это позволит нам быстро выяснить, как пользователи могут взаимодействовать с результатами модели, и таким образом собрать полезную информацию, позволяющую упростить обучение модели.

Сосредоточившись на инференсе, на время забудем об обучении. И поскольку пока мы не обучаем модель, мы можем вместо этого составить ряд простых правил. Создание таких правил, или эвристических алгоритмов, часто является прекрасным способом начать работу. Это самый быстрый способ создать прототип и сразу получить упрощенную версию всего приложения.

Хотя это может показаться избыточным, ведь в конечном итоге планируется реализовать решение на базе МО (как это и будет сделано далее в книге), но создание такого прототипа играет важную роль, заставляя нас внимательно изучить задачу и выработать исходный набор гипотез для оптимального решения этой задачи.

Создание, проверка и доработка гипотез наилучшего способа моделирования данных — ключевые составляющие итеративного процесса создания модели, который начинается еще до появления первой модели.

Вот два примера отличных эвристических алгоритмов, примененных в компании Insight Data Science, которые мне довелось курировать.

- *Оценка качества кода.* Решив создать модель, способную на основе примера кода предсказать, какого успеха может добиться программист на сайте HackerRank (посвященном соревновательному программированию), Даниэль начал с подсчета количества открытых и закрытых круглых, квадратных и фигурных скобок.

  Поскольку в хорошо работающем коде количество открытых и закрытых скобок, как правило, совпадает, это правило оказалось достаточно эффективной отправной точкой и также позволило понять, что при моделировании следует сосредоточиться на сборе информации о структуре кода с помощью абстрактного синтаксического дерева[1].

- *Подсчет деревьев.* Майк решил создать модель для подсчета количества растущих в городе деревьев на основе спутникового снимка. Изучив некоторые данные, он начал с разработки правила для оценки насыщенности местности деревьями на основе относительной доли зеленых пикселей на изображении.

  Как оказалось, этот подход хорошо работает для отдельно стоящих деревьев, но дает сбой, когда приходится иметь дело с большим количеством растущих рядом деревьев. Это позволило понять, что при моделировании следует сосредоточиться на создании пайплайна, способного обрабатывать группы плотно растущих деревьев.

Работу над большинством проектов МО следует начинать с создания такого эвристического алгоритма. Главное — помнить о том, что этот алгоритм нужно создавать на основе экспертных знаний и изучения данных, а затем использо-

[1] https://oreil.ly/L0ZFk

вать для подтверждения начальных предположений и ускорения итеративной доработки.

После того как вы разработаете эвристический алгоритм, можно будет создать пайплайн сбора и предобработки данных, применения к ним ваших правил и поставки результатов пользователям. Зачастую для этого достаточно написать запускаемый из консоли Python-скрипт или веб-приложение, принимающее сигнал от камеры пользователя и выдающее результат в реальном времени.

Идея та же, что и в выборе метода МО: максимально упростить продукт, чтобы получить предельно простую работоспособную версию. Создание такого минимально жизнеспособного продукта (MVP) — проверенный способ скорейшего получения полезных результатов.

Прототип МО-редактора

Для нашего МО-редактора возьмем общие рекомендации по редактированию текстов, выработаем ряд правил — что такое хорошие или плохие вопросы, а затем отобразим пользователям результаты применения этих правил.

Чтобы создать минимальную версию нашего продукта, возвращающую рекомендации на основе пользовательского ввода, нам потребуется написать всего четыре функции:

```
input_text = parse_arguments()
processed = clean_input(input_text)
tokenized_sentences = preprocess_input(processed)
suggestions = get_suggestions(tokenized_sentences)
```

Давайте подробно рассмотрим каждую из этих функций. Функция синтаксического анализатора `parse_arguments()` очень простая — она принимает от пользователя строку без каких-либо параметров. Исходный код этого и других примеров кода можно найти в GitHub-репозитории нашей книги.

Парсинг и очистка данных

Прежде всего, мы выполним парсинг данных, принимаемых из командной строки. Это достаточно просто реализовать на языке Python:

```
def parse_arguments():
    """

    :return: Текст, который необходимо отредактировать
    """
```

```
    parser = argparse.ArgumentParser(
        description="Receive text to be edited"
    )
    parser.add_argument(
        'text',
        metavar='input text',
        type=str
    )
    args = parser.parse_args()
    return args.text
```

Перед тем как передавать модели любые входные данные, их сначала нужно валидировать и верифицировать. Поскольку в нашем случае данные вводятся пользователем, мы должны позаботиться о том, чтобы они содержали символы, поддающиеся синтаксическому разбору. Чтобы очистить входные данные, удалим из них любые символы не в кодировке ASCII. Это позволит нам делать обоснованные допущения о содержании текста и в то же время не сильно ограничит креативность наших пользователей.

```
def clean_input(text):
    """

    :param text: Введенный пользователем текст
    :return: Очищенный текст, содержащий только символы в кодировке ASCII
    """
    # В целях упрощения сначала оставим только символы в кодировке ASCII
    return str(text.encode().decode('ascii', errors='ignore'))
```

Теперь нам нужно предобработать входные данные и выдать рекомендации. Для начала мы можем опереться на результаты исследований в области классификации текста, о которых упоминалось в разделе «Простейший подход: “сам себе алгоритм”» на с. 37. Это подразумевает определение степени сложности текста путем подсчета сводной статистики по количеству слогов, слов и предложений.

Для подсчета статистики на уровне слов мы должны научиться выделять в предложениях отдельные слова. В сфере обработки естественного языка этот процесс называют *токенизацией*.

Токенизация текста

Реализовать токенизацию не так просто, как кажется, поскольку большинство очевидных методов, например разбиение входного текста на слова с помощью пробелов и точек, плохо работает на реальных текстах в силу того, что слова могут отделяться друг от друга множеством разных способов. Например, взгляните

на следующее предложение, которое приводится в качестве примера на курсе по обработке естественного языка[1] в Стэнфордском университете.

«Mr. O'Neill thinks that the boys' stories about Chile's capital aren't amusing.»

Большинство простых методов дадут сбой на этом предложении из-за наличия в нем точек и апострофов с разным смыслом[2]. Вместо того чтобы создавать собственный токенизатор, мы воспользуемся популярной библиотекой с открытым исходным кодом nltk[3], которая позволяет реализовать токенизацию двумя простыми действиями:

```
def preprocess_input(text):
    """

    :param text: Очищенный текст
    :return: Текст, подготовленный к анализу: предложения разбиты на лексемы
    """
    sentences = nltk.sent_tokenize(text)
    tokens = [nltk.word_tokenize(sentence) for sentence in sentences]
    return tokens
```

После того как мы предобработаем данные, их можно будет использовать для генерирования признаков, позволяющих оценить качество вопроса.

Генерирование признаков

Последний шаг — написать несколько правил для выдачи рекомендаций пользователям. Для нашего простого прототипа начнем с определения частоты употребления ряда распространенных глаголов, союзов и наречий, а затем вычислим индекс удобочитаемости Флеша (Flesch readability score)[4]. После этого мы можем вывести пользователю отчет с этими метриками:

```
def get_suggestions(sentence_list):
    """

    Возвращает строку, содержащую наши рекомендации
```

[1] https://oreil.ly/vdrZW

[2] Апострофы имеют разное значение, и простое разделение данного предложения на слова по неалфавитным символам не принесет нужного результата. В слове «O'Neill» апостроф отделяет первую букву фамилии; в словах «boys'» и «Chile's» образует притяжательную форму, причем в лексических единицах во множественном числе апостроф стоит после окончания «s»; а в «aren't» он указывает на опускаемую в слове букву. — *Примеч. ред.*

[3] https://www.nltk.org/

[4] https://oreil.ly/iKhmk

```
    :param sentence_list: список из предложений, каждое из которых
    является списком слов
    :return: рекомендации по улучшению входных данных
    """
    told_said_usage = sum(
        (count_word_usage(tokens, ["told", "said"]) for tokens in sentence_list)
    )
    but_and_usage = sum(
        (count_word_usage(tokens, ["but", "and"]) for tokens in sentence_list)
    )
    wh_adverbs_usage = sum(
        (
            count_word_usage(
                tokens,
                [
                    "when",
                    "where",
                    "why",
                    "whence",
                    "whereby",
                    "wherein",
                    "whereupon",
                ],
            )
            for tokens in sentence_list
        )
    )
    result_str = ""
    adverb_usage = "Adverb usage: %s told/said, %s but/and, %s wh adverbs" % (
        told_said_usage,
        but_and_usage,
        wh_adverbs_usage,
    )
    result_str += adverb_usage
    average_word_length = compute_total_average_word_length(sentence_list)
    unique_words_fraction = compute_total_unique_words_fraction(sentence_list)

    word_stats = "Average word length %.2f, fraction of unique words %.2f" % (
        average_word_length,
        unique_words_fraction,
    )
    # Используем HTML-тег для отображения в веб-приложении разрыва строки
    result_str += "<br/>"
    result_str += word_stats

    number_of_syllables = count_total_syllables(sentence_list)
    number_of_words = count_total_words(sentence_list)
    number_of_sentences = len(sentence_list)

    syllable_counts = "%d syllables, %d words, %d sentences" % (
        number_of_syllables,
        number_of_words,
        number_of_sentences,
```

```
    )
    result_str += "<br/>"
    result_str += syllable_counts

    flesch_score = compute_flesch_reading_ease(
        number_of_syllables, number_of_words, number_of_sentences
    )

    flesch = "%d syllables, %.2f flesch score: %s" % (
        number_of_syllables,
        flesch_score,
        get_reading_level_from_flesch(flesch_score),
    )

    result_str += "<br/>"
    result_str += flesch

    return result_str
```

Вот и всё! Теперь мы можем вызвать свое приложение из командной строки и сразу же увидеть результаты. И хотя пока оно не отличается удобством, мы имеем отправную точку для дальнейшего тестирования и итеративной доработки, чем и займемся далее.

Оцените рабочий процесс

Теперь, имея прототип, можем проверить, насколько правильно мы сформулировали задачу и насколько полезно наше решение. В данном разделе мы поговорим о том, как можно объективно оценить качество исходных правил и выяснить, в удобной ли форме мы предоставляем результаты.

Как сказала ранее Моника Рогати, «часто продукт терпит неудачу, несмотря на успешную работу модели». Если выбранный нами метод будет хорошо справляться с оценкой качества вопросов, но мы не сможем предоставить пользователям какие-либо рекомендации по улучшению формулировки, то практической пользы от нашего продукта не будет, несмотря на высокое качество примененного метода. Давайте рассмотрим весь пайплайн и оценим практичность текущего пользовательского опыта параллельно с результатами нашей вручную сделанной модели.

Пользовательский опыт

Для начала давайте оценим удобство использования продукта безотносительно к качеству модели. Допустим, что в конечном итоге мы получим достаточно эффективную модель. Будет ли в таком случае способ представления результатов пользователям, который используется сейчас, лучшим?

Так, в приложении по подсчету деревьев, вероятно, потребуется представить результаты в виде итогового отчета по всему городу. Нужно указать количество обнаруженных деревьев по городу в целом и по каждому району и измерить ошибку на эталонном тестовом наборе.

Иначе говоря, нужно убедиться в том, что результаты удобны в использовании (или станут таковыми после доработки модели), а также что модель работает хорошо. Именно это мы и сделаем на следующем шаге.

Результаты моделирования

В разделе «Оценка успешности» на с. 44 мы уже говорили о том, как важно сфокусироваться на правильной метрике. Наличие работоспособного прототипа на ранних этапах поможет нам выявить и затем итеративно уточнить набор метрик, отражающий степень успешности продукта.

Так, например, если бы мы решили создать систему, помогающую пользователям искать ближайшие прокатные автомобили, мы могли бы использовать метрику дисконтированного совокупного выигрыша (discounted cumulative gain, DCG). Эта метрика измеряет качество ранжирования: наиболее высокое значение выдается в том случае, когда наиболее релевантные результаты возвращаются раньше других (более подробные сведения о метриках ранжирования см. в статье о DCG[1] в Википедии). При создании первой версии этого продукта мы могли бы допустить, что полезна хотя бы одна из первых пяти рекомендаций, то есть принять DCG равным 5. Однако при тестировании этого продукта на пользователях может оказаться, что они обращают внимание только на первые три результата в выдаче. В таком случае надо изменить метрику успешности на DCG равный 3, а не 5.

Оценивать и пользовательский опыт, и степень эффективности модели нужно, чтобы направить усилия на доработку того, что более важно. Если пользовательский опыт будет плохим, то вам вряд ли поможет улучшение модели. Более того, в таком случае стоит подумать об использовании совершенно другой модели! Давайте рассмотрим два примера.

Нахождение узкого места

Рассмотрев результаты моделирования и текущее представление продукта, мы должны определиться с тем, что делать дальше. Обычно дальше следует итеративная доработка способа представления результатов пользователям (что часто подразумевает изменение метода обучения моделей) или повышение производительности модели путем выявления основных проблемных мест.

[1] https://oreil.ly/b_8Xq

Анализом ошибок мы вплотную займемся в части III, однако уже сейчас следует установить основные виды проблем и возможные способы их устранения. Необходимо определить, что важнее исправить — модель или продукт, поскольку первый и второй случаи требуют разного подхода. Давайте рассмотрим соответствующие примеры.

Проблема с продуктом

Допустим, что вы создали модель, которая должна на основе фотографий научной статьи предсказывать, будет ли она принята к рассмотрению на ведущих конференциях (см. посвященную этому статью Цзя-Биня Хуана (Jia-Bin Huang) «Гештальт научной статьи» («Deep Paper Gestalt»[1])). Однако вы заметили, что простое информирование пользователя о вероятности отказа принесет ему мало пользы. В таком случае вам *не поможет* улучшение модели. Будет разумнее сфокусироваться на извлечении из модели рекомендаций по доработке статьи для повышения шансов на то, что она будет принята к рассмотрению.

Проблема с моделью

Допустим, что вы создали модель для оценки платежеспособности заемщиков и заметили, что она присваивает более высокий риск неплатежеспособности представителям определенной этнической группы при прочих равных факторах. Это может объясняться необъективностью обучающих данных, что можно исправить путем сбора более репрезентативного датасета и создания нового пайплайна его очистки и пополнения. В таком случае вам *потребуется исправить модель*, вне зависимости от способа представления результатов. Поскольку такие ситуации происходят часто, всегда следует учитывать не только агрегатные метрики, но и производительность модели на различных срезах данных. Об этом мы подробно поговорим в главе 5.

Давайте рассмотрим все это на примере нашего МО-редактора.

Оценка прототипа МО-редактора

Давайте посмотрим, насколько хорош наш исходный пайплайн с точки зрения и пользовательского опыта, и производительности модели. Для начала попробуем представить нашему приложению несколько примеров входных данных. Как оно поведет себя в случае простого вопроса, сложного вопроса и целого абзаца?

Поскольку мы используем индекс удобочитаемости, то в идеале надо организовать рабочий процесс так, чтобы мы получали высокий индекс для простого предложения, низкий индекс для сложного и рекомендации по улучшению для

[1] https://oreil.ly/RRfIN

абзаца. Что ж, давайте наконец прогоним эти несколько образцов через наш прототип.

Простой вопрос:

```
$ python ml_editor.py "Is this workflow any good?"
Adverb usage: 0 told/said, 0 but/and, 0 wh adverbs
Average word length 3.67, fraction of unique words 1.00
6 syllables, 5 words, 1 sentences
6 syllables, 100.26 flesch score: Very easy to read
```

Сложный вопрос:

```
$ python ml_editor.py "Here is a needlessly obscure question, that"\
"does not provide clearly which information it would"\
"like to acquire, does it?"
Adverb usage: 0 told/said, 0 but/and, 0 wh adverbs
Average word length 4.86, fraction of unique words 0.90
30 syllables, 18 words, 1 sentences
30 syllables, 47.58 flesch score: Difficult to read
```

Целый абзац (отличный от предыдущих вопросов):

```
$ python ml_editor.py "Ideally, we would like our workflow to return
                       a positive"\
" score for the simple sentence, a negative score for the convoluted one, and "\
"suggestions for improving our paragraph. Is that the case already?"
Adverb usage: 0 told/said, 1 but/and, 0 wh adverbs
Average word length 4.03, fraction of unique words 0.76
52 syllables, 33 words, 2 sentences
52 syllables, 56.79 flesch score: Fairly difficult to read
```

Давайте посмотрим, возникли ли проблемы с моделью или пользовательским опытом.

Модель

Пока неясно, насколько хорошо наши результаты соответствуют представлениям о качественном тексте. Сложному предложению и целому абзацу был присвоен почти одинаковый индекс удобочитаемости. Признаюсь, мои тексты иногда сложны для восприятия, но все же данный абзац более понятен, чем проверенное перед ним сложное предложение.

Вероятно, извлекаемые атрибуты не всегда хорошо коррелируют с нашими представлениями о «хорошем» тексте. Это объясняется тем, что мы недостаточно четко определили критерий успешности: если мы возьмем два вопроса, как можно определить, какой из них лучше сформулирован? Мы определим это более четко в следующей главе, где займемся созданием датасета.

Очевидно, нам еще потребуется доработать модель, но может быть, мы уже добились удобного представления результатов?

Пользовательский опыт

Исходя из результатов выше, очевидны две проблемы: возвращаемая информация слишком громоздкая и малополезная. Наша цель — предоставить пользователям практичные рекомендации. И хотя признаки и индекс удобочитаемости дают представление о качестве текста, они не помогают пользователю понять, как улучшить свой текст. Вероятно, нам стоит сократить свои рекомендации до одного показателя и добавить практические советы по улучшению текста.

Мы могли бы давать некие общие рекомендации, например использовать меньше наречий, или быть более конкретными, предлагая изменения на уровне слов и предложений. В идеале в выводимом результате могли бы подчеркиваться или выделяться цветом те части входного текста, на которые следует обратить внимание. Макет того, как это может выглядеть, представлен на рис. 3.1.

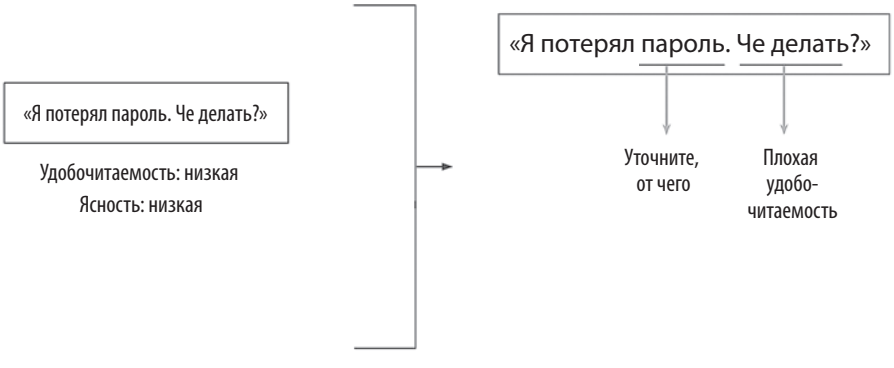

Рис. 3.1. Применимые на практике рекомендации

Даже если мы не сможем давать рекомендации, выделяя отдельные части входной строки, предоставление рекомендаций в формате, показанном справа на рис. 3.1, пойдет нашему продукту на пользу, поскольку такие рекомендации более применимы на практике, чем список показателей.

Заключение

Мы создали исходный прототип инференса и использовали его для оценки качества нашего эвристического алгоритма и последовательности операций.