МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение
высшего образования
«Тольяттинский государственный университет»

Институт математики, физики и информационных технологий
(наименование института полностью)

Кафедра «Прикладная математика и информатика»
(наименование)

02.03.03 Математическое обеспечение и администрирование информационных систем
(код и наименование направления подготовки, специальность)

Мобильные и сетевые технологии
(направленность (профиль) / специализация)

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА (БАКАЛАВРСКАЯ РАБОТА)

на тему «Разработка алгоритма для приоритизации дефектов программного обеспечения»

Студент В.А. Ячевский
(И.О. Фамилия) (личная подпись)

Руководитель канд.тех.наук, доц., В.С. Климов
(Ученая степень, звание, И.О. Фамилия)

Консультант М.В. Дайнеко
(И.О. Фамилия)

Тольятти 2021

Аннотация

Тема выпускной квалификационной работы: «Разработка алгоритма для приоритизации дефектов программного обеспечения».

В данной выпускной работе рассматривается процесс автоматизации присвоения приоритета дефектам ПО.

В работе предложен алгоритм классификатора дефектов программного обеспечения, чья работа основана на использовании рекуррентной нейронной сети.

Выпускная квалификационная работа содержит: введение, три главы, заключение, список используемой литературы и используемых источников.

Во введении дается характеристика актуальности вышеозначенной темы, а также описывается основная цель исследования и задачи, выполнение которых необходимо для реализации алгоритма.

В первой главе рассматриваются принципы работы с дефектами программного обеспечения, а также проводится анализ существующих работ, посвященных теме.

Во второй главе описывается разработка алгоритма классификатора для приоритизации дефектов программного обеспечения.

В третьей главе проводится анализ результатов работы классификатора, а также осуществляется оценка точности работы алгоритма средствами статистического анализа.

В заключении приводятся выводы по выполненной работе.

В работе представлено: 6 формул, 16 рисунков, 2 таблицы и 20 ссылок на использованные источники. Общий объём выпускной квалификационной работы составляет 44 страницы.

ABSTRACT

The topic of the present graduation work is *Designing an algorithm for prioritizing software bugs and errors.*

This research is devoted to automating the prioritization of software bugs and errors.

The graduation work consists of 6 formulae, 16 figures, 2 tables and a list of 20 references.

The work proposes an algorithm of a software bugs and errors classifier. Its operation is based on using a recurrent neural network.

The graduation work is divided into several logically connected parts which are an introduction, three chapters, conclusion and the list of references.

The introduction explains the relevance of the topic, as well as states the main purpose of the study and sets the objectives to be attained to implement the algorithm.

The first chapter reviews the principles of dealing with software bugs and errors, as well as analyzes the existing scientific works on the topic of the research.

The second chapter describes the development of a classifier algorithm for software bugs and errors prioritization. We also develop the application programming interface for testing purposes and for demonstration of possible integration.

The third chapter analyzes the results of the classifier operation and assesses the accuracy of the algorithm by means of statistical analysis.

In conclusion, we would like to emphasize that the developed software can be used in combination with the existing tools for providing software maintenance due to high predictive abilities of the proposed algorithm.

Оглавление

Введение

В машинном обучении классификация относится к задаче прогнозного моделирования, при решении которой метка класса прогнозируется для некоторого набора входных данных. Этот раздел машинного обучения достаточно проработан в теоретическом и практическом плане, поскольку проблема классификации находит применение в различных прикладных задачах. В качестве примера подобных задач можно привести задачи медицинской диагностики, оценки кредитоспособности заёмщиков, распознавания речи, а также классификации документов.

О классификации документов, а именно отчётов о дефектах программного обеспечения (ПО), и пойдет речь в данной выпускной квалификационной работе. Актуальность этой проблемы обусловлена тем, что поддержка программного обеспечения является неотъемлемой частью процесса его разработки. В частности, разрешение той или иной проблемы в ПО, на которую был заведён отчёт о дефекте, в значительной степени зависит от приоритета конкретного дефекта, заданного в системе отслеживания ошибок. Поскольку дефекты зачастую заводятся пользователями, то есть людьми, не обладающими специфическими техническими знаниями для объективной оценки возникшей проблемы, задача приоритизации дефектов ложится на плечи разработчиков ПО. В связи с этим, очевидной становится возможность автоматизации данного процесса.

В качестве объекта исследования выступает процесс определения приоритета дефектов ПО.

Предметом настоящего исследования является алгоритм классификатора для автоматизированной приоритизации дефектов ПО.

Целью данной работы является разработка алгоритма приоритизации дефектов ПО.

Для достижения вышеописанной цели необходимо решить следующие задачи:

1. Рассмотреть проблематику работы с дефектами ПО;

2. Провести анализ изученности представленной проблемы в научной и учебно-методической литературе;

3. Предложить способ осуществления автоматизации работы с дефектами ПО;

4. Выбрать подход к реализации алгоритма приоритизации дефектов ПО средствами машинного обучения;

5. Осуществить программную реализацию выбранного подхода;

6. Подготовить программный интерфейс для разработанного алгоритма;

7. Определить эффективность разработанного алгоритма;

8. Обобщив полученные экспериментальные результаты, сделать вывод об эффективности работы алгоритма.

Глава 1 Особенности приоритизации дефектов

1.1 Описание проблематики работы с дефектами

Программные проекты (как с открытым исходным кодом, так и с закрытым) получают огромное количество отчётов об ошибках, а наличие ошибок, в свою очередь, обычно влияет на надежность, качество и регулирование издержек программного обеспечения (ПО). На практике невозможно иметь ПО без ошибок, поскольку трудоёмкость такого программного обеспечения будет однозначно превышать пользу от его эксплуатации. Из-за этого программных дефектов избежать нельзя, и многие программные продукты поставляются с ошибками. Для их устранения и повышения качества выпускаемых продуктов используются системы отслеживания ошибок, такие как JIRA, которые позволяют пользователям и разработчикам сообщать о дефектах.

Системы отслеживания ошибок помогают прогнозировать прогресс тех или иных этапов разработки на основе полученных отчётов об ошибках. Они позволяют пользователям добавлять истории (функциональные требования) и разделять их на задачи, а также создавать отчёты об ошибках и наборы тестов. Разработчики и тестировщики могут создавать новые отчёты об ошибках, отслеживать их состояние, а также обновлять существующие отчёты об ошибках. Как правило, отчёты об ошибках проходят серию определенных состояний, начинающейся с момента обнаружения дефекта и заканчивающейся закрытием отчёта.

Отчёты об ошибках могут впоследствии использоваться для того, чтобы осуществлять корректирующее сопровождение ПО – то есть производить исправление или обход ошибок, выявленных в ходе эксплуатации – и способствовать созданию более стабильных программных систем. Понятие приоритета – важности дефекта по оказываемому им влиянием на ПО – является значимым в процессе сортировки ошибок и

позволяет разработчикам корректно расставлять приоритеты в своей работе по исправлению дефектов, осуществляя тем самым отладку наиболее влияющих на работу с ПО дефектов в первую очередь. Разработчики часто получают многочисленные отчёты об ошибках, и им не всегда удается исправить всё из-за различных ограничений, включая временные. Приоритизация дефектов, как правило, осуществляется вручную, и занимает много времени. Таким образом, становится очевидной потребность в модели для предсказания приоритета дефектов ПО, которая смогла бы помочь автоматизировать процесс определения данного параметра. С её помощью удалось бы достичь таких результатов, как:

– повышение точности и эффективности при определении приоритета программных дефектов,

– увеличение эффективности за счёт снижения временных затрат при ручном определении приоритета,

– уменьшение производственных затрат (человеко-часов) при неверном определении приоритета.

В качестве исходных данных в работе выступают отчёты о дефектах, извлеченные в csv-формате из закрытого JIRA-репозитория компании Netcracker. Они послужат источником информации для обучения прогнозной модели. В связи с этим, необходимо ознакомиться со структурой и сущностью отчётов о дефектах, чтобы выявить наиболее значимые для модели параметры.

1.2 Жизненный цикл дефектов и их содержание

Отчёты о дефектах проходят через определенный жизненный цикл в процессе своего существования. Он подразделяется на пять состояний: Open, In Progress, Resolved, Closed и Reopened.

Когда тестировщик создаёт отчёт о дефекте в системе отслеживания ошибок, отчёт переходит в статус Open. После этого, руководитель команды

назначает разработчика, подходящего для оценки проблемы и её разрешения. Как только разработчик начинает анализ и работу над исправлением дефекта, статус отчета изменятся на In Progress.

В случае, если дефект был создан более одного раза, он переводится в статус Closed с решением Duplicate. Если же дефект не удаётся воспроизвести, то есть все попытки повторить проблемное поведение не увенчались успехом, или же для воспроизведения проблемы был дан недостаточный объём сведений), то отчёт о дефекте также переводится в статус Closed, но уже с типом решения Cannot Reproduce.

После того, как разработчик исправил проблему и подтвердил внесённые изменения, статус изменяется на Resolved. За этим следует этап тестирования, в ходе которого тестировщик или подтверждает внесенное изменение, или повторно тестирует проблемное поведение, чтобы убедиться в отсутствии ошибки в программном обеспечении. При подтверждении разрешения проблемы отчёт об ошибке переводится в статус Closed. В случае же, если дефект всё ещё присутствует в продукте и не исправлен должным образом, статус изменяется на Reopened.

Наконец, когда подходит время для выдачи программного обеспечения (например, в рамках Agile-спринтов), неисправленные дефекты с низким приоритетом переносятся владельцем продукта на следующий релиз со статусом Open. На рисунке 1 представлена диаграмма жизненного цикла дефектов:

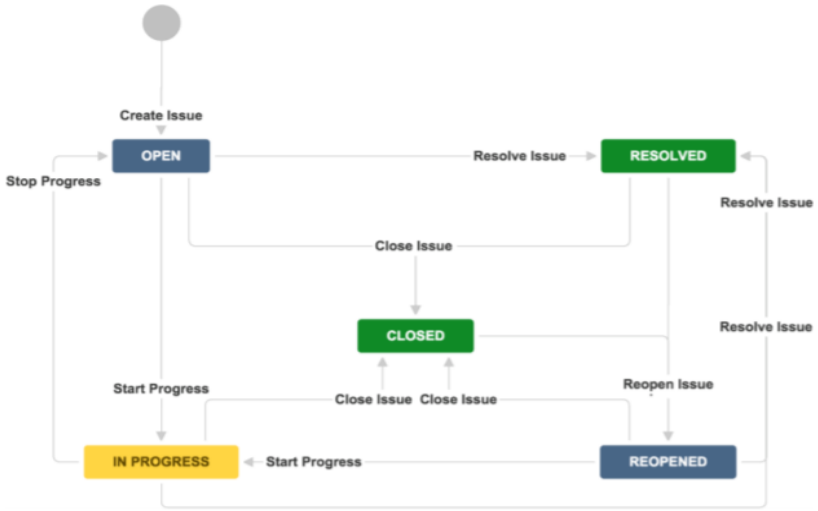


Рисунок 1 – Жизненный цикл дефектов ПО

Отчёт о дефекте содержит информацию о том, как ошибка может быть воспроизведена, а также информацию, которая может помочь в отладке и трассировке. Он включает в себя такие параметры, как название (Summary), описание (Description), приоритет (Priority), описание окружения (Environment), вложения (Attachment), Assignee, Reporter, дата создания (Created Date), статус отчёта (Status), Fix Version и Component. Подробное описание каждого из полей дано в таблице 1:

Таблица 1 – Описание полей, содержащихся в отчёте о дефекте

Наименование поля	Описание
Summary	Краткое описание дефекта в одну строчку
Description	Детальное описание тестовых шагов для воспроизведения проблемы, фактический результат при выполнении сценария, а также ожидаемый результат
Priority	Показатель того, как быстро должен быть исправлен дефект
Environment	Окружение, в котором был обнаружен дефект
Attachment	Документы, снимки экрана и прочие элементы, которые могут помочь в воспроизведении и исправлении дефекта
Assignee	Сотрудник, создавший отчёт о дефекте
Reporter	Сотрудник, который ответственен за исправление ошибки
Created Date	Дата создания отчёта о дефекте
Status	Состояние дефекта согласно жизненному циклу
Fix Version	Версия продукта, содержащая ошибку
Component	Компонент продукта, с которым связана ошибка

В процессе жизненного цикла отчёта о дефекте, приоритет дефекта может изменяться. Это обусловлено человеческим фактором в ходе работы над дефектами – так, приоритет может понизиться в случае, если для ошибки находится обходной путь, то есть способ её избегания в контексте работы с ПО. Поэтому отчёты, взятые в качестве исходных данных, содержат только дефекты со статусом Closed – это позволяет убедиться в том, что заданный им приоритет является окончательным и уже изменяться не будет.

Для того, чтобы обучить прогнозную модель, необходимо использовать параметры из отчёта о дефекте. Исходя из описания в таблице 1, такие параметры, как Status, Created Date, Fix Version, Attachment, Environment, либо не содержат информации, подходящей для того, чтобы использовать ее для классификации, либо, как параметры Assignee и Reporter, не имеют достаточной репрезентативности при их использовании на более широком наборе данных. Такой вывод делается исходя из того факта, что на разных проектах задействованы разные сотрудники, и применение параметров Assignee и Reporter может снизить точность модели в процессе обучения.

Таким образом, в качестве входных данных для модели допустимо использовать параметры Summary, Description и Component, поскольку текстовые данные, содержащиеся в них, соответствуют вышеописанным требованиям.

1.3 Анализ существующих работ по представленной проблеме

В ходе поиска информации по проблеме классификации дефектов ПО было обнаружено, что работы русскоязычных учёных по данному вопросу составляют незначительную часть всего корпуса научных работ. Это связано с тем, что вопросы автоматизации в процессах сопровождения и обеспечения качества ПО стали освещаться в отечественных научных работах сравнительно недавно.

Исследования можно подразделить на два вида: применение традиционного подхода к созданию классификаторов и использование алгоритмов глубокого обучения. К первым относится применение таких алгоритмов классификации, как метод опорных векторов (SVM, Support Vector Machine), C4.5 (алгоритм для построения деревьев решений), наивный байесовский классификатор, метод k-ближайших соседей и прочие. Среди средств глубокого обучения выделяют методики, основанные на использовании рекуррентных нейронных сетей.

Традиционный подход к работе с классификаторами рассмотрен достаточно широко. Так, Q. Umer и его коллеги предложили подход к автоматизации предсказания приоритета дефекта, основанный на оценке эмоционального окраса содержания отчёта о дефекте [17]. Их подход комбинирует методы обработки естественного языка (NLP) и алгоритмы машинного обучения. Это позволило членам команды назначать подходящий уровень приоритета отчётам о дефектах в автоматическом режиме. Результаты показали, что предлагаемый подход превосходит существующие, увеличивая показатель F1-меры более чем на 6%.

М. Михайлов провел исследование, целью которого являлось изучение поведения нейронных сетей при определении приоритета отчётов о дефектах [10]. Внимание в нём было сосредоточено на анализе важности добавления числовых характеристик к текстовым характеристикам путём комбинирования различных типов нейронных сетей. Результаты показали, что добавление численных характеристик позволяет достичь точности (precision) в среднем равной 85.5%.

Z. Lin и её коллеги применили сразу два метода машинного обучения – метод опорных векторов и алгоритм для построения деревьев решений C4.5. Применив полученный классификатор к различным полям отчёта об ошибке, им удалось достигнуть точности в 77.64% [8]. Использованный ими набор данных содержал в себе 2576 отчётов об ошибках.

M. Sharma предложила подход к приоритизации с использованием ещё большего числа средств машинного обучения – это и метод опорных векторов, и наивный байесовский классификатор, и метод k-ближайших соседей. Результаты исследования показали, что точность использованных методов в предсказании приоритета отчётов о дефектах была в среднем выше 70% [14], за исключением байесовского классификатора, являющегося алгоритмически тривиальным для представленной задачи.

В работах, в которых подход к классификации дефектов основан на технологиях глубокого обучения, преобладает использование рекуррентных нейронных сетей. Это связано с тем, что именно данный тип нейронных сетей лучше всего адаптирован для задач текстовой классификации. S. Mani был предложен алгоритм представления отчёта о дефекте с использованием двунаправленной сетевой модели RNN под названием DBRNN-A [9]. В качестве входных данных в этой модели были выбраны два поля, а именно название отчёта о дефекте и его описание. Набор данных, в свою очередь, был взят из систем управления дефектами в ПО с открытым исходным кодом – таких проектов, как Chromium, Mozilla Core и Mozilla Firefox. Результаты продемонстрировали, что DBRNN-A достигает улучшения производительности на 37-43% при сравнении с другими классификаторами на основе алгоритмов глубокого обучения.

W. Ramay с соавторами описал подход к приоритизации дефектов, основанный на использовании глубокой нейронной сети. В качестве входных данных для модели при этом предлагается использовать исторические данные отчётов об ошибках. Результаты показали, что улучшение F-меры при использовании такого подхода составляет 7.90% [13].

В таблице 2 резюмируются вышеописанные работы по таким критериям, как производительность, использованные характеристики входных данных и используемые классификаторы.

Таблица 2 – Краткое изложение подходов к определению приоритета дефектов, доступных в литературе

Производительность	Используемые характеристики	Классификатор(ы)
Улучшает F-меру более чем на 6%	Summary	NLP + ML алгоритм
Точность = 85.5%	Анализ тональности текста и текстовый анализ	MLP, CNN, LSTM
Точность = 77.64%	--/--	SVM, C4.5
Выше 70%, за исключением байесовского классификатора	--/--	SVM, NB, KNN
Улучшение = 12-15%, точность = 37-43%	Title, Description	SVM, MNB, косинусный коэффициент
Улучшение F-меры на 7.90%	--/--	CNN, LSTM, MNB, RF

Большинство решений, представленных в научной литературе, хоть и имеют преимущество друг перед другом в вопросах производительности, но незначительное. Это может быть обусловлено как используемыми исходными данными (в большинстве работ наборы данных брались из одних и тех же открытых источников), так и схожестью применяемых подходов. Однако основным – и достаточно серьезным недостатком всех описанных подходов –является отсутствие доступа к их исходному коду для дальнейшего переиспользования. Это, в свою очередь, является одной из главных причин, из-за которой требуется собственноручная реализация алгоритма. В то же время из имеющегося многообразия представленных классификаторов, наибольшую эффективность в применении демонстрируют алгоритмы, основанные на использовании рекуррентных нейронных сетей. В связи с этим, целесообразно взять за основу классификатора именно алгоритмы глубокого обучения.

Выводы и результаты по главе 1

1. В ходе рассмотрения предметной области, была выявлена потребность в автоматизации производственного процесса поддержки

программного обеспечения, а именно – определения приоритета отчётов о дефектах программного обеспечения.

2. При рассмотрении структуры отчётов о дефектах было определено, какие входные параметры целесообразно использовать для прогнозной модели – таковыми оказались параметры Summary, Description и Component.

3. Анализ существующей литературы по представленной проблеме продемонстрировал незначительные различия в производительности описанных алгоритмов машинного обучения, однако наибольшими показателями обладали модели, основанные на глубоком обучении;

4. Отсутствие доступа к исходному коду алгоритмов, представленных в научных работах, обусловливает необходимость самостоятельной реализации классификатора.

Глава 2 Разработка алгоритма приоритизации дефектов

2.1 Математическое описание и сравнительная характеристика рекуррентных нейронных сетей

Рекуррентные нейронные сети (РНС) представляют собой тип искусственных нейронных сетей, использующих последовательные данные или данные, имеющие временную зависимость. Эти алгоритмы глубокого обучения обычно используются для таких задач, как языковой перевод, обработка естественного языка, распознавание речи и добавление субтитров к изображениям. Такой круг применения обусловлен одной из важных особенностей рекуррентных нейронных сетей, а именно – наличием в таких сетях обратных связей, что позволяет с их помощью сохранять информацию и формировать таким образом подобие «памяти». Это позволяет рекуррентным нейронным сетям эффективно работать с данными, в которых важно не только содержание, но и порядок, в котором поступает информация. Такого преимущества лишены классические нейронные сети –

так называемые нейросети прямого распределения – в которых информация передаётся только вперед по сети, от слоя к слою. В рекуррентных сетях обмен происходит обмен информацией между нейронами – так, при получении нового фрагмента входных данных, нейрон получает некоторую информацию о предыдущем состоянии сети. На рисунке 2 приведено сравнение рекуррентной сети (слева) и сети прямого распределения (справа):

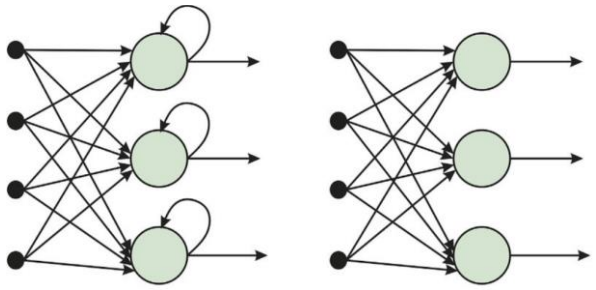


Рисунок 2 – Сравнение рекуррентной и традиционной сетей

Другой отличительной особенностью рекуррентных сетей является то, что они разделяют параметры на каждом уровне сети. В то время, как сети с прямой связью имеют разные веса для каждого узла, рекуррентные нейронные сети имеют один и тот же параметр веса на каждом уровне сети. Тем не менее, эти веса претерпевают корректировку в процессе обратного распространения ошибки и градиентного спуска для того, чтобы облегчить обучение с подкреплением [18].

Рекуррентные нейронные сети используют алгоритм обратного распространения ошибки во времени (BPTT) для определения градиентов, который немного отличается от традиционного обратного распространения ошибки, поскольку он специфичен для последовательных данных. Принцип BPTT такой же, как и при традиционном обратном распространении, где модель обучается, вычисляя ошибки от выходного слоя до входного слоя. Эти вычисления позволяют исследователям соответствующим образом скорректировать параметры модели. BPTT отличается от традиционного подхода тем, что BPTT суммирует ошибки на каждом временном шаге, в то

время как в сетях прямого распределения нет необходимости суммировать ошибки, поскольку они не разделяют параметры на каждом слое сети.

В ходе этого процесса РНС, как правило, сталкивается с проблемой исчезающих градиентов. Она определяется размером градиента, представляющего собой наклон функции потерь вдоль кривой ошибки. Когда градиент становится слишком мал, он продолжает уменьшаться, обновляя весовые параметры до тех пор, пока значение градиента не становится несущественным, то есть опускается до нуля. Когда это происходит, алгоритм перестаёт обучаться. В научной литературе это явление получило название проблемы краткосрочной памяти [2]. Для преодоления этой проблемы были созданы две специализированные версии РНС, управляемые рекуррентные блоки (Gated recurrent unit, GRU) и долгая краткосрочная память (Long short-term memory, LSTM).

Архитектура управляемых рекуррентных блоков представлена на рисунке 3:

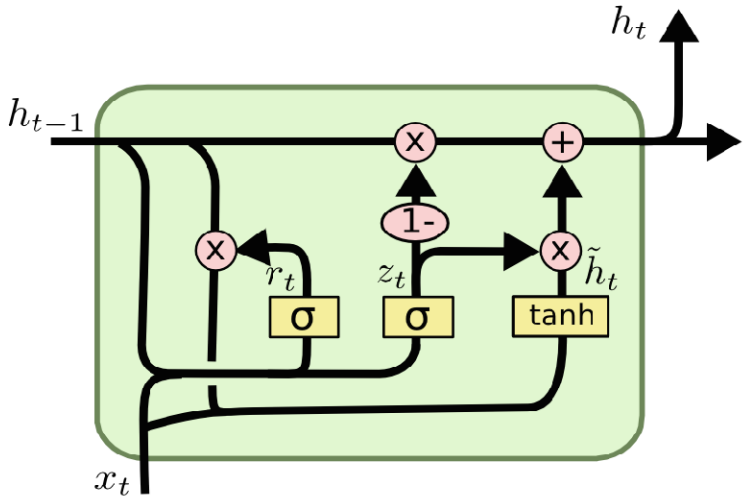


Рисунок 3 – Блок GRU в графическом представлении

Рабочий процесс GRU в целом не отличается от процессов РНС, отличия выявляются в операциях внутри блока GRU. Математически архитектура описывается в виде формул (1-4):

$$z_t = \sigma(W_z \times [h_{t-1}, x_t]) \quad (1)$$

$$r_t = \sigma(W_r \times [h_{t-1}, x_t]) \quad (2)$$

$$\tilde{h}_t = \tanh(W \times [r_t * h_{t-1}, x_t]) \quad (3)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t \quad (4)$$

где z_t – вектор вентиля обновления,

r_t – вектор вентиля сброса,

h_t – выходной вектор, описывающий состояние блока,

x_t – входной вектор,

σ – функция активации на основе сигмоиды,

W_z, W_r – матрицы параметров.

Ключевой особенностью как GRU, так и LSTM является наличие структур, называющихся логическими вентилями. В GRU, упрощённой версии LSTM, применяется два вентиля – вентиль сброса и вентиль обновления, в качестве функции активации использующий гиперболический тангенс и сигмоидную, следовательно, принимая значение от 0 до 1. В ходе вычисления градиент содержит вектор вентиля обновления, что позволяет сети лучше контролировать значение градиента, на каждом шагу используя подходящие обновления параметров вентиля обновления. Наличие активации у вентиля обновления даёт алгоритму возможность решать, какая информация должна быть забыта, а какая нет, и соответствующим образом обновлять параметры модели. Именно за счёт этого модифицированным вариантам РНС удаётся избежать проблемы исчезающих градиентов [3].

LSTM, в свою очередь, обладает более сложной архитектурой. Её графическое представление изображено на рисунке 4:

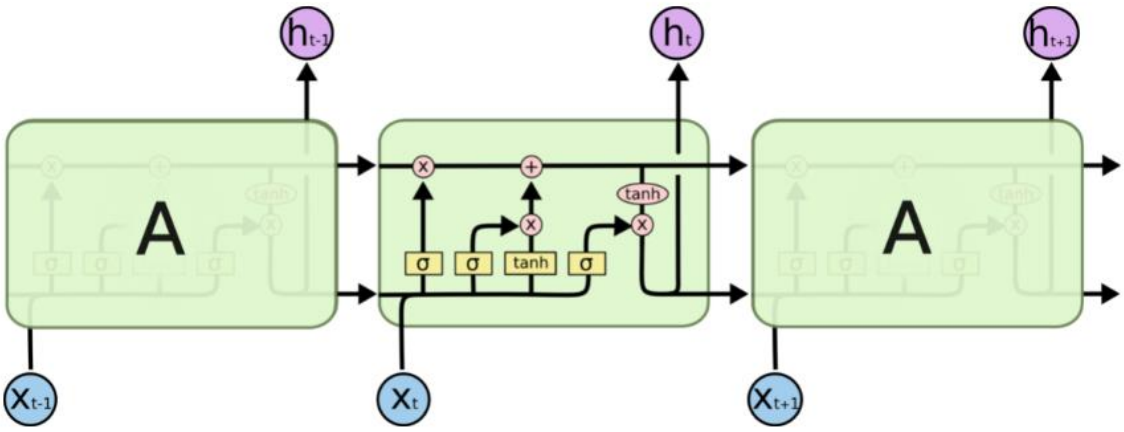


Рисунок 4 – Блок LSTM

Отличие от GRU обнаруживается в том, что логических вентилей у LSTM больше – это входной вентиль, вентиль забывания и выходной вентиль. Кроме того, LSTM имеет более сложную архитектуру памяти в форме разделения внутреннего состояния блока от его вывода, что позволяет выводить данные, полезные для задачи, без необходимости их запоминать. Это, с одной стороны, усложняет как модель, так и вычислительную сложность у LSTM, но при этом позволяет достичь большей производительности при подаче на вход более длинных последовательностей данных [20]. Поскольку данные, которые предлагается использовать в данной работе, состоят из достаточно больших по объёму последовательностей, очевидным становится использование LSTM в качестве текстового классификатора.

Таким образом, в ходе рассмотрения математической модели рекуррентных нейронных сетей, были выявлены практические преимущества их использования, а также объяснена необходимость использования модификаций РНС. В частности, для поставленной задачи оптимальной моделью служит алгоритм LSTM.

2.2 Программная реализация классификатора

Для построения прогнозных моделей в глубоком обучении существует достаточно обширный выбор программного инструментария. К нему относятся такие решения, как TensorFlow, Theano, PyTorch и Keras. При этом, начиная с версии 2.0, TensorFlow является лидирующим инструментом в области глубокого обучения, поскольку он получил полную интеграцию с библиотекой Keras, расширив и без того значительный функционал. Прочие из библиотек, представленные в списке, либо обладают сравнительно неудобным программным интерфейсом (API), что снижает скорость разработки, либо не имеют требуемого функционала, в частности, поддержки рекуррентных нейронных сетей. В связи с этим, наиболее оптимальным в рамках данной работы является применение связки TensorFlow и Keras для построения прогнозной модели. Одним из преимуществ в пользу использования такой инструментальной связки является возможность написания программного кода на Python, языке, в настоящее время являющимся самым используемым в области машинного обучения.

Как уже отмечалось в первой главе, в качестве входных параметров предполагается использовать несколько полей отчёта о дефектах, а именно – Summary, Description и Component. В TensorFlow Keras для работы с моделями существует два вида API – последовательное (Sequential) и функциональное (Functional). Для поддержки множественного набора входных данных будет применяться именно функциональное API в связи с тем, что оно способно работать с моделями нелинейной топологии, к которым относятся и модели с несколькими входными признаками [16].

Структура предлагаемого решения предполагает разбиение на две фазы – в первой из которых осуществляется сбор данных, кодирование прогнозной метки (то есть поля Priority), а также предварительная обработка текстовых полей, во второй – выбор признаков, подготовка обучающего и тестового наборов данных, построение и обучение модели, и, наконец, процесс её оценки. В приложении А описанная структура представлена в виде рисунка,

в данном разделе будет осуществлен последовательный разбор всех этапов, указанных в обозначенном приложении.

При экспорте данных в формате csv, в системе отслеживания ошибок JIRA есть возможность указать разделитель колонок в таблице, как представлено на рисунке 5. Поскольку текстовые данные зачастую содержат знаки препинания, такие как запятая, нужно изменить разделитель по умолчанию на точку с запятой, для того чтобы данные из csv-файла корректно экспортировались.

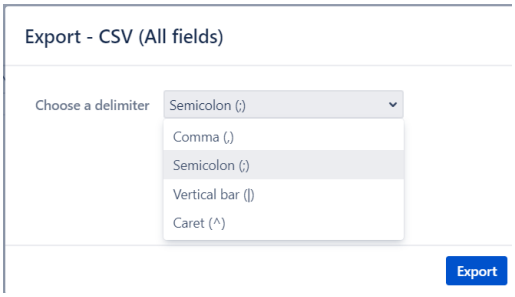


Рисунок 5 – Окно выбора разделителя при экспорте csv-файла

После загрузки данных из файла средствами Pandas, осуществляется разбиение данных на набор признаков и набор меток, предстающими в виде переменных X и Y соответственно. Для набора меток с помощью класса LabelEncoder библиотеки sklearn производится кодирование признаков. Стоит отметить, что класс OneHotEncoder из той же библиотеки не подойдет для данной задачи, поскольку он предназначается для кодирования таких категориальных данных, которые не связаны определенным порядком или иерархией.

При компиляции обучающейся модели одним из обязательных параметров является функция потерь. Поскольку перед алгоритмом стоит задача мультиклассовой классификации, в качестве функции потерь необходимо использовать кросс-энтропию, или логарифмическую функцию, в библиотеке TensorFlow Keras представленную как categorical_crossentropy [15]. Это, в свою очередь, обязывает при работе с набором меток

(переменной y) привести вектор численных значений (что уже сделано средствами LabelEncoder) в бинарное матричное представление. Для этого используется функция to_categorical из библиотеки TensorFlow Keras.

Следующим этапом является предобработка текста (нормализация) и удаление стоп-слов. Для этого была реализована функция preprocess_text, код которой представлен на рисунке 6:

```
def preprocess_text(sen):
    # Удаление URL-адресов
    sentence = re.sub(r'(https?:\/\/)?(www\.)?[-a-zA-Z0-9@:%._\+~#=]{2,256}\.[a-z]{2,6}\b([-a-zA-Z0-9@:%_\+.~#()?&//=]*)', ' ', sen)

    # Удаление стоп-слов
    sentence = ' '.join([word for word in sentence.split() if word not in STOPWORDS])

    # Удаление фрагментов текста, характерного для отчётов о дефектах
    sentence = re.sub('{code}[^>]+{code}', '', sentence)
    sentence = re.sub('{code:java}[^>]+{code}', '', sentence)
    sentence = re.sub('{noformat}[^>]+{noformat}', '', sentence)
    sentence = re.sub('!(.*)!', '', sentence)

    # Удаление пунктуации и чисел
    sentence = re.sub('[^a-zA-Z]', ' ', sentence)

    # Удаление единичных символов, встречающихся в тексте
    sentence = re.sub(r"\s+[a-zA-Z]\s+", ' ', sentence)

    # Удаление двойных и более пробелов
    sentence = re.sub(r'\s+', ' ', sentence)

    return sentence
```

Рисунок 6 – Программный код функции для предобработки текста

Предобработка текста в машинном обучении является важным этапом подготовки данных перед началом обучения модели. Помимо стандартных операций, производимых над текстом в рамках подготовки, были добавлены дополнительные регулярные выражения, удаляющие содержимое разметки в отчётах о дефектах. В имеющемся наборе данных для того, чтобы сообщить нужную разработчикам информацию об ошибке, такую как трассировка стека, её заключают в определенные теги для удобства восприятия текста. Содержимое этих тегов в ходе предобработки текста удаляется, поскольку, например, список вызовов методов, приведших к исключению, не несёт никакой смысловой нагрузки для модели.

После применения вышеописанной функции к тренировочной и тестовой выборкам каждого из входных параметров, необходимо привести эти признаки к численной форме. Для этого используется класс Tokenizer из библиотеки TensorFlow Keras, позволяющий векторизовать корпус текста, преобразуя каждый текст в составе корпуса в последовательность целых

чисел, каждое из которых, в свою очередь, является индексом токена в словаре.

При этом существует проблема неравномерности текстовых данных. Это означает, что разные отчёты о дефектах содержат разные объёмы текстовой информации – следовательно, входные векторы при загрузке в модель будут иметь различную размерность, вследствие чего интерпретатор Python терминирует выполнение программы с ошибкой IndexError. Для того, чтобы избежать такого поведения, используется метод pad_sequences класса Tokenizer, который приводит все числовые последовательности к одной, фиксированной размерности путём добавления достаточного количества нулей в конец последовательности, либо удаляя лишние данные. У каждого из входных признаков при этом определяется своя размерность, поскольку, например, поле Summary содержит меньше текста, чем поле Description.

Важнейшим элементом алгоритма является построение обучаемой модели. Как уже было сказано ранее, для этого используется функциональное API, позволяющее строить модели с множественным набором входных данных. В приложении Б можно ознакомиться с графическим представлением разработанной модели, сгенерированной средствами TensorFlow Keras. Программное описание модели, в свою очередь, представлено на рисунке 7:

```
175  def create_model(maxlen, maxlen_summary, maxlen_component):
176      # Инстанциируем тензоры
177      input_1 = Input(shape=(maxlen,))
178      input_2 = Input(shape=(maxlen_summary,))
179      input_3 = Input(shape=(maxlen_component,))
180
181      # Инициализируем необучаемые Embedding-слои, в которых весовые коэффициенты вносятся из GloVE,
182      # а также слой LSTM
183      embedding_layer_1 = Embedding(vocab_size_description, 100, weights=[embedding_matrix_description], trainable=False)(input_1)
184      LSTM_Layer_1 = LSTM(128, recurrent_dropout=0.3)(embedding_layer_1)
185      embedding_layer_2 = Embedding(vocab_size_summary, 100, weights=[embedding_matrix_summary], trainable=False)(input_2)
186      LSTM_Layer_2 = LSTM(128, recurrent_dropout=0.3)(embedding_layer_2)
187      embedding_layer_3 = Embedding(vocab_size_component, 100, weights=[embedding_matrix_component], trainable=False)(input_3)
188      LSTM_Layer_3 = LSTM(128, recurrent_dropout=0.3)(embedding_layer_3)
189
190      # Объединяем слои LSTM в один
191      concat_layer = Concatenate()([LSTM_Layer_1, LSTM_Layer_2, LSTM_Layer_3])
192      dense_layer_1 = Dense(128, activation='relu')(concat_layer)
193      output = Dense(5, activation='softmax')(dense_layer_1) # 5 блоков означает 5 различных выводов
194      model = Model(inputs=[input_1, input_2, input_3], outputs=output)
195
196      opt = tf.keras.optimizers.Adam(learning_rate=0.001)
197      model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy', tf.keras.metrics.Recall(), tf.keras.metrics.Precision()])
198      print(model.summary())
199
200      return model
```

Рисунок 7 – Построение модели нейросети

Для каждого из входных признаков в прогнозной модели применяется набор слоев, из которых первым является InputLayer. Он необходим для инстанциации тензоров – структур данных в TensorFlow, технически представляющих из себя массив, количество измерений в котором составляет от 0 до N. В данном случае размерность тензоров, представляющих текстовые поля Description, Summary и Component равняется максимально возможному размеру этих полей, то есть длине самого большого текста для каждого из представленных признаков.

Следующим шагом в построении модели является определение Embedding-слоя, являющегося первым функциональным слоем разрабатываемой модели. Поступающие на вход этот слоя данные представляют собой последовательности целых чисел, обозначающие определенные слова. Embedding-слой преобразует каждое из чисел i в i-ю строчку матрицы весов векторного представления слов. Результирующий вектор представляет собой плотный вектор с действительными значениями. Фиксированная длина векторов слов позволяет лучше представлять слова в численной форме, при этом позволяя снизить размерность входных данных. Это, в свою очередь, способствует снижению ресурсоёмкости выполняемой программы, поскольку снижение размерности данных позволяет снизить сложность вычислений, применяемых к этим данным. Результатом его работы, в случае использования обучаемого Embedding-слоя, является сформированная матрица весов векторного представления слов. Однако в рамках данной работы обучать данный слой нецелесообразно ввиду следующих причин [12]:

– снижение скорости обучения модели в целом – число обучаемых параметров увеличивается в случае, если обучать слой с нуля, что ведёт к замедлению процесса обучения;

– разреженность обучающих данных – поскольку применяемый в качестве входных данных набор не содержит в себе объёмного словаря уникальных слов, матрица весов, полученная в результате обучения Embedding-слоя, не будет достаточно корректно представлять значение того или иного слова. Это, в свою очередь, приведёт к снижению точности прогнозирования выходного значения модели, то есть метки.

В связи с этим, было принято решение воспользоваться заранее обученной матрицей векторного представления слов. Одним из примеров заранее обученных матриц являются векторные представления, формируемые алгоритмом GloVe. Он представляет собой алгоритм обучения без учителя, предназначенный для получения векторных представлений слов. Обучение выполняется на агрегированной глобальной статистике совместной встречаемости слова и слова из корпуса [12]. Такая статистика берётся на основе данных Wikipedia и Gigaword. За счёт этого заранее обученные векторные представления имеют очень обширный словарный запас, вплоть до 2,2 млн. слов. Подобный объём словарного запаса позволяет гораздо лучше представить значение слова в матрице векторных представлений, чем это было бы возможно при обучении на тестовой выборке. На рисунке 8 изображен программный код для формирования матриц весов для каждого из входных признаков:

```
glove_file = open('glove.6B.100d.txt', encoding="utf8")

for line in glove_file:
    records = line.split()
    word = records[0]
    vector_dimensions = asarray(records[1:], dtype='float32')
    embeddings_dictionary[word] = vector_dimensions

glove_file.close()

embedding_matrix_description = zeros((vocab_size_description, 100))
for word, index in tokenizer.word_index.items():
    embedding_vector = embeddings_dictionary.get(word)
    if embedding_vector is not None:
        embedding_matrix_description[index] = embedding_vector

embedding_matrix_summary = zeros((vocab_size_summary, 100))
for word, index in tokenizer.word_index.items():
    embedding_vector = embeddings_dictionary.get(word)
    if embedding_vector is not None:
        embedding_matrix_summary[index] = embedding_vector

embedding_matrix_component = zeros((vocab_size_component, 100))
for word, index in tokenizer.word_index.items():
    embedding_vector = embeddings_dictionary.get(word)
    if embedding_vector is not None:
        embedding_matrix_component[index] = embedding_vector
```

Рисунок 8 – Формирование матриц весов на основе заранее обученной матрицы векторного представления слов

Необходимость создания матриц весов для входных признаков по отдельности обусловлена тем, что они имеют различный объём словарного запаса. Это видно по отладочному выводу программы на рисунке 9:

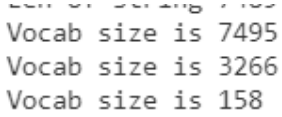


Рисунок 9 – Вывод объёма словарного запаса для входных признаков

Так, наибольшим объёмом словарного запаса обладает признак Description, поскольку он содержит наибольший объём текстовой информации. Далее по убыванию следуют признаки Summary и Component. Для того, чтобы воспользоваться заранее обученной матрицей, в каждом из созданных Embedding-слоев добавляется аргумент weights, в котором

задается переменная, хранящая сформированную матрицу. В то же время необходимо заморозить возможность изменения весов с помощью аргумента trainable со значением false – иначе использование вышеупомянутой матрицы теряет смысл, поскольку веса при обучении будут корректироваться.

Следующим слоем в прогнозной модели является слой рекуррентной нейронной сети, LSTM. Внутри него происходит основная работа по прогнозированию метки. Основным параметром здесь является количество блоков внутри слоя – оно определяет размерность выходных данных и число параметров в слое LSTM. Оно задано таким образом, чтобы размерность выходных данных равнялась размерности выходных данных – таким образом для более сложных по наполнению признаков нейросеть будет запоминать более сложные шаблоны в процессе работы. Дополнительным используемым параметром служит recurrent_dropout – значение, характеризующее исключение определенного процента случайных нейронов в ходе обучения нейронной сети. Этим параметром в Keras реализовано рекуррентное исключение – метод регуляризации нейронных сетей, предназначенный для уменьшения переобучения сети вышеописанным способом. Его использование обусловлено необходимостью оптимизации модели с целью повышения её точности.

Далее выходные данные всех трёх LSTM-слоёв объединяются с помощью слоя конкатенации в один вектор. Это делается для того, чтобы привести данные к одной размерности. Затем полученный вектор подаётся на вход Dense слоя – в котором осуществляется применение функции активации ReLU (Rectified Linear Unit). Её использование позволяет существенно повысить скорость сходимости стохастического градиентного спуска (до 6 раз, согласно Krizhevsky A. [7]) по сравнению с сигмоидой и гиперболическим тангенсом. Это объясняется тем, что ReLU имеет линейный характер и не подвержен насыщению. В работе A.M. Javid, посвященной изучению использования Dense слоя с ReLU, также заявляется, что

применение слоя в такой конфигурации позволяет увеличить производительность прогнозной модели [6].

Последним слоем прогнозной модели является ещё один Dense слой с функцией активации softmax. Данная функция активации выводит вектор значений, суммирующихся до единицы. Эти значения, в свою очередь, интерпретируются как вероятности отношения к тому или иному классу. Для корректного вывода вероятностей Dense слою задаётся пять блоков (по количеству уникальных значений в списке прогнозных меток). Использование этой функции активации необходимо, поскольку она специально предназначена для задач мультиклассовой классификации, к которой относится представленная в работе задача.

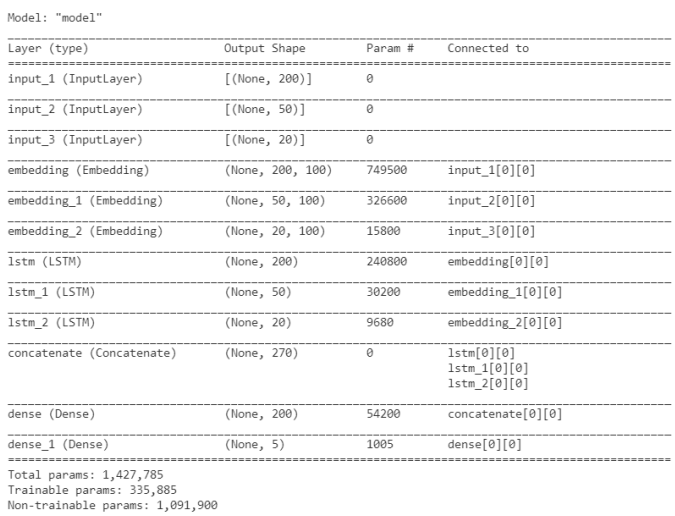
После того, как модель сформировалась с точки зрения топологии, необходимо её скомпилировать. При компиляции задаются следующие параметры:

– функция потерь – как уже ранее было отмечено, используется categorical_crossentropy в соответствии с типом решаемой задачи;

– метод оптимизации – предназначен для оптимизации градиентного спуска, для применения выбран оптимизатор Adam, поскольку является наиболее распространённым и применимым оптимизатором в большинстве задач [17];

– метрики – необходимы для оценки производительности алгоритма после компиляции, в качестве метрик применяются Accuracy, Recall, Precision.

В результате компиляции происходит вывод сформированной модели в консоль. Её текстовое представление изображено на рисунке 9:



```
Model: "model"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_1 (InputLayer)            [(None, 200)]        0
__________________________________________________________________________________________________
input_2 (InputLayer)            [(None, 50)]         0
__________________________________________________________________________________________________
input_3 (InputLayer)            [(None, 20)]         0
__________________________________________________________________________________________________
embedding (Embedding)           (None, 200, 100)     749500      input_1[0][0]
__________________________________________________________________________________________________
embedding_1 (Embedding)         (None, 50, 100)      326600      input_2[0][0]
__________________________________________________________________________________________________
embedding_2 (Embedding)         (None, 20, 100)      15800       input_3[0][0]
__________________________________________________________________________________________________
lstm (LSTM)                     (None, 200)          240800      embedding[0][0]
__________________________________________________________________________________________________
lstm_1 (LSTM)                   (None, 50)           30200       embedding_1[0][0]
__________________________________________________________________________________________________
lstm_2 (LSTM)                   (None, 20)           9680        embedding_2[0][0]
__________________________________________________________________________________________________
concatenate (Concatenate)       (None, 270)          0           lstm[0][0]
                                                                 lstm_1[0][0]
                                                                 lstm_2[0][0]
__________________________________________________________________________________________________
dense (Dense)                   (None, 200)          54200       concatenate[0][0]
__________________________________________________________________________________________________
dense_1 (Dense)                 (None, 5)            1005        dense[0][0]
==================================================================================================
Total params: 1,427,785
Trainable params: 335,885
Non-trainable params: 1,091,900
```

Рисунок 9 – Текстовое представление скомпилированной модели

Здесь даётся информация о каждом из слоёв модели, включая размерность выходных данных, количество обучаемых параметров, а также характеристика связности слоёв друг с другом. Следует отметить, что значительная часть параметров определяется как необучаемая – это обусловлено тем, что Embedding-слои были отмечены как необучаемые, при этом имея в себе большое число параметров. Именно поэтому функциональными, с точки зрения алгоритма, являются LSTM слои, а также Dense-слои, применяющие функции активации.

Обучение модели предваряет задание следующих обязательных аргументов для метода model.fit:

– значение X – набор входных признаков, в данном случае три списка с набором тренировочных данных, сформированных заранее;

– значение Y – набор меток, список со значениями приоритета, приведённый к численной форме;

– batch_size – количество обучающих примеров за одну итерацию (эпоху), проходящих через нейронную сеть, значение установлено в 256;

– epochs – количество итераций, значение установлено в 20, поскольку увеличение числа итераций больше этого значения не привело к существенному улучшению результатов прогнозирования.

В процессе обучения Keras выводит данные по производительности алгоритма в консоль. Пример такого вывода представлен на рисунке 10:

```
Epoch 18/20
18/18 [==============================] - 23s 1s/step - loss: 1.1572 - accuracy: 0.4569 - recall: 0.1139 - precision: 0.5129
.0352 - val_precision: 0.5000
Epoch 19/20
18/18 [==============================] - 23s 1s/step - loss: 1.1561 - accuracy: 0.4474 - recall: 0.1122 - precision: 0.5146
.0607 - val_precision: 0.4759
Epoch 20/20
18/18 [==============================] - 23s 1s/step - loss: 1.1496 - accuracy: 0.4505 - recall: 0.0882 - precision: 0.5095
.0273 - val_precision: 0.4429
45/45 [==============================] - 3s 59ms/step - loss: 1.2132 - accuracy: 0.4258 - recall: 0.0239 - precision: 0.5075
Test Loss: 1.2131919860839844
Test Accuracy: 0.4257565140724182
```

Рисунок 10 – Вывод данных о прогрессе обучения в консоль

Поскольку условия работы с алгоритмом предполагают его частое использование, необходимо, чтобы при работе с ним определение приоритета не требовало переобучения модели – в первую очередь из-за временных издержек. Для этого обученную модель можно сохранить с помощью метода model.save, предоставляемого TensorFlow Keras. При таком сохранении модель записывается в специальном формате – TensorFlow SavedModel, который содержит в себе:

– конфигурацию модели (то есть её топологию),

– веса модели, вычисленные в ходе обучения,

– состояние оптимизатора модели.

Набор этих данных позволяет достаточно быстро применять модель для определения приоритета, что положительно сказывается на скорости работы с алгоритмом.

Таким образом, в данном подразделе был подробно описан подход к реализации прогнозной модели с учетом рекомендаций, даваемых в научной литературе, посвященной машинному обучению. В следующем подразделе рассматривается реализация прототипа программного интерфейса (API) для взаимодействия с разработанным алгоритмом приоритизации дефектов.

2.3 Прототипирование программного интерфейса для алгоритма

Для того, чтобы облегчить тестирование алгоритма и продемонстрировать прототип для его использования, было решено разработать для него программный интерфейс (API). Наиболее логичным в контексте использования Python в качестве основного языка разработки является применение Python-библиотек, предназначенных для построения программных интерфейсов. Из существующих библиотек для данной задачи лучше всего подойдёт Flask, поскольку данный веб-фреймворк, обладая минимальным функционалом, позволяет быстрее всего реализовать RESTful API с минимальными трудозатратами [19]. Кроме этого, реализованный интерфейс имеет возможность удобного встраивания в уже существующие системы, что даёт дополнительное преимущество при интеграции.

Предлагаемый веб-интерфейс по своей структуре достаточно прост – при запуске веб-приложения осуществляется регистрация маршрута с конечной точкой /predict. По этому маршруту принимается POST-запрос, из которого извлекаются параметры, соответствующие входным признакам для прогнозирования. На рисунке 11 представлена реализация данного маршрута:

```
46  @app.route('/predict', methods=['POST'])
47  def predict():
48      data = request.get_json()
49      summary = data['summary']
50      description = data['description']
51      component = data['component']
```

Рисунок 11 – Регистрация маршрута и определение принимаемых им данных

После извлечения полей из JSON запроса, текстовые данные проходят предобработку теми же средствами нормализации текста, что описаны в предыдущем подразделе. Из нормированных данных формируется список, который подаётся методу model.predict из библиотеки TensorFlow.

Перед запуском веб-приложения осуществляется загрузка обученной модели средствами TensorFlow. С одной стороны, это позволяет существенно сократить время, затрачиваемое на обработку POST-запроса, с помощью которого выполняется определение приоритета. С другой стороны, для добавления новых данных в модель требуется её переобучение, что в свою очередь ограничивает гибкость предлагаемого алгоритма.

Для отправки запроса к веб-приложению можно использовать различные средства работы с REST-запросами – это и утилита curl в комплектации unix-подобных систем, и Windows-приложения для работы с API, такие как Postman. На рисунке 12 изображен пример отправки запроса на созданный маршрут:

Рисунок 12 – Отправка запроса на прогнозирование приоритета

В теле запроса передаются текстовые значения признаков, подаваемых на вход прогнозной модели в JSON формате – это summary, description и component. Сервер, в свою очередь, отвечает на запрос вектором вероятностей – численными значениями, характеризующими вероятность принадлежности дефекта к тому или иному приоритету. Вектор является результатом вычислений метода model.predict. Ввиду того, что API разрабатывался как прототип, декодирование меток в ответе сервера не осуществляется, однако определить приоритет по представленному списку

значений несложно – вероятности выводятся в порядке возрастания приоритета (то есть Normal, Minor, Major, Critical, Blocker).

Конечным результатом во второй главе стала программная реализация алгоритма для приоритизации дефектов программного обеспечения. Для удобства его тестирования и демонстрации возможностей по применению алгоритма был реализован программный интерфейс в виде прототипа веб-приложения, позволяющего с помощью REST-запроса вводить в алгоритм входные признаки и получать в ответ от сервера прогнозную метку, то есть определенный приоритет для отчёта о дефекте.

Выводы и результаты по главе 2

1. В ходе анализа архитектуры рекуррентных нейронных сетей, была приведена аргументация с математической точки зрения, согласно которой при реализации алгоритма требуется использовать модификацию РНС, сеть долгосрочной памяти (LSTM).

2. Приведено детальное описание процесса реализации алгоритма в виде программного обеспечения с учётом рекомендаций, даваемых в профильных источниках для улучшения производительности прогнозной модели.

3. Для нужд тестирования и для демонстрации возможностей по интеграции разработанного алгоритма в существующие решения был разработан прототип программного интерфейса (API) на основе веб-приложения.

Глава 3 Тестирование разработанного программного обеспечения

3.1 Описание разработанного программного обеспечения

Результатом программной реализации алгоритма приоритизации стала следующая структура модулей:

– train-lstm.py – данный модуль содержит в себе программный код непосредственно алгоритма приоритизации, реализация которого была описана в подразделе 2.2;

– predict-lstm.py – в этом модуле находится реализация программного интерфейса для алгоритма классификатора с использованием веб-фреймворка Flask.

Перед началом работы с вышеперечисленными модулями, необходимо установить зависимости, необходимые для работы приложения. Для этого можно использовать систему управления пакетами в Python, pip. В приложении В, помимо программного кода модулей, представлен файл requirements.txt, в котором перечислены требуемые зависимости. Для их установки необходимо ввести команду «pip install -r requirements.txt», в результате чего система управления пакетами автоматически установит перечисленные в текстовом файле зависимости.

Для того, чтобы произвести процесс определения приоритета программного дефекта, первоначально нужно сформировать и обучить прогнозную модель. Для её генерации нужно запустить модуль train-lstm.py через интерпретатор Python с ключом «-i», указав в качестве аргумента наименование csv-файла с набором данных, который будет использоваться для обучения. Сигналом об окончании обучения служит вывод в консоль значений функции потерь и точности обученного классификатора. Также в результате работы данного модуля будет создана директория с файлами, представленными на рисунке 13:

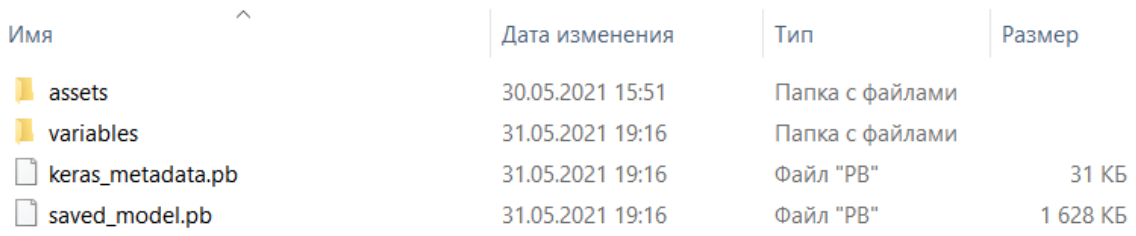


Рисунок 13 – Сохраненные файлы обученной модели

Процесс работы с классификатором начинается с запуска Flask-сервера. Для этого необходимо запустить модуль predict-lstm.py с помощью интерпретатора Python без каких-либо дополнительных аргументов. После этого веб-приложение начинает принимать запросы на порт 5000, по умолчанию используемый в Flask. Пример отправки запроса на конечную точку представлен в подразделе 2.3 данной работы.

3.2 Тестирование и оценка эффективности разработанного классификатора

Производительность и эффективность алгоритмов классификации в машинном обучении оценивается с использованием таких метрик, как accuracy (доля правильных ответов алгоритма), precision (точность) и recall (чувствительность). Рассмотрим каждую из метрик подробнее.

Метрика accuracy, несмотря на то что является наиболее очевидной для использования при оценке качества классификации, на практике оказывается бесполезной в задачах с несбалансированными классами, к которой относится проблема приоритизации дефектов – в этом заключается так называемый парадокс точности [1]. Так, при прогнозировании метки класса, представляющего в наборе данных большинство, на основе большого количества «корректно» определенных меток метрика демонстрирует высокий уровень точности классификации, что на практике оказывается неверно.

Для преодоления вышеописанной проблемы существуют метрики точности и чувствительности, которые в отличие от accuracy, учитывают вклад каждого из классов. Для оценки качества работы алгоритма на каждом из классов по отдельности введём метрики precision (точность) и recall (чувствительность), представленные в виде формул (5-6):

$$precision = \frac{TP}{TP + FP} \tag{5}$$

$$recall = \frac{TP}{TP + FN} \tag{6}$$

где TP – True Positive, или истинно положительные, классификатор верно отнёс объект к рассматриваемому классу,

FP – False Positive, также ошибка первого рода, обозначает, что классификатор неверно отнёс объект к рассматриваемому классу,

FN – False Negative, или ошибка второго рода, классификатор неверно утверждает, что объект не принадлежит к рассматриваемому классу.

Precision интерпретируется как доля объектов, названных классификатором положительными и при этом действительно являющимися положительными, в то время как Recall показывает, какую долю объектов положительного класса из всех объектов положительного класса нашёл алгоритм. Благодаря использованию метрики Precision все объекты не записываются в один класс, что в противном случае привело бы к росту ложноположительных определений приоритета. Recall демонстрирует способность алгоритма обнаруживать тот или иной класс объектов вообще, в то время как Precision – способность отличать этот класс от других классов.

Для обучения модели набор данных был разделен в следующем соотношении: 80% данных было отнесено к обучающему множеству, в то время как 20% было отнесено к тестовой выборке. При этом половина

тестовой выборки была использована в качестве третьего набора наблюдений, проверочного, используемого для контроля за обучением модели. Для графического вывода метрик использовалась библиотека matplotlib. Построенные ей графики представлены на рисунках 14-15:

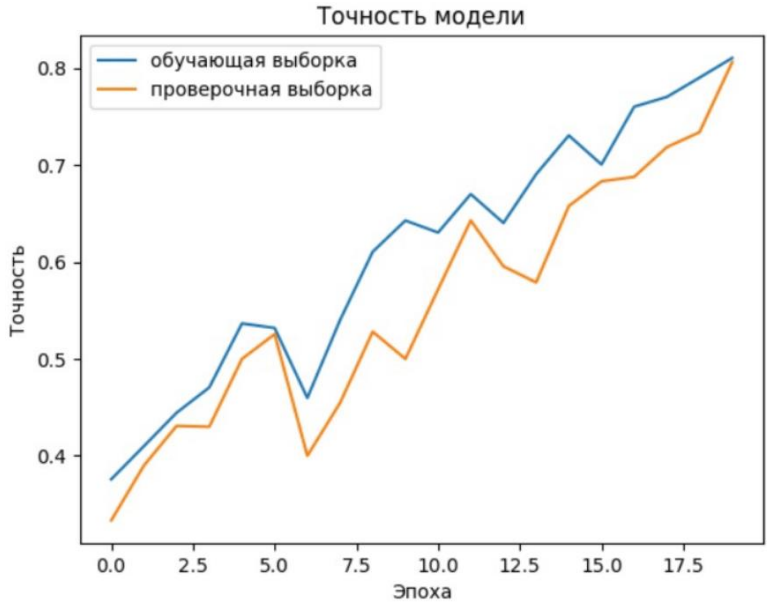


Рисунок 14 – График точности модели

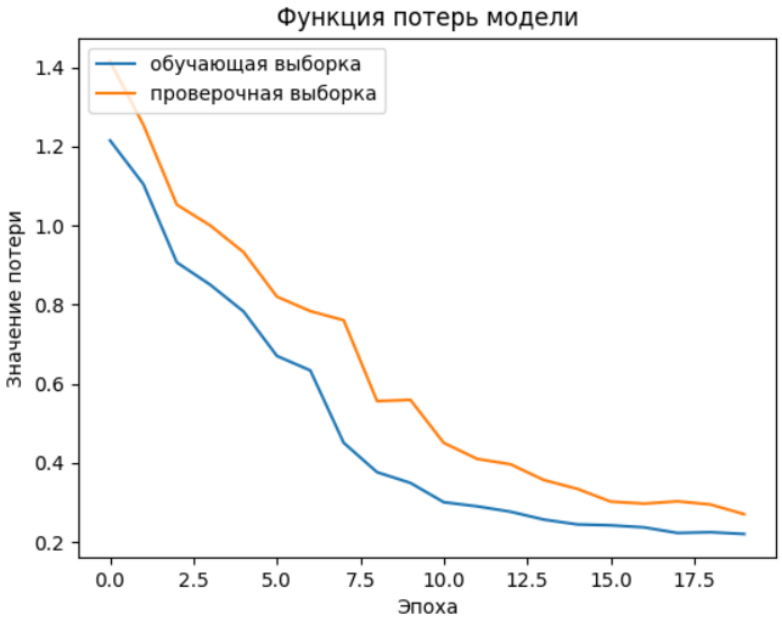


Рисунок 15 – График функции потерь

На данных графиках представлены средневзвешенные значения метрики точности и функции потерь для каждого из классов, представленных в наборе данных. В среднем, модель достигает точности в 81-83% как на обучающей выборке, так и на проверочной. В то же время функция потерь на всем протяжении обучения минимизируется и характеризуется такой же сходимостью графиков, как в случае с метрикой точности. Это означает, что модель корректно обучается, не допуская явления переподгонки, при котором построенная модель хорошо объясняет примеры из обучающей выборки, но плохо работает на данных из тестовой выборки [4].

Для того, чтобы подвести итоги по эффективности алгоритма, рассмотрим также метрику чувствительности, график которой представлен на рисунке 16:

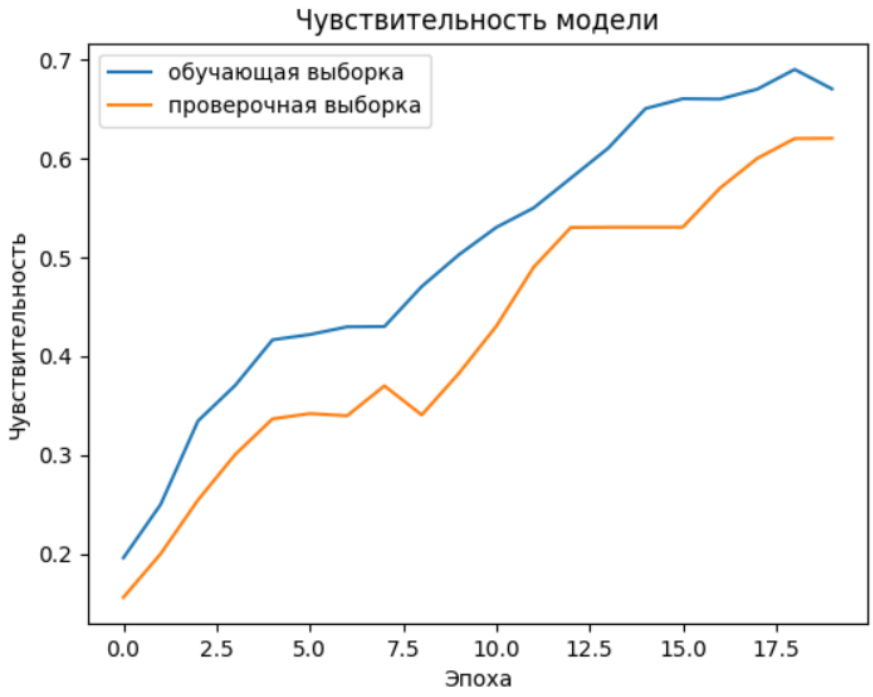


Рисунок 16 – График чувствительности модели

Как видно из представленного выше рисунка, значение метрики чувствительности на проверочной выборке достигает 60%. Более низкие значения как по обучающей, так и по проверочной выборке вкупе с

рассмотренным ранее графиком точности свидетельствуют о том, что для разработанной модели минимизация ошибки первого рода играет большую роль, нежели минимизация ошибки второго рода. Другими словами, более высокие значения точности в сравнении с чувствительностью характеризуют большую склонность модели к выработке ложноотрицательных результатов, чем ложноположительных. Для задачи приоритизации дефектов такой результат более предпочтителен, поскольку осуществляется многоклассовая, а не бинарная классификация, и существует определенная градация приоритетов, допускающая возможность отклонения от однозначно корректного ответа. Идеальным результатом для прогнозирования были бы высокие значения метрик как точности, так и чувствительности, однако на практике, в случае несбалансированного набора данных, необходимо идти на компромисс между вышеупомянутыми метриками.

Таким образом, в результате тестирования алгоритма была выявлена его пригодность к использованию в задачах классификации дефектов со сравнительно высокими показателями метрик точности и чувствительности в 80 и 60% соответственно.

Выводы и результаты по главе 3

1. Для разработанного программного обеспечения дано подробное описание работы с алгоритмом с точки зрения пользователя.
2. В результате рассмотрения существующих метрик для алгоритмов классификации было выявлено, что метрика accuracy не подходит для определения эффективности алгоритма из-за парадокса точности.
3. В ходе анализа метрик, полученных после обучения модели, таких как точность и чувствительность, было определено, что модель имеет склонность к выработке ложноотрицательных результатов – тем не менее, было отмечено, что условия задачи приоритизации допускают такой перекос.

Заключение

В результате выполнения данной выпускной квалификационной работы была осуществлена реализация алгоритма для приоритизации дефектов программного обеспечения. В ходе его разработки были сделаны следующие выводы:

1. Рассмотрение предметной области, то есть вопросов разработки и сопровождения ПО, показало наличие зависимости перечисленных процессов от человеческого фактора, что обусловливает потребность в автоматизации таких процессов. Поэтому использование технологий машинного обучения в таких областях является перспективной и актуальной задачей.

2. В ходе анализа научных источников было выявлено, что несмотря на сравнительно высокую эффективность большинства используемых для задачи приоритизации классификаторов, наиболее эффективным из них оказались алгоритмы, основанные на использовании рекуррентных нейронных сетей.

3. Был проведён сравнительный анализ существующих РНС, из которых ввиду эффективности на больших объёмах текста был выбран LSTM.

4. В ходе реализации ПО было приведено последовательное описание алгоритма и процесса его разработки на языке Python.

5. Предложенное ПО было протестировано на предмет эффективности с помощью общеиспользуемых метрик для оценки качества классификации, в результате чего был сделан вывод о пригодности обученной модели к выполнению задачи приоритизации ввиду высоких значений метрик.

Перечисленные выше выводы позволяют заключить, что алгоритм в перспективе может успешно использоваться в качестве компонента систем, применяемых для автоматизации сопровождения ПО.

Список используемой литературы и используемых источников

1. Accuracy Paradox [Электронный ресурс] : Wikipedia. URL: https://en.wikipedia.org/wiki/Accuracy_paradox (дата обращения: 4.05.2021).

2. Aggraval M., Murty M.N. Machine Learning in Social Networks: Embedding Nodes, Edges, Communities, and Graphs // Google Книги. URL: https://cutt.ly/OnupXAf (дата обращения: 22.03.2021).

3. Arbel N. How LSTM networks solve the problem of vanishing gradients //
Data Driven Investor. URL: https://medium.datadriveninvestor.com/how-do-lstm-networks-solve-the-problem-of-vanishing-gradients-a6784971a577 (дата обращения: 26.03.2021).

4. Brownlee J. How to Diagnose Overfitting and Underfitting of LSTM models [Электронный ресурс] // URL: https://machinelearningmastery.com/diagnose-overfitting-underfitting-lstm-models/ (дата обращения: 7.05.2021).

5. Giordano D. 7 tips to choose the best optimizer [Электронный ресурс] // Towards Data Science. URL: https://towardsdatascience.com/7-tips-to-choose-the-best-optimizer-47bb9c1219e (дата обращения: 26.04.2021).

6. Javid A.M., Das S., Skoglund M., Chatterjee S. A ReLU Dense Layer to Improve the Performance of Neural Networks [Электронный ресурс] // arXiv.org. URL: https://arxiv.org/pdf/2010.13572.pdf (дата обращения: 22.04.2021).

7. Krizhevsky A., Sutskever I., Hinton G.E. ImageNet Classification with Deep Convolutional Neural Networks [Электронный ресурс] // URL: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf (дата обращения: 15.04.2021).

8. Lin Z., Shu F., Yang Y. An Empirical Study on bug assignment automation using Chinese bug data [Электронный ресурс] // URL:

http://www.xuebalib.com/cloud/literature-3ggDIF7j50C.html (дата обращения: 05.03.2021).

9. Mani S., Sankaran A., Aralikatte R. Deep-Triage: Exploring the Effectiveness of Deep Learning for Bug Triaging // arXiv.org. URL: https://arxiv.org/pdf/1801.01275.pdf (дата обращения: 12.03.2021).

10. Mihaylov M., Roper M. Predicting the Resolution Time and Priority of Bug Reports: A Deep Learning Approach // Department of Computer and Information Sciences, University of Strathclyde. 2019. URL: https://local.cis.strath.ac.uk/wp/extras/msctheses/papers/strath_cis_publication_2727.pdf (дата обращения: 03.03.2021).

11. Pennington J., Socher J., Manning, C.D. GloVe: Global Vectors for Word Representation [Электронный ресурс] // Computer Science Department, Stanford University. URL: https://nlp.stanford.edu/pubs/glove.pdf (дата обращения: 12.04.2021).

12. Pretrained Word Embeddings [Электронный ресурс] : An Essential Guide to Pretrained Word Embeddings for NLP Practitioners. URL: https://www.analyticsvidhya.com/blog/2020/03/pretrained-word-embeddings-nlp (дата обращения: 11.04.2021).

13. Ramay W.Y., Umer Q., Yin X.C. Deep neural network-based severity prediction of bug reports // IEEE Access. 2019. №7. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8685106 (дата обращения: 14.03.2021).

14. Sharma M., Bedi P., Chaturvedi K.K. и V.B. Singh. Predicting the priority of a reported bug using machine learning techniques and cross project validation // IEEE Access. 12th International Conference on Intelligent Systems Design and Applications. 2012. URL: https://ieeexplore.ieee.org/document/6416595 (дата обращения: 07.03.2021).

15. tf.keras.losses.CategoricalCrossentropy [Электронный ресурс] : TensorFlow Core v2.5.0. URL:

https://www.tensorflow.org/api_docs/python/tf/keras/losses/CategoricalCrossentropy (дата обращения: 05.04.2021).

16. The Functional API [Электронный ресурс] : TensorFlow Core. URL: https://www.tensorflow.org/guide/keras/functional (дата обращения: 07.04.2021).

17. Umer Q., Hui L., Sultan Y. Emotion based automated priority prediction for bug reports // IEEE Access. 2018. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8401501 (дата обращения: 02.03.2021).

18. What are Recurrent Neural Networks? [Электронный ресурс] : IBM Cloud Learn Hub. URL: https://www.ibm.com/cloud/learn/recurrent-neural-networks (дата обращения: 18.03.2021).

19. What is Flask used for? [Электронный ресурс] : DEV Community // URL: https://dev.to/amigosmaker/what-is-flask-used-for-2do5 (дата обращения: 1.05.2021).

20. When to use GRU over LSTM? [Электронный ресурс] : Data Science StackExchange. URL: https://datascience.stackexchange.com/questions/14581/when-to-use-gru-over-lstm (дата обращения: 31.03.2021).

Приложение А

Структура алгоритма для приоритизации дефектов

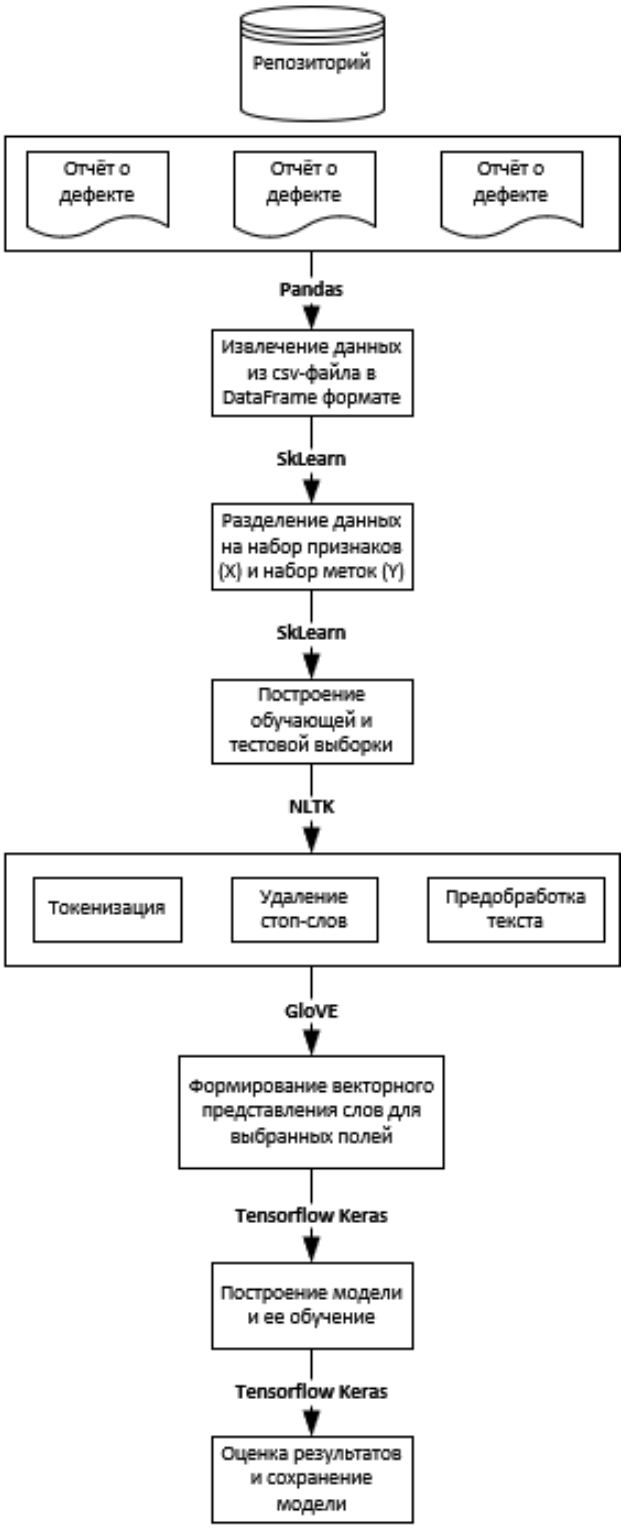


Рисунок А.1 – Схема для описания структуры алгоритма приоритизации

Приложение Б

Графическое представление обучаемой модели нейросети

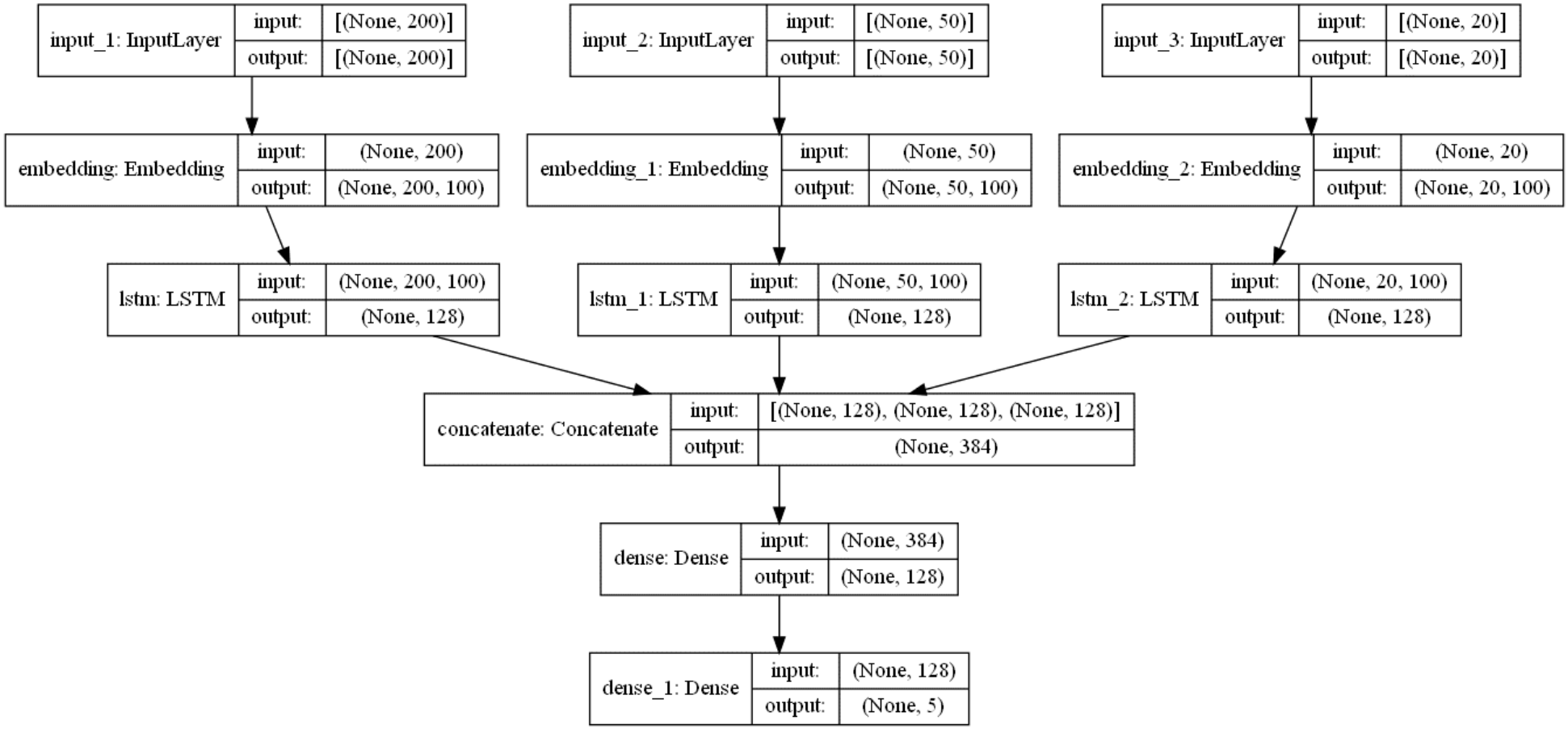


Рисунок Б.1 – Схематическое изображение сформированной нейросети