DOI 10.37882/2223-2966.2024.11-2.22

ПРИМЕНЕНИЕ БОЛЬШИХ ЯЗЫКОВЫХ МОДЕЛЕЙ ДЛЯ АВТОМАТИЗАЦИИ ТЕСТИРОВАНИЯ И ОТЛАДКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

APPLICATION OF LARGE LANGUAGE MODELS FOR SOFTWARE TESTING AND DEBUGGING AUTOMATION

E. Mozharovskii
A. Aluev
A. Dudak
A. Ishankhonov

Summary. The article explores the potential of using large language models (LLMs) for automating software testing and debugging processes. Models such as GPT and BERT, which demonstrate high potential in source code analysis, test scenario generation, and error detection, are discussed. The use of LLMs significantly improves testing accuracy, reduces development time, and automates labor-intensive processes related to identifying software vulnerabilities. Special attention is given to examples of LLM implementation at companies like Google, Microsoft, and Sberbank, where models are used for static code analysis, test generation, and improving software quality. The article also highlights the need for developing new testing methodologies for LLMs to enhance their resilience to errors, bias, and incorrect data. Challenges related to the computational resources required for model operation are also addressed. The authors conclude that the adoption of LLMs can significantly increase the efficiency of software development and testing, improve product quality, and reduce time to market, making these technologies promising for widespread use in software engineering.

Keywords: large language models (LLM), test automation, software debugging, GPT, BERT, testing methodologies.

Можаровский Евгений Александрович
Московский государственный университет имени М.В. Ломоносова
mozharovsky_ea@rambler.ru
Алуев Андрей Сергеевич
Уральский Федеральный Университет (Екатеринбург)
aluev_andrei@rambler.ru
Дудак Алексей Алиевич
Томский государственный университет систем управления и радиоэлектроники
aleksei.dudak@rambler.ru
Ишанхонов Азизхон Юнусхон угли
Национальный исследовательский технологический университет «МИСиС» (Москва)

Аннотация. В статье исследуются возможности применения больших языковых моделей (LLM) для автоматизации процессов тестирования и отладки программного обеспечения (ПО). Рассматриваются такие модели, как GPT и BERT, демонстрирующие высокий потенциал в анализе исходного кода, генерации тестовых сценариев и выявлении ошибок. Использование LLM позволяет существенно повысить точность тестирования, сократить время разработки, а также автоматизировать трудоемкие процессы, связанные с обнаружением уязвимостей в ПО. Особое внимание уделяется примерам внедрения LLM в компаниях Google, Microsoft и Сбербанк, где модели применяются для статического анализа кода, генерации тестов и улучшения качества ПО. В статье также подчеркивается необходимость разработки новых методик тестирования LLM для повышения их устойчивости к ошибкам, предвзятости и некорректным данным. Отдельно обсуждаются вызовы, связанные с вычислительными ресурсами, необходимыми для работы моделей. Авторы делают вывод о том, что внедрение LLM может значительно повысить эффективность разработки и тестирования ПО, улучшить качество конечных продуктов и сократить время их выхода на рынок, что делает данные технологии перспективными для массового применения в сфере программной инженерии.

Ключевые слова: большие языковые модели (LLM), автоматизация тестирования, отладка программного обеспечения, GPT, BERT, методики тестирования.

Введение

Современные большие языковые модели (LLM, Large Language Models), например, GPT (Generative Pretrained Transformer) и BERT (Bidirectional Encoder Representations from Transformers), демонстрируют высокие результаты в задачах генерации текста, машинного перевода, суммаризации и других областях.

Одной из ключевых сфер применения LLM является автоматизация тестирования и отладки программного обеспечения (ПО). В условиях роста сложности программных систем и объема кода, необходимость эффективного и автоматизированного тестирования становится все более актуальной. LLM могут анализировать текстовые данные, генерировать тестовые сценарии и находить ошибки в программном коде. Способность к анализу синтаксиса и семантики программ делает их перспективными инструментами для использования в программной инженерии.

Несмотря на значительный потенциал, использование LLM в тестировании и отладке сопряжено с рядом проблем. Основные из них связаны с точностью генера-

ции, возможными ошибками и предвзятостью, а также с ограничениями моделей в понимании сложных логических связей. Эти вызовы требуют разработки специфических методологий для тестирования самих LLM, а также методов улучшения их производительности. Целью данной статьи является исследование возможностей использования LLM для тестирования и отладки ПО, а также анализ существующих подходов к решению связанных с этим проблем.

Основные результаты

Теоретическая основа LLM

Сложные нейросетевые архитектуры, такие как LLM, предназначены для обработки и генерации естественного языка. Основной принцип их работы заключается в предсказании следующего слова или последовательности слов на основе анализа предыдущего контекста. Эти модели обучаются на огромных наборах текстовых данных, что позволяет им усваивать сложные языковые структуры и закономерности. Одной из самых известных и широко применяемых архитектур в данной области является трансформер, который лег в основу таких моделей, как GPT и BERT.

Архитектура **трансформеров** была предложена в 2017 году в работе «Attention is All You Need» команды исследователей из Google, и с тех пор стала доминирующим подходом в обработке естественного языка. Трансформеры используют механизм внимания (attention mechanism), который позволяет модели фокусироваться на значимых частях входных данных, игнорируя менее важные элементы. В отличие от традиционных рекуррентных нейронных сетей (RNN), трансформеры не обрабатывают данные последовательно, что значительно ускоряет процесс обучения и генерации текста. Этот механизм обеспечил прорыв в решении задач, связанных с анализом длинных текстовых последовательностей, где контекст может изменяться на протяжении всего текста.

Модель GPT (Generative Pretrained Transformer) является одной из самых известных LLM. Она основана на архитектуре трансформера и работает по принципу авторегрессии, что означает предсказание следующего элемента последовательности на основе предыдущих [1]. GPT обучается на обширных текстовых данных без конкретных меток, что позволяет ей генерировать связный текст на различных языках и для разных задач.

Основной инновацией GPT стало разделение процесса обучения на два этапа: предварительное обучение (pretraining) и дообучение (fine-tuning). На первом этапе модель обучается на большом объеме неразмеченных данных для понимания общих языковых закономерностей [2]. На втором этапе модель дообучается на небольших размеченных наборах данных для решения конкретных задач.

Такой подход позволяет добиться высокой универсальности модели, а также ее способности адаптироваться к новым контекстам с минимальными затратами на дополнительное обучение. Это значительно упрощает внедрение GPT в различные прикладные области и задачи.

BERT (Bidirectional Encoder Representations from Transformers) представляет собой другую модель, основанную на архитектуре трансформеров. В отличие от GPT, которая предсказывает следующую часть текста в последовательности, BERT использует двунаправленный подход к обработке данных [3]. Это означает, что модель анализирует весь контекст предложения, включая как предыдущие, так и последующие слова относительно текущего. Это обеспечивает более точное понимание смысловых связей в тексте и делает BERT особенно полезной для задач классификации текста, поиска информации и ответов на вопросы.

Токены входного текста кодируются в эмбеддинги — числовые представления слов или токенов, которые позволяют модели понимать их семантическое значение. Эти эмбеддинги отражают не только сами слова, но и их контекст в предложении, что помогает модели распознавать смысловые связи между ними. После преобразования токенов в эмбеддинги, они обрабатываются трансформером, который анализирует взаимосвязи между всеми токенами. Затем модель устанавливает смысловые связи между запросами, определяя начальные и конечные отрезки, которые могут соответствовать ответу на поставленный вопрос.

BERT также использует стратегию предварительного обучения и дообучения, но обучение основано на двух задачах: маскированное моделирование (masked language modeling), где некоторые слова во входных данных скрыты, и задача предсказания следования предложений (next sentence prediction), что помогает модели лучше понять смысловую структуру текста. Этот двунаправленный подход делает BERT мощным инструментом для задач, требующих глубокого понимания контекста.

Области применения LLM в тестировании и отладке

Способности LLM к обработке естественного языка и анализу больших объемов данных позволяют решать сложные задачи, связанные с поиском ошибок, оптимизацией кода и генерацией тестовых сценариев [4]. Одной из наиболее значимых сфер применения LLM является **автоматизация тестирования ПО**. Этот процесс традиционно требует значительных временных и чело-

вческих ресурсов для написания **тестовых сценариев и проведения тестов**. LLM могут существенно облегчить эти задачи, создавая автоматизированные тесты на основе описания функционала или исходного кода программы [5].

Генерация тестовых сценариев предполагает проверку различных аспектов работы программы, включая функциональное тестирование (соответствие программы заданным требованиям) и тестирование на отказоустойчивость (выявление возможных критических ситуаций). Пример такого тестового сценария может быть представлен следующим образом:

Тест на отказоустойчивость:
1. Ввести некорректный email (например, «user@com»).
2. Нажать кнопку «Зарегистрироваться».

Ожидаемый результат: появляется сообщение об ошибке, указывающее на некорректный формат email.

Способность LLM генерировать тестовые данные помогает не только анализировать код, но и предлагать оптимизированные решения для устранения ошибок. Например, в веб-приложениях модели могут создавать наборы данных, имитирующие поведение пользователей, что позволяет тестировать программу в условиях, близких к реальной эксплуатации (таблица 1).

В этих примерах набор данных имитирует различные сценарии поведения пользователей при регистрации. Тестовые данные включают как корректные, так и некорректные комбинации вводимых данных для проверки различных аспектов работы системы. LLM могут автоматически генерировать подобные наборы данных, учитывая потенциальные ошибки пользователей (например, ввод некорректных email-адресов или несовпадение паролей), что позволяет проводить тестирование программы в условиях, максимально приближенных к реальной эксплуатации.

Еще одной важной областью применения LLM в тестировании является **автоматическая генерация кода для тестов** [6]. LLM могут анализировать исходный код программы, определять возможные уязвимости и предлагать тестовые алгоритмы для проверки этих уязвимостей. Например:

```
import unittest
class TestEmailValidation(unittest.TestCase):
 def test_valid_email(self):
 self.assertTrue(is_valid_email(«user@example.com»))
 def test_invalid_email(self):
 self.assertFalse(is_valid_email(«userexample.com»))
if __name__ == «__main__»:
 unittest.main(), (1)
```

Этот код выполняет автоматическое тестирование функции is_valid_email, которая проверяет корректность email-адресов. Функция test_valid_email проверяет, что email, соответствующий правильному формату (например, «user@example.com»), признан корректным. Функция test_invalid_email проверяет, что email без символа «@» (например, «userexample.com») имеет некорректный формат email-адреса.

Для совершенствования процесса тестирования LLM могут **генерировать код для тестов на основе функциональных требований** [7]. Для требования «Программа должна принимать два числа и возвращать их сумму» код может выглядеть следующим образом:

```
import unittest
class TestSumFunction(unittest.TestCase):
 def test_sum_of_two_numbers(self):
 self.assertEqual(sum(2, 3), 5)
 self.assertEqual(sum(-1, 1), 0)
 self.assertEqual(sum(0, 0), 0)
if __name__ == «__main__»:
 unittest.main(), (2)
```

В данном примере, на основе функционального требования (сложение двух чисел), модель автоматически

Таблица 1.

Примеры наборов тестовых данных для веб-приложения регистрации пользователей

Имя пользователя	Email	Пароль	Повтор пароля	Ожидаемый результат
john_doe	john.doe@example.com	Passw0rd!	Passw0rd!	Успешная регистрация
jane_smith	jane_smith@example.net	MySecret12	MySecret12	Успешная регистрация
invalid_user	invalid_email.com	Test1234	Test1234	Ошибка: некорректный email
short_pass	user@domain.com	123	123	Ошибка: слишком короткий пароль
mismatch_pass	new_user@example.org	Qwerty123	Qwerty124	Ошибка: пароли не совпадают
duplicate_user	john.doe@example.com	NewPassword	NewPassword	Ошибка: имя пользователя уже существует

генерирует тест, проверяющий правильность выполнения этой задачи для нескольких случаев. Тесты проверяют, возвращает ли функция корректный результат для различных комбинаций чисел, что облегчает программистам процесс валидации кода.

Отладка ПО является одной из самых трудоемких задач в процессе разработки, так как требует детального анализа работы программы для выявления и исправления ошибок. LLM могут **анализировать логи** выполнения программ и находить закономерности, указывающие на наличие ошибок. Это может включать как синтаксические ошибки, так и логические ошибки, которые могут проявляться в неправильной работе программы при определенных условиях. Модели автоматически интерпретируют результаты выполнения программы и предлагают гипотезы о причинах сбоев [8]. Например, LLM может выявить, что ошибка возникает в случае передачи некорректных данных в определенную функцию, и предложить соответствующие корректировки.

Кроме того, LLM могут использоваться для **автоматической коррекции кода**. Используя знания о типичных ошибках программирования, модели предлагают способы исправления кода. Такой подход особенно полезен при работе с большими и сложными кодовыми базами, где ручная отладка может быть неэффективной.

Еще одной важной задачей, в которой LLM могут быть полезны, является **обработка текстовой информации**, связанной с ошибками и их устранением. Это может включать анализ отчетов, отзывов пользователей и других текстовых данных, которые часто используются для диагностики проблем в ПО [9].

Тестирование LLM

Перед внедрением LLM в работу необходимо оценить их функциональность, производительность, устойчивость и качество генерации. Важно учитывать, что модели могут работать в различных контекстах, что делает процесс тестирования многоступенчатым (таблица 2).

Тестирование LLM сталкивается с рядом значительных проблем, которые необходимо учитывать при их внедрении. Среди ключевых трудностей выделяются сложности объективной оценки качества генерации, высокая чувствительность моделей к некорректным или шумным данным, а также риски предвзятости, сформированные в процессе обучения. Проблемы с производительностью, связанные с высокими вычислительными требованиями, также могут ограничивать масштабирование моделей. Эти ограничения подчеркивают необходимость разработки более точных и комплексных методов тестирования для повышения надежности и эффективности использования LLM для тестирования и отладки.

Одним из потенциальных решений этих проблем является внедрение комплексного подхода к тестированию LLM, включающего комбинированные методики автоматизированного и ручного контроля качества. На-

Таблица 2.

Методологии тестирования LLM [10, 11]

Методология тестирования	Описание	Особенности
Функциональное тестирование	Проверка того, как модель выполняет определенные задачи (генерация текста, перевод, суммаризация).	Сложность объективной оценки качества выполнения задач, особенно в творческих или сложных заданиях.
Тестирование точности	Модели тестируются на предмет точности их ответов в задачах, требующих конкретных решений (например, математические задачи или ответы на вопросы).	Модель может давать правильный ответ случайно или на основе неточного анализа данных.
Тестирование устойчивости	Проверка модели на устойчивость к некорректным данным, изменениям ввода и другим потенциальным ошибкам.	LLM может быть слишком чувствительной к шуму и некорректным данным, что снижает ее полезность.
Оценка предвзятости	Модели проверяются на наличие культурной, социальной и гендерной предвзятости в их ответах.	Высокий риск воспроизведения предвзятых суждений, заложенных в исходных данных обучения.
Тестирование отказоустойчивости	Оценка поведения модели в условиях сбоев, некорректных данных или запросов.	Модель может неправильно интерпретировать ошибки и давать непредсказуемые или вредные ответы.
Генеративное тестирование	Оценка качества создаваемого моделью контента на естественных языках (тексты, описания, ответы).	Трудности в объективной оценке качества и связности генерируемого текста, особенно в длинных последовательностях.
Тестирование производительности	Оценка времени отклика и вычислительных ресурсов, необходимых для выполнения задач.	Высокие вычислительные требования могут снижать производительность на практике, особенно при масштабировании.

пример, для борьбы с предвзятостью моделей можно использовать специально разработанные наборы тестов, нацеленные на выявление культурных, социальных и гендерных предубеждений в ответах. Для повышения устойчивости к некорректным данным могут применяться техники повышения устойчивости, такие как аугментация данных или обучение на дополнительных наборах шумных данных. Также возможно оптимизировать производительность LLM с помощью алгоритмов, снижающих вычислительные затраты, например, через квантование моделей или внедрение методов распределенной обработки, что позволит эффективно масштабировать их для более широкого использования.

Внедрение LLM в работу компаний

Корпорации по всему миру активно используют LLM для автоматизации процессов тестирования и отладки ПО. В условиях растущей сложности современных приложений, LLM предлагают эффективные решения для повышения качества продуктов и сокращения времени разработки. Такие модели способны автоматизировать многие процессы, ранее требовавшие ручного вмешательства, включая генерацию тестов, анализ исходного кода и выявление ошибок.

Одна из крупнейших технологических компаний в мире, **Google** (США), активно внедряет LLM. Она использует модели на базе архитектуры трансформеров, включая свои собственные разработки, для автоматизации тестирования различных продуктов, таких как Google Cloud и Google Search. Одним из ключевых направлений является автоматическое генерирование тестов для больших кодовых баз [12]. Модели помогают находить ошибки на ранних стадиях разработки и проверять новые функции, что значительно сокращает время выхода продукта на рынок. Также Google использует LLM для статического анализа кода, что позволяет находить потенциальные уязвимости до их эксплуатации.

Один из примеров успеха Google в этой области — разработка совместно с UC Berkeley фреймворка **Self-Debugging**, который улучшает точность генерации кода до 12 % благодаря механизмам самокоррекции. Эта технология позволяет моделям выявлять и исправлять ошибки в коде, что повышает надежность программных продуктов, созданных с использованием LLM. Такой подход значительно ускоряет процессы тестирования и повышает эффективность разработки ПО, минимизируя ошибки до их развертывания в продакшн-среде.

Американская компания **Microsoft** активно использует LLM для улучшения процессов тестирования и отладки ПО, в первую очередь через интеграцию **GitHub Copilot** [13]. Этот инструмент, разработанный в партнерстве с OpenAI, значительно улучшает производительность разработчиков. GitHub Copilot был внедрен в различные продукты компании, включая Azure и Windows, и по состоянию на 2024 год он уже используется на более чем 75 млн устройств с Windows 10 и 11. Такая технология позволяет разработчикам быстрее писать код и автоматизировать генерацию тестов, что помогает сократить количество ошибок и повысить точность конечного продукта.

Кроме того, GitHub Copilot используется для обеспечения безопасности, что особенно важно для облачных сервисов Azure, где стабильность и защита данных являются приоритетными. Например, Copilot помогает более чем 700 тыс компаниям улучшать безопасность своих приложений и ускорять процессы DevSecOps. Организации, использующие Copilot, отмечают значительное увеличение производительности и сокращение времени, затрачиваемого на разработку, на 29 % для пользователей ранних версий Microsoft 365 Copilot.

В России одним из лидеров по внедрению LLM для автоматизации тестирования и отладки является Сбербанк. Он активно использует собственную языковую модель GigaChat для улучшения качества своих цифровых сервисов, включая мобильные приложения и платформы интернет-банкинга. Модели помогают автоматизировать процесс тестирования новых функций, анализировать проблемы и обнаруживать потенциальные сбои на ранних стадиях. Это позволяет значительно сократить время на отладку ПО, особенно в условиях масштабных проектов с миллионами пользователей. Кроме того, LLM используются для обработки обратной связи от клиентов, что помогает своевременно выявлять и исправлять проблемы, улучшая пользовательский опыт и надежность сервисов. Внедрение таких технологий позволило Сберу значительно повысить эффективность разработки ПО и снизить вероятность возникновения критических ошибок.

Заключение

Исследования LLM, предназначенных для тестирования и отладки ПО, демонстрируют значительный потенциал для автоматизации этих процессов. Современные LLM, такие как GPT и BERT, могут эффективно генерировать тестовые сценарии, обнаруживать ошибки в коде и предлагать способы их исправления. Однако их использование сопряжено с рядом вызовов, включая проблемы с точностью, устойчивостью к некорректным данным, а также риск предвзятости в ответах. Для решения этих проблем необходимы усовершенствования существующих методик и разработка новых подходов к тестированию самих моделей. Внедрение таких технологий может существенно повысить эффективность процессов тестирования и отладки, минимизируя затраты и ускоряя разработку сложных программных продуктов.

ЛИТЕРАТУРА

1. Bengesi S., El-Sayed H., Sarker M.K., Houkpati Y., Irungu J., Oladunni T. Advancements in Generative AI: A Comprehensive Review of GANs, GPT, Autoencoders, Diffusion Model, and Transformers. IEEE Access. 2024.
2. Вакушин А.А., Клебанов Б.И. Проектирование многокомпонентных имитационных моделей с помощью БЯМ GPT //Инженерный вестник Дона. 2024. №. 7 (115). С. 14.
3. Tewari A. LegalPro-BERT: Classification of Legal Provisions by fine-tuning BERT large language model //arXiv preprint arXiv:2404.10097. 2024.
4. Кондратюк А.П. Надежность и качество тестирования с использованием искусственного интеллекта //Материалы международной научной конференции «Мухтаровские чтения: актуальные проблемы математики, методики ее преподавания и смежные вопросы». Махачкала: ДГТУ, 2024 г. 228 с. 2024. С. 102.
5. Малыгин Д.С. Микросервисная архитектура в облачных системах: риски и возможности применения в 2024–2030 гг. // Моделирование, оптимизация и информационные технологии. 2024. Т. 12. № 2.
6. Korostin O. Innovations in the automation of electronic message processing for maritime shipping // Cold Science. 2024. № 6. P. 22–30.
7. Дроздов В.А. Применение больших языковых моделей для анализа уязвимостей //ББК 1 Н 34. С. 5631.
8. Пономарёв Е.В. Разработка кредитных приложений на Android: особенности и вызовы // Вестник науки. 2024. № 9(78). Т. 2. С. 319–327.
9. Пономарёв Е.В. Использование искусственного интеллекта для улучшения пользовательского опыта в мобильных приложениях // Инновационная наука. 2024. № 9-2. С. 54–59.
10. Самонов А.В., Бурова И.О. Методика разработки автоматизированных средств генерации программного кода посредством настройки больших языковых моделей //Вопросы кибербезопасности. 2024. №. 3. С. 61.
11. Pshychenko D. Study of artificial intelligence models for big data analysis in project management // International Journal of Humanities and Natural Sciences. 2024. Т. 8-3. № 95. С. 180–185.
12. Borra P. A Survey of Google Cloud Platform (GCP): Features, Services, and Applications //International Journal of Advanced Research in Science, Communication and Technology (IJARSCT). 2024. V. 4. №. 3. P. 191–199.
13. Yeverechyahu D., Mayya R., Oestreicher-Singer G. The Impact of Large Language Models on Open-source Innovation: Evidence from GitHub Copilot //arXiv preprint arXiv:2409.08379. 2024.