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

Институт цифровых технологий
(наименование)

Департамент магистратуры
(наименование)

09.04.03 Прикладная информатика
(код и наименование направления подготовки)

Технология бизнес - анализа
(направленность (профиль))

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА (МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ)

на тему: Оптимизация процесса тестирования программного обеспечения на основе моделирования предметной области

Обучающийся А.М. Гнучева
(Инициалы Фамилия) (личная подпись)

Научный руководитель канд. техн. наук, доцент, О.В. Аникина
(ученая степень (при наличии), ученое звание (при наличии), Инициалы Фамилия)

Тольятти 2025

Оглавление

Введение 4
Глава 1 Анализ современного состояния проблемы и теоретическая база исследования 9
1.1 Введение в тестирование на основе моделей 9
1.1.1 Типы диаграмм UML 11
1.1.2 Методы тестирования программного обеспечения 15
1.1.3 Тестовый случай 16
1.2 Генерация тестовых случаев и расстановка приоритетов 17
1.2.1 Основы алгоритма поиска в глубину DFS 18
1.2.2 Использование XML в генерации тестовых случаев 21
1.3 Методы машинного обучения 21
1.3.1 Алгоритмы контролируемого обучения 24
1.3.2 Алгоритмы неконтролируемого обучения 26
1.4 Обзор существующих методов генерации тестовых случаев на основе UML-диаграмм 30
Глава 2 Разработка и обоснование методологии исследования 39
2.1 Анализ деятельности компании ООО «Стильные кухни» 39
2.2 Сбор и подготовка данных 44
2.3 Предлагаемая модель 48
2.3.1 Экспорт диаграммы активности в формат XML 51
2.3.2 Преобразование диаграммы активности 52
2.3.3 Создание графа зависимостей 54
2.3.4 Генерация тестовых случаев 57
2.3.5 Классификация тест-кейсов по уровню приоритета 59
2.3.6 Оценка модели 60
2.3.7 Матрица ошибок 61
Глава 3 Реализация и экспериментальные результаты 65
3.1 Генерация тест-кейсов с использованием TestUML 65

3.1.1 Реализация 65
3.1.2 Подготовка и обработка данных 66
3.1.3 Анализ и интерпретация результатов 66
3.2 Оценка сгенерированных тест-кейсов 79
3.3 Результаты классификации тест-кейсов 81
3.3.1 Генерация признаков 81
3.3.2 Анализ работы алгоритма классификации KNN 83
3.3.3 Анализ результатов классификации методом NB 87
3.3.4 Результаты и анализ алгоритмов классификации SVM 91
3.4 Анализ результатов 95
Заключение 102
Список используемой литературы и используемых источников 104

Введение

Тестирование программного обеспечения является одним из наиболее важных и дорогостоящих этапов жизненного цикла разработки программных продуктов. Данный процесс занимает более 50% времени, а также ресурсов, выделяемых на проект, и направлен на выявление сбоев и ошибок, которые могут привести к несоответствию функциональности и требованиям пользователя [19]. Традиционные методы тестирования требуют значительных временных и человеческих ресурсов, что делает их малоэффективными для современных систем, отличающихся высокой сложностью и динамичностью. Современное программное обеспечение становится настолько тяжелым, что выполнение полного тестирования автоматического или ручного становится практически невозможным [26]. Более того, ручное тестирование подвержено ошибкам и требует значительных затрат во времени. В связи с этим особую актуальность приобретают методы автоматической генерации и приоритизации тестовых случаев. Одним из перспективных подходов является использование UML-диаграмм активности для автоматического создания тестовых случаев. Предлагаемый подход позволяет сократить время на создание тестов и повысить их качество за счет более полного покрытия требований. Однако, помимо генерации тестовых случаев, важную роль в повышении вероятности обнаружения ошибок на ранних этапах жизненного цикла разработки играет процесс их приоритизации [13]. Данная процедура заключается в присвоении каждому тестовому случаю определенного приоритета и последующее выполнение тестов в порядке возрастания приоритета. Оптимизировать процесс тестирования возможно при помощи минимизации выполнения тестовых случаев в порядке убывания приоритетов. Актуальность данной темы обусловлена необходимостью поиска новых подходов к автоматизированному тестированию, которые позволят сократить затраты

времени и ресурсов на тестирование и улучшить качество программных продуктов.

Научная проблема работы заключается в недостаточной разработке методов автоматической приоритезации тестовых случаев на ранних этапах жизненного цикла программного обеспечения, что приводит к увеличению затрат на тестирование и снижению его эффективности.

Гипотеза исследования заключается в том, что если применять алгоритмы машинного обучения на основе UML - диаграмм активности для автоматической генерации и приоритезации тестовых случаев, то данный процесс позволит сократить используемое время и затраты на тестирование программного обеспечения, повысить обнаружение ошибок программного обеспечения на ранних этапах и улучшить качество тестируемых систем.

Целью исследования является разработка модели автоматической генерации тестовых случаев, с использованием алгоритма поиска в глубину. Для генерации тестовых случаев была предложена стратегия, предполагающая покрытие всех переходов диаграммы активности, как минимум один раз. И приоритизации, сгенерированных тестовых случаев на основе UML-диаграмм активности, с использованием алгоритмов машинного обучения (метода опорных векторов (SVM), алгоритма метода К-ближайших соседей (KNN) и наивного байесовского ((NB) классификатора). Внедрение предлагаемой модели направлено на оптимизацию процедуры тестирования и повышение уровня надежности итоговых программных решений.

Для реализации поставленной цели в рамках работы планируется решение ряда задач:

- провести анализ научных публикаций и современных наработок, касающихся автоматического создания и расстановки приоритетов для тестовых сценариев, использующих UML-диаграммы;
- разработать новую модель, использующую в качестве основы UML-диаграммы активности;

- провести оценку эффективности предложенной модели в контексте генерации тестовых данных;
- исследовать возможности и адаптировать алгоритмы машинного обучения для автоматической классификации тестовых случаев по степени их важности;
- выполнить практическую реализацию созданной модели и проверить её работоспособность и результативность на реальных примерах.

Объектом исследования является процесс автоматического тестирования программного обеспечения.

Предметом исследования являются методы генерации и приоритезации тестовых случаев на основе UML-диаграмм активности.

Методология исследования: в основе предлагаемого подхода лежит комбинация двух методов. Для преобразования UML-диаграмм активности в тестовые маршруты применяется алгоритм поиска в глубину. Последующая сортировка полученных тестовых случаев по уровню критичности выполняется с привлечением моделей машинного обучения.

В рамках методологии исследования были выполнены следующие этапы:

- генерация тестовых случаев на основе покрытий всех переходов в UML-диаграммах активности;
- применение методов машинного обучения для классификации тестовых случаев по уровням приоритета;
- анализ эффективности предложенной модели на основе метрик покрытия и времени выполнения тестов.

Теоретической основной исследования стали:

- отечественные и зарубежные исследования по обеспечению качества программного обеспечения;
- публикации на сайтах, посвящённые тестированию программного обеспечения.

Теоретическая значимость работы заключается в разработке новой методологии генерации и приоритизации тестовых случаев, которая позволяет использовать UML-диаграммы активности на ранних этапах разработки программного обеспечения, что способствует повышению общей эффективности тестирования.

Практическая значимость работы заключается во внедрении предложенной методики в деятельность компании ООО «Стильные кухни» для улучшения процессов тестирования.

Научная новизна работы состоит в предложении модели, которая объединяет методы автоматической генерации тестовых случаев на основе UML - диаграмм активности с алгоритмами машинного обучения для их классификации по уровню приоритета.

Основные этапы исследования проводились с 2023 по 2025 год в несколько периодов:

На начальном этапе была сформулирована тема исследования, выполнялся сбор информации по теме исследования из различных источников, таких как научные статьи, книги и отчеты. Формулировалась гипотеза, определялись цели, задачи, объект и предмет исследования, а также выявлялась проблематика автоматизации процесса генерации и приоритезации тестовых случаев. Особое внимание уделялось анализу современных тенденций в области тестирования сложных программных систем и выявлению пробелов в существующих подходах к автоматизации.

На следующем этапе производился анализ существующих методологий тестирования программного обеспечения и генерации тестовых случаев. Изучались подходы к приоритезации тестовых случаев на основе машинного обучения. Была разработана теоретическая модель автоматической генерации тест-кейсов из UML-диаграмм активности с применением алгоритма поиска в глубину и методов машинного обучения. В процессе разработки модели проводились многократные циклы по ее совершенствованию, включая оптимизацию алгоритмов обработки UML-диаграмм и тестирование

различных конфигураций параметров машинного обучения. В рамках данного этапа была подготовлена и опубликована научная статья по теме исследования в научном журнале.

На завершающем этапе исследования проводилась проверка и оценка предложенной модели автоматизированной генерации и приоритезации тестовых случаев. Для валидации модели использовались реальные данные коммерческого проекта, что позволило оценить ее практическую применимость в условиях, максимально приближенных к реальной разработке программного обеспечения. На основании полученных результатов были сформулированы выводы об эффективности предложенной модели и ее влиянии на сокращение затрат на тестирование и повышение качества программных продуктов.

На защиту выносятся:

- модель автоматической генерации и приоритизации тестовых случаев, на основе UML-диаграмм активности;
- результаты проверки предложенной модели автоматизированной генерации тестовых случаев.

По теме исследования опубликована статья: Гнучева А.М. Оптимизация процесса тестирования программного обеспечения на основе моделирования предметной области // Вестник науки. 2024, №11(80) том 3. С. 925 – 929 [3].

Диссертация состоит из введения, трех глав, заключения и списка используемой литературы. Работа изложена на 109 страницах и включает 42 рисунка, 7 таблиц и 35 источника.

Глава 1 Анализ современного состояния проблемы и теоретическая база исследования

1.1 Введение в тестирование на основе моделей

Тестирование на основе моделей (Model-Based Testing, MBT) является одним из наиболее перспективных подходов к автоматизации процесса тестирования программного обеспечения. Основная идея данного метода заключается в использовании моделей, описывающих функциональность и поведение системы, для автоматической генерации тестовых случаев. Этот подход позволяет значительно сократить затраты времени и ресурсов на тестирование, а также повысить качество программного продукта за счёт более полного покрытия функциональных требований [9]. В отличие от традиционного тестирования, которое часто основывается на анализе исходного кода или ручном составлении тестовых случаев, тестирование на основе моделей опирается на формализованные описания системы. Это могут быть модели требований, поведенческие модели или модели, созданные на основе кода. Использование указанных моделей позволяет автоматизировать процесс тестирования на ранних этапах жизненного цикла разработки, что способствует более раннему обнаружению дефектов и их устранению [4].

В методологии тестирования на основе моделей (MBT) применяются различные типы моделей, каждая из которых решает свои задачи:

- модели требований. Они фиксируют функциональные характеристики системы, а также её нефункциональные атрибуты, такие как производительность или безопасность;
- поведенческие модели. Эти модели специфицируют ожидаемые реакции системы на разнообразные внешние стимулы, включая пользовательский ввод и события;

– модели кода. Данный тип моделей дает представление о внутренней архитектуре системы, отображая её структуру и алгоритмическую логику.

Процедура автоматического формирования тестовых сценариев в MBT строится по определенному алгоритму. На первом этапе разрабатывается формальная модель, являющаяся абстрактным представлением ключевой функциональности системы. На следующем этапе специальные алгоритмы анализируют эту модель, чтобы экстрагировать из нее набор конкретных тестовых случаев. Широкой популярностью при создании таких моделей пользуется унифицированный язык моделирования (UML). Этот язык предоставляет богатый набор визуальных нотаций, позволяющих наглядно описать структуру и поведение системы. Язык UML предлагает широкий выбор графических элементов для наглядного представления архитектуры и работы системы. Данный язык фактически стал общепринятым стандартом для визуального проектирования в объектно-ориентированном программировании. Его применяют для описания бизнес-процедур, построения системных архитектур и отображения структур организаций [12]. Он признан основной системой моделирования для объектно-ориентированных программ и повсеместно применяется в сфере создания программного обеспечения.

Различные типы диаграмм UML - активности, последовательности и состояний - дают возможность четко представить системные требования и логику работы. К примеру, диаграммы активности отображают очередность операций в системе, что помогает планировать проверки и автоматизировать их выполнение. Тестирование делится на два ключевых метода:

– проверка по спецификациям (черный ящик) тестирует работу системы без изучения ее внутреннего устройства. Тестовые примеры формируются согласно требованиям и ожидаемым реакциям системы;

– проверка по коду (белый ящик) изучает внутреннюю структуру программы. Тестовые примеры извлекаются из исходного кода, позволяя контролировать правильность работы отдельных модулей.

Основной трудностью тестирования на основе моделей является создание точных системных моделей. Ошибки в моделях ведут к формированию ошибочных тестов и снижают результативность проверок. Поэтому в данном тестировании важно применять формальные языки моделирования и средства их контроля.

1.1.1 Типы диаграмм UML

Унифицированный язык моделирования UML служит популярным средством для проектирования объектно-ориентированных программных решений. В современной разработке UML позиционируется как основной стандарт для инженерного проектирования программ. Он помогает визуализировать, специфицировать, разрабатывать и документировать программные продукты. Благодаря UML разработчики могут строить точные описания системной архитектуры, ее элементов и функций, что ускоряет процесс создания и проверки программ [6].

UML представляет собой графическую систему для определения и описания компонентов программ. Он дает возможность строить графические модели, демонстрирующие разные стороны программного решения: архитектуру, логику работы, потоки информации и связи между модулями. Этот язык моделирования охватывает разные варианты представления системы и позволяет изучать ее с различных позиций [33]. Такие возможности делают UML ценным инструментом для проектирования сложных программных комплексов.

Проверка программ - необходимый этап создания любого программного продукта [18]. Она обеспечивает качество программ, повышает их надежность и стабильность. Ключевой задачей тестирования остается подготовка тестовых примеров, которые подтверждают соответствие системы установленным требованиям. Главными источниками для формирования

тестов служат требования к программному продукту. Спецификации программ могут быть представлены как UML - модели, формальные языковые описания или тексты на естественном языке. Применение UML дает возможность автоматизировать подготовку тестовых примеров, увеличивая общую продуктивность тестирования.

Применение тестирования на основе моделей позволяет сократить время разработки программного обеспечения, так как тестовые случаи могут быть созданы ещё на ранних этапах проектирования системы. Одной из ключевых практик современной разработки является подход, при котором тестирование интегрируется в процесс написания кода. Такой симбиоз позволяет минимизировать потенциальные риски и существенно повысить общую продуктивность работы. Когда тестовые сценарии создаются на ранних этапах, это не только ускоряет валидацию функциональности готового продукта, но и способствует оперативному выявлению и устранению дефектов. Для проектирования таких систем часто применяется язык UML, представляющий собой стандартизированный инструмент графического моделирования. С его помощью разработчики могут наглядно описывать структуру и поведение объектно-ориентированных приложений, создавая разнообразные проектные артефакты. В арсенале UML существует множество типов диаграмм, выбор которых зависит от конкретных целей моделирования. Всё это многообразие принято классифицировать на две основные группы: структурные диаграммы, отображающие статическую композицию системы, и поведенческие, которые демонстрируют динамику взаимодействия её компонентов в процессе работы.

Кратко охарактеризуем структурные типы диаграммы UML, они иллюстрируют структуру системы, включая ее классы, объекты, пакеты, компоненты и другие элементы, а также установленные между ними связи. Структурные диаграммы можно разделить на следующие виды.

Диаграммы классов, в них реализуется статичная структура системы, включая классы, их атрибуты, поведение и взаимосвязи.

Диаграммы компонентов, более подробная версия диаграммы классов, и в той, и в другой действуют одни и те же правила. Диаграмма компонентов позволяет разбить комплексную систему на более мелкие составляющие и наглядно продемонстрировать установленные между ними связи.

Диаграммы развертывания иллюстрируют, структуру аппаратного обеспечения системы. Данные диаграммы особенно полезны, когда программное решение распределено по нескольким машинам, каждая из которых имеет свою конфигурацию. Диаграммы развертывания позволяют разработчикам визуализировать архитектуру системы, включая распределение компонентов программного обеспечения на физические узлы.

Диаграммы объектов дают нам примеры того, как структуры данных выглядят в конкретный момент времени. Если статичную структуру удобно представить в виде диаграммы классов, то диаграммы объектов, как контрольные примеры, позволяют судить, все ли ее компоненты на месте;

Диаграммы пакетов наглядно демонстрируют зависимости между пакетами в составе системы.

Перечислим основные типы поведенческих диаграмм UML.

Диаграммы состояния или диаграммы конечного автомата, фиксируют динамическое поведение объектов, переходящих из одного состояния в другое. Они описывают переходы состояний объектов в ответ на определённые события. Диаграммы конечных автоматов обычно применяются для моделирования поведения экземпляров классов, но могут быть использованы и для описания поведения других сущностей, таких как варианты использования, акторы, подсистемы, операции или методы.

Диаграммы вариантов использования, один из самых популярных видов поведенческих диаграмм UML. Они обеспечивают визуальное представление взаимодействий между участниками системы, функций, необходимых этим участникам, и связей между функциями. Этот вид диаграмм помогает систематизировать требования и понять, какие действия ожидаются от системы в различных сценариях её использования.

Диаграммы последовательностей, относятся к типу диаграмм взаимодействия, фиксирующих последовательность взаимодействий между объектами с учётом временных аспектов. Они отображают хронологическую последовательность сообщений, включая их названия, ответы и возможные аргументы. Данные диаграммы помогают визуализировать временную структуру взаимодействий между объектами в системе.

Диаграмма действий, или диаграмма активностей, активити - диаграмма. Иллюстрирует поведение системы с точки зрения рабочего процесса. Поскольку действия представляют собой состояние выполнения определённых операций, данные диаграммы напоминают диаграммы состояний. Диаграмму активности можно сравнить с блок-схемой, отображающей компьютерный код. На такой диаграмме представлены события, которые приводят к решениям и действиям в коде, а также последовательность выполнения бизнес-логики. Диаграммы активности показывают, как работает определенный процесс в системе. На них можно увидеть последовательные и параллельные операции, а также условия выполнения тех или иных действий. Эти диаграммы наглядно представляют взаимодействие между компонентами системы, отображая точки принятия решений и одновременное выполнение различных процессов.

Обычно диаграммы активности читают сверху вниз - это помогает понять последовательность шагов и логику процесса. Такой способ представления информации делает анализ системы более простым и понятным. Таким образом, использование UML диаграмм в тестировании программного обеспечения открывает новые возможности для автоматизации, особенно в области генерации тестовых случаев. В частности, диаграммы активности являются удобным инструментом для визуализации последовательности действий системы и могут служить основой для создания тестовых сценариев.

1.1.2 Методы тестирования программного обеспечения

Для создания качественных тестовых случаев используются различные методы тестирования программного обеспечения. Тестовый случай представляет собой универсальный программный артефакт, описывающий совокупность шагов, конкретных условий и параметров, необходимых для проверки реализации тестируемой функции или её части [7]. Методы автоматизированного тестирования позволяют сократить количество требуемых тестовых случаев, одновременно увеличивая покрытие тестирования. Это особенно важно, поскольку выполнение полного тестирования и всех возможных сценариев часто оказывается нецелесообразным. Применение указанных методов упрощает процесс генерации тестовых ситуаций, делая их более эффективными [23]. Методы тестирования программного обеспечения предполагают использование различных стратегий для проверки функциональности, производительности и внешнего вида приложения. Основными этапами тестирования являются:

- анализ требований;
- планирование тестирования;
- создание тестовых случаев, а также определение их условий, переменных и ожидаемых результатов;
- создание тестовых случаев;
- подготовка к тестированию и настройка окружения;
- выполнение тестов, проведение проверок с использованием разработанных тестовых случаев;
- оценка результатов и анализ полученных данных для выявления несоответствий.

На рисунке 1 представлена схема процесса жизненного цикла тестирования ПО.

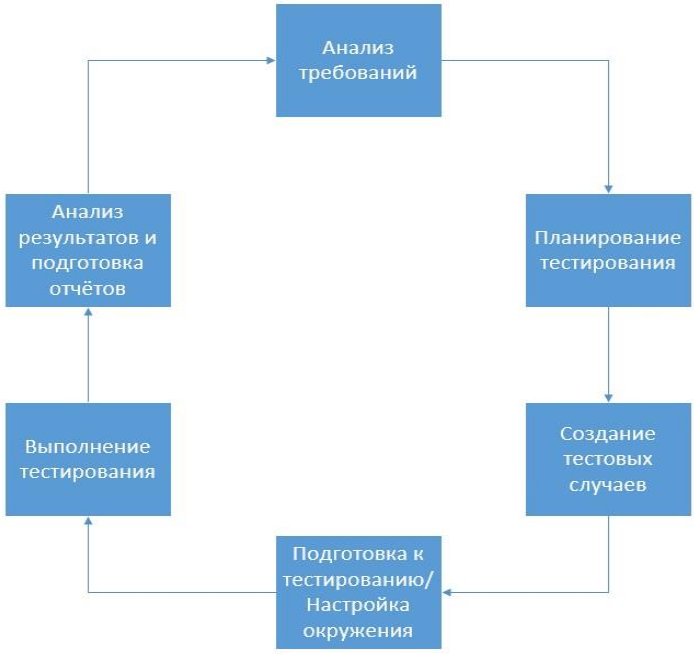


Рисунок 1 – Жизненный цикл тестирования ПО

Автоматизация тестирования становится необходимостью для выполнения тщательной проверки ПО. Данный процесс позволяет сократить временные и финансовые затраты, а также улучшить качество тестирования [1]. Применение критериев окончания тестирования помогает ограничить процесс и определить, когда тестирование можно считать завершённым. С увеличением размеров и сложности программных систем тестирование требует всё больше времени и усилий инженеров по тестированию.

1.1.3 Тестовый случай

Тестовый случай представляет собой набор условий, данных и ожидаемых результатов, предназначенных для проверки точности и качества ПО [8]. Каждый тестовый случай включает:

- набор входных данных, необходимых для выполнения теста,
- условия выполнения, определяющие контекст выполнения теста,
- ожидаемые результаты, которые сравниваются с фактическими для оценки соответствия.

Тестовые случаи создаются с целью проверки определённых сценариев или последовательностей операций, а также для проверки соответствия требованиям [2]. Процесс разработки тестовых случаев включает: определение условий тестирования и тестовых случаев на основе спецификаций и проектной документации. Каждый тестовый случай содержит элементы: ожидаемый результат, цель теста, входные данные и метод тестирования. Качество тестирования определяется тем, насколько тестовые случаи соответствуют спецификациям и требованиям. Корректность тестирования также зависит от того, насколько тщательно тестовые случаи проверены и соответствуют заявленным целям.

1.2 Генерация тестовых случаев и расстановка приоритетов

Тестовый случай является важным элементом тестирования программного обеспечения, представляя собой набор условий, данных и ожидаемых результатов, необходимых для проверки функциональности программной системы. Выходные данные, генерируемые программным обеспечением при выполнении тестового случая, используются для сравнения с ожидаемыми результатами и оценки фактического поведения системы. Согласно современным исследованиям, тестовый случай является артефактом программной инженерии, который включает входные значения, ожидаемые выходные данные и условия, необходимые для выполнения теста [17]. Одним из наиболее сложных и ресурсоёмких этапов тестирования программного обеспечения является именно создание тестовых случаев. Согласно исследованию «Keyvanpour, M. R., Homayouni, H., & Shirazee, H. Automatic software test case generation» [26], автоматизированные подходы к генерации тестовых случаев направлены на быстрое определение ограниченного числа сценариев, которые удовлетворяют критериям покрытия. Указанный процесс позволяет сократить затраты на тестирование, повысить эффективность и улучшить качество программных продуктов. Для автоматической генерации

тестовых случаев используются различные методы, но большинство из них применяют мета эвристические алгоритмы поиска, которые помогают находить оптимальные тестовые сценарии [16]. Генерация тестовых случаев на этапе проектирования имеет особую важность, так как она позволяет выявлять дефекты на уровне дизайна, что способствует повышению надёжности программного обеспечения. Тестирование на основе моделей, при котором тестовые случаи создаются на основе исходного кода, проектных спецификаций или требований, считается более эффективным и оперативным по сравнению с традиционными методами, основанными на коде [9]. Расстановка приоритетов тестовых случаев представляет собой один из ключевых подходов для оптимизации и повышения общей эффективности тестирования. Она позволяет организовать тестовые случаи таким образом, чтобы первоочередные проверки охватывали наиболее критичные аспекты системы. Традиционные методы приоритизации основываются на анализе: истории работы приложения, включая частоту ошибок в различных компонентах, а также способности тестов выявлять ошибки. Однако большинство этих методов ориентированы на код и применяются на этапе тестирования после внедрения. Это ограничивает их эффективность для ранних стадий жизненного цикла разработки ПО. Для решения этой проблемы необходима стратегия расстановки приоритетов, основанная на модели, что делает возможным применение порядка очередности на ранних этапах разработки ПО [22].

1.2.1 Основы алгоритма поиска в глубину DFS

Алгоритм поиска в глубину (Depth First Search, DFS) – это рекурсивный алгоритм, основанный на принципе возврата. Его суть заключается в том, чтобы выполнять исчерпывающий поиск всех узлов графа, двигаясь как можно дальше вперёд, а затем возвращается назад, если дальнейшее продвижение невозможно [31]. Алгоритм DFS широко применяется в различных задачах, включая топологическую сортировку, планирование

задач, обнаружение циклов в графах и решение головоломок с единственным решением, таких как лабиринты или судоку [31].

Алгоритм DFS начинается с корневого узла (в случае графа может быть выбран любой узел в качестве начального) и проверяет каждую ветвь, продвигаясь максимально далеко, прежде чем вернуться назад [31]. На рисунке 2 изображен пример реализации алгоритма поиска в глубину.

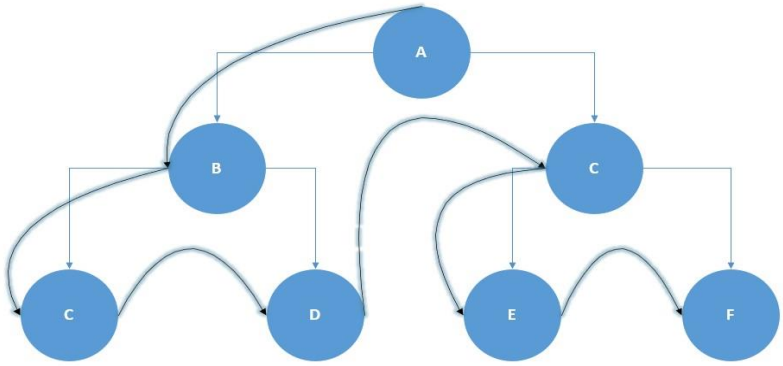


Рисунок 2 – Последовательность реализации алгоритма DFS

Алгоритм поиска в глубину DFS является эффективным методом обхода графов, позволяя последовательно исследовать узлы, уходя вглубь структуры перед возвратом к предыдущим уровням. В контексте автоматизированного тестирования DFS играет ключевую роль в генерации тестовых путей на основе UML-диаграмм активности, обеспечивая покрытие всех возможных маршрутов выполнения сценариев. Но, для автоматизации данного процесса необходимо представлять UML-диаграммы в удобном для обработки формате. Одним из наиболее распространённых способов хранения и передачи структурированных данных является Extensible Markup Language (XML). В следующем разделе рассмотрим, каким образом XML используется для представления диаграмм активности и каким образом преобразование UML-диаграмм в XML-файлы способствует генерации тестовых случаев [20].

В таблице 1 рассмотрим альтернативные алгоритмы, а также их преимущества и недостатки.

Таблица 1 – Сравнение альтернативных алгоритмов с DFS

Алгоритм	Плюсы	Почему не выбрали?
Поиск в ширину (BFS)	Гарантирует кратчайший путь, подходит для невзвешенных графов	Тестовые случаи получаются слишком короткими, не учитываются все возможные ветки
Поиск пути (алгоритм Дейкстра)	Оптимален для нахождения кратчайшего пути	Не нужен поиск кратчайшего пути, так как в тестировании важно полное покрытие
Жадные алгоритмы	Использует эвристики, находит оптимальный путь	Сложнее в реализации, требует оценки эвристик
Генетический алгоритм	Эффективен в задачах оптимизации, способен находить сложные зависимости	Случайный характер мутаций может приводить к неоптимальным решениям, требует высокой вычислительной мощности
Алгоритм K-средних (K-Means)	Прост в реализации, быстро находит кластеры в данных	Не применим для поиска тестовых путей, так как предназначен для кластеризации данных

Как показал анализ представленных алгоритмов, наилучшие характеристики для генерации тестовых случаев из UML - диаграмм активности у алгоритма поиска в глубину. В отличие от поиска в ширину BFS, DFS проходит по одному пути до конца перед возвратом, что делает его удобным для построения последовательных тестовых случаев. DFS гарантирует покрытие всех путей на UML - диаграмме активности, что позволяет учесть все возможные сценарии работы системы. Это особенно важно при тестировании программного обеспечения, так как помогает обнаружить редкие, но потенциально критичные ошибки. UML - диаграммы активности могут быть представлены в виде графов, где DFS естественным образом обходит все возможные пути. Также DFS не требует значительных вычислительных ресурсов. И в отличие от алгоритмов поиска кратчайшего

пути (например, Дейкстры), DFS не требует хранения информации о весах рёбер, что упрощает его применение.

1.2.2 Использование XML в генерации тестовых случаев

Extensible Markup Language (XML) - это расширяемый язык разметки, схожий с HTML, но без предопределённого набора тегов. В отличие от HTML, XML позволяет разработчикам создавать собственные теги, которые могут быть специально адаптированы под их задачи и требования. Данный язык стал основным способом хранения данных в формате, который можно легко сохранять, искать и использовать совместно с другими системами. Одним из ключевых преимуществ XML является его стандартизированный формат, благодаря которому передача данных между системами или платформами (локально или через интернет) остаётся простой и эффективной. Получатель данных, даже находясь на другой платформе, может успешно проанализировать данные, так как синтаксис XML является универсальным и стандартизированным [5]. В рамках нашего исследования формат XML используется для генерации тестовых случаев на основе диаграмм активности UML. Диаграммы активности, созданные в инструменте разработки Enterprise Architect, экспортируются в формате XML. XML - файл содержит структурированное описание диаграммы активности, что позволяет автоматизировать процесс извлечения тестовых случаев. Применение XML обеспечивает универсальность и независимость от используемых платформ, что делает указанный подход удобным и эффективным для реализации задач автоматизированного тестирования.

1.3 Методы машинного обучения

Машинное обучение - это форма анализа данных, которая автоматизирует разработку аналитических моделей [11]. Оно является частью искусственного интеллекта и базируется на предположении, что машина

способна обучаться на данных, выявлять закономерности и принимать решения практически без участия человека [11].

Основная задача машинного обучения заключается в создании алгоритмов, которые, получая и обрабатывая данные, способны преобразовывать накопленный опыт в знания. Машинное обучение стало одной из самых востребованных областей компьютерной науки, находя широкое применение в таких задачах, как автоматическое обнаружение закономерностей, предсказание результатов и классификация данных. Данный процесс включает использование статистических методов, которые позволяют компьютерам обучаться на данных без необходимости их явного программирования [28]. Контролируемое обучение представляет собой метод машинного обучения, в котором используются размеченные данные для обучения модели. В контролируемом обучении используются наборы данных, содержащие входные параметры и соответствующие им результаты. Это помогает алгоритмам выявлять закономерности и делать прогнозы для новых, ранее не встречавшихся данных. Главная задача такого подхода - создание моделей, способных точно распределять данные по категориям или предсказывать числовые значения.

Контролируемое обучение включает два основных типа задач. Первый - классификация, где алгоритм относит объекты к определенным классам. Например, он может различать изображения яблок и апельсинов или определять спам - письма в электронной почте. Для решения таких задач применяют метод опорных векторов (SVM), деревья решений и другие классификаторы. Второй тип - регрессия, которая предназначена для прогнозирования численных показателей. Алгоритмы регрессии устанавливают связи между входными переменными и целевыми значениями, позволяя предсказывать результаты на основе имеющихся данных. Например, такие модели могут прогнозировать будущую выручку компании на основе данных о продажах. Среди распространённых алгоритмов регрессионного анализа - линейная и логистическая регрессия.

Неконтролируемое обучение отличается тем, что модели работают с неразмеченными данными. Алгоритмы автоматически анализируют и группируют данные, выявляя закономерности без вмешательства человека. Такой подход часто применяется для решения задач кластеризации, ассоциации и снижения размерности.

Кластеризация, этот метод используется для группировки данных на основе их сходства. Например, алгоритмы K - ближайших соседей (KNN) формируют группы данных, анализируя их схожесть. Кластеризация применяется в маркетинговой сегментации, обработке изображений и других задачах;

Ассоциация или ассоциативные методы направлены на выявление закономерностей и связей между различными переменными в данных. Такой подход востребован в анализе покупательского поведения для создания рекомендательных систем, например, разделов типа: «Покупают вместе с этим товаром»;

Снижение размерности, данный метод используется для упрощения сложных данных путём уменьшения количества признаков, сохраняя ключевую информацию. Снижение размерности часто применяется в обработке изображений, например, с использованием авто кодировщиков для устранения шумов [15].

Ключевое различие между контролируемым и неконтролируемым обучением заключается в использовании размеченных данных. Контролируемое обучение требует наличия обучающего набора, содержащего точные входные и выходные данные, что позволяет модели учиться путём корректировки прогнозов на основе полученных результатов. Неконтролируемые алгоритмы, напротив, изучают внутреннюю структуру данных без предварительной разметки, но их результаты часто требуют проверки или доработки со стороны человека [10]. Данный подход полезен для выявления скрытых закономерностей и создания решений для задач, где

разметка данных невозможна или экономически нецелесообразна. Далее рассмотрим более подробно несколько из вышеупомянутых алгоритмов.

1.3.1 Алгоритмы контролируемого обучения

Наивный байесовский алгоритм (NB).

Наивный байесовский классификатор широко используется в задачах классификации и кластеризации. Он является одним из самых простых и эффективных методов классификации, основанных на теореме Байеса. Упрощенные алгоритмы Байеса позволяют вычислить вероятность какого-либо события в зависимости от связанного события. За один проход вычисляется условная вероятность каждого признака, затем применяется теорема Байеса для нахождения распределения вероятности наблюдений.

Принцип работы наивного байесовского классификатора основан на вероятностном анализе признаков объекта. Например, если перед нами находится зеленый объект, алгоритм анализирует его возможные классы (мяч, собака, ёжик) и присваивает наибольшую вероятность наиболее подходящей категории. Это делается путем сравнения признаков объекта с известными данными и выбора класса с наибольшей вероятностью. На рисунке 3 показан принцип работы наивного байесовского классификатора.

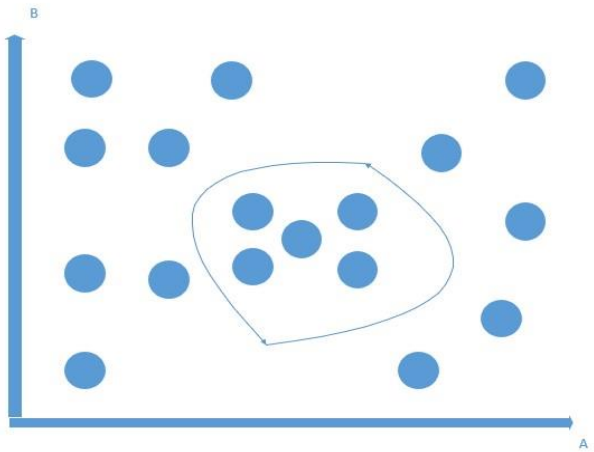


Рисунок 3 – Принцип работы наивного байесовского классификатора

Наивный байесовский классификатор обладает рядом преимуществ. Он прост в реализации, требует минимальных вычислительных ресурсов и быстро обучается даже на небольших наборах данных. Кроме того, он эффективно работает в много классовых задачах и хорошо справляется с обработкой категориальных признаков. Однако у него есть также и недостатки. Один из ключевых минусов - проблема нулевой частоты: если в тестовых данных встречается категория, которой не было в обучающем наборе, алгоритм присваивает ей нулевую вероятность и не может корректно классифицировать объект [30]. Для решения указанной проблемы применяется метод сглаживания Лапласа.

Метод опорных векторов (SVM)

Метод опорных векторов (Support Vector Machine, SVM) представляет собой алгоритм классификации, который разделяет входные данные на два класса с помощью оптимальной разделяющей гиперплоскости. Цель SVM - максимизировать расстояние между ближайшими векторами (опорными векторами) и гиперплоскостью, обеспечивая минимизацию ошибок классификации. Данный подход применим, как для линейно разделимых данных, так и для задач, требующих обработки нелинейных зависимостей. На рисунке 4 представлен принцип работы линейного SVM.

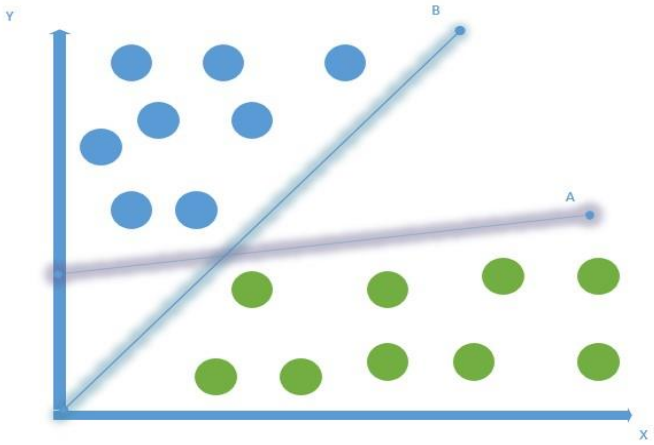


Рисунок 4 – Примеры разделяющих гиперплоскостей в линейном SVM

В основе алгоритма SVM лежит принцип поиска такой гиперплоскости, которая способна с максимальной эффективностью разделить классы в пространстве данных. Ключевую роль в этом процессе играют так называемые опорные векторы - это точки данных, расположенные ближе всего к границе принятия решения. Главная задача алгоритма - добиться максимально возможного расстояния (зазора) между этими критическими точками и разделяющей гиперплоскостью. Именно эта особенность обеспечивает высокую устойчивость и достоверность классификации.

Алгоритм SVM нашел широкое практическое применение в таких областях, как компьютерное зрение, текстовый анализ и прогнозирование на финансовых рынках. Его важным достоинством является способность показывать высокую точность даже на ограниченных наборах тренировочных данных. В отличие от многих других алгоритмов, SVM не всегда требует огромных выборок для эффективного обучения.

Кроме того, метод обладает повышенной устойчивостью к аномальным выбросам и шуму в исходных данных. Он также демонстрирует хорошую производительность при работе с несбалансированными классами, что открывает возможности для его использования в решении широкого круга прикладных задач, где данные часто далеки от идеала. Еще одним достоинством SVM является относительная простота реализации: современные библиотеки машинного обучения позволяют легко интегрировать этот метод в существующие системы. Однако и у данного метода есть ограничения. Он может быть вычислительно затратным при работе с большими наборами данных.

1.3.2 Алгоритмы неконтролируемого обучения

Алгоритм K-ближайших соседей (KNN)

Алгоритм K-ближайших соседей может помочь решить задачу классификации, в случае, когда категорий больше, чем 2. В случае использования метода для классификации объект присваивается тому классу, который является наиболее распространённым среди K - соседей данного

элемента, классы которых уже известны. В случае использования метода для регрессии, объекту присваивается среднее значение по K ближайшим к нему объектам, значения которых уже известны. На рисунке 5 представлен принцип работы алгоритма KNN.

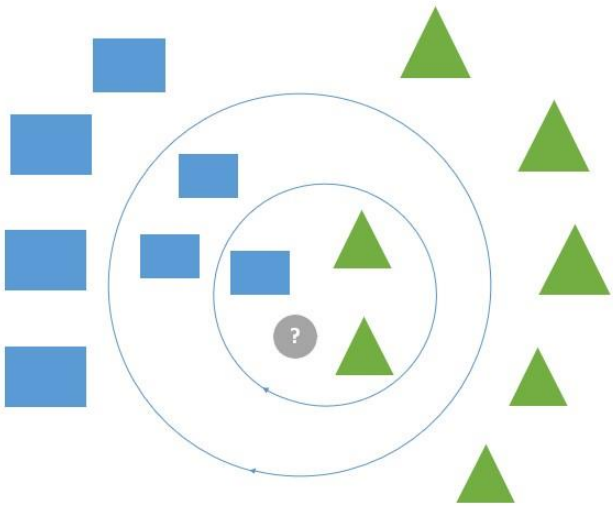


Рисунок 5 – Работа алгоритма KNN

Алгоритм K - ближайших соседей (KNN) по праву считается одним из самых простых для понимания в машинном обучении. Его универсальность позволяет решать задачи как классификации, так и регрессии, что делает KNN востребованным инструментом для анализа разнообразных данных. Главным достоинством KNN является его концептуальная простота. Алгоритм не нуждается в построении сложной математической модели и требует минимальных усилий по настройке. Кроме того, KNN демонстрирует устойчивость к аномальным выбросам. Это достигается за счет того, что итоговое решение (например, класс объекта) определяется большинством голосов среди его ближайшего окружения в пространстве данных, что нивелирует влияние редких или ошибочных значений.

Однако за простоту концепции приходится расплачиваться на практике. Главный минус алгоритма KNN - сильная зависимость от объема данных. При увеличении размера обучающей выборки скорость работы алгоритма заметно снижается. Это происходит из-за того, что для определения класса нового объекта требуется рассчитать его удаленность от всех существующих точек, что вызывает высокую нагрузку на вычислительные ресурсы и тормозит обработку больших массивов информации. Другой сложностью становится подбор параметра K, от которого зависит число соседей, учитываемых при классификации. Слишком низкое значение K повышает чувствительность к случайным отклонениям в данных, а слишком высокое - уменьшает различия между классами и ухудшает точность прогнозов. Тем не менее, KNN сохраняет свою полезность при работе с небольшими наборами данных и для создания прототипов, где главным плюсом становится легкость его применения. Но в промышленных системах, обрабатывающих значительные объемы данных, его ограничения обычно превосходят преимущества.

В таблице 2 сравним самые популярные и применяемые алгоритмы машинного обучения.

Таблица 2 – Сравнительная таблица алгоритмов машинного обучения

Название алгоритма	Преимущества	Недостатки
Дерево решений	Хорошо интерпретируемый, не требует масштабирования данных, устойчив к выбросам	Склонен к переобучению, плохо работает на непрерывных данных
Наивный байесовский классификатор	Быстро работает, требует мало данных для обучения, хорошо работает на текстах	Предполагает независимость признаков, чувствителен к выбросам
Метод опорных векторов	Эффективен для малых выборок, хорошо справляется с линейно разделимыми данными	Медленный на больших данных, сложно подобрать ядро
Логистическая регрессия	Простота, работает с бинарными и много классовыми задачами, применим к большим данным	Чувствителен к выбросам, плохо работает на сложных данных

Продолжение таблицы 2

Название алгоритма	Преимущества	Недостатки
Метод K-ближайших соседей	Простота, легко обучается, работает с нелинейными данными	Медленный при больших объемах данных, зависит от выбора K
Случайный лес	Снижает переобучение, хорошо работает на сложных данных, не требует нормализации	Медленный, сложно интерпретировать, требует много ресурсов
Градиентный бустинг	Высокая точность, хорошо справляется с нелинейностями, стабилен	Медленный на больших данных, чувствителен к выбору параметров
Автоэнкодеры	Эффективен для снижения размерности, выявляет скрытые паттерны в данных	Может переобучаться, сложно настраивать
Глубокие нейронные сети	Высокая точность на сложных данных, хорошо обрабатывает изображения и текст	Требует больших вычислительных мощностей, сложен в настройке и интерпретации

Как показывает анализ, наилучшие характеристики, в рамках исследования для классификации тестовых случаев у метода K - ближайших соседей (KNN), метода опорных векторов (SVM) и наивного байесовского классификатора (NB) по следующим причинам:

- KNN подходит для обработки сложных и нелинейных данных, что важно при анализе тестовых случаев, поскольку их структура может варьироваться;
- SVM обеспечивает хорошее качество классификации, особенно для задач с четкими границами классов, что критично при разделении тестов по приоритетам;
- NB показывает высокую скорость работы, не требует больших объемов данных для обучения и хорошо справляется с анализом тестов, имеющих дискретные признаки.

Предложенные алгоритмы позволяют обеспечить баланс между точностью, скоростью работы и возможностью обработки сложных структур данных, что делает их оптимальными для исследования в рамках автоматизированного тестирования.

1.4 Обзор существующих методов генерации тестовых случаев на основе UML-диаграмм

Одним из направлений совершенствования тестирования является автоматическая генерация тестовых случаев на основе UML - диаграмм - данный процесс стал предметом множества научных исследований. Сам процесс генерации тестовых случаев из UML - диаграмм охватывает различные подходы, направленные на обеспечение максимального покрытия возможных сценариев и функционирования системы. Рассмотрим более подробно данные исследования.

В исследовании «Ismail, N., Ibrahim, R., & Ibrahim, N. Automatic Generation of Test Cases from Use-Case Diagram» был разработан интеллектуальный метод автоматического создания тестовых случаев на основе требований системы [25]. Предложенный инструмент реализует двухэтапный процесс генерации тестов. На первом этапе функциональные требования преобразуются в диаграммы вариантов использования, представленные унифицированным языком моделирования UML. Затем, на следующем этапе, автоматически создаются тестовые сценарии на основе вариантов использования. Однако, данный метод охватывает только функциональные требования и не учитывает нефункциональные аспекты системы, такие как производительность, безопасность и удобство использования. Кроме того, в рамках исследования не рассматривалось установление порядка очередности выполнения тестовых случаев, что могло бы повысить эффективность тестирования.

Следующий подход основан на применении диаграмм последовательностей для генерации тестов. Исследователи «Meiliana, Septian, I., Alianto, R. S., Daniel, & Gaol, F. L. Automated Test Case Generation from UML Activity Diagram and Sequence Diagram using Depth First Search Algorithm» преобразовали диаграмму последовательностей в граф последовательности, из которого затем извлекались тестовые сценарии [29]. Узлы графа

последовательности содержали информацию, полученную из диаграмм вариантов использования. Текущий метод обеспечивал полное покрытие всех возможных сообщений между компонентами системы. В качестве примера исследования рассматривалась работа банкомата и процесс проверки PIN-кода. Однако, несмотря на успешную генерацию тестовых сценариев, данная методика не включала механизм расстановки приоритетов тестов, что могло бы существенно улучшить их практическое применение.

Еще один исследовательский проект предложил использование комбинации диаграмм последовательностей и диаграмм состояний для генерации тестовых случаев [27]. Выбор этих UML-диаграмм обусловлен тем, что диаграмма последовательностей четко отображает взаимодействие между элементами системы, а диаграмма состояний фиксирует возможные переходы между состояниями. В рамках исследования диаграмма последовательностей трансформировалась в граф последовательностей, а диаграмма состояний — в граф состояний. Затем эти два графа объединялись в единый тестовый граф системы. Для улучшения созданных тестовых сценариев применялся генетический алгоритм, нацеленный на достижение наилучшего охвата системы и поиск потенциальных сбоев. В качестве испытательной среды использовалась платформа для онлайн - голосования. Эксперимент подтвердил, что совместное применение различных UML-диаграмм увеличивает полноту тестирования и способствует обнаружению распространенных ошибок, включая сбои в сценариях работы и неправильную обработку информации. При этом использованный метод имел и определенный недостаток: он приводил к созданию чрезмерного количества тестовых случаев, что впоследствии затрудняло определение их важности и очередности выполнения.

Другой метод автоматической генерации тестов основывается на тестировании серого ящика и использует диаграммы активности UML. Согласно исследованию «Teixeira, F. A. D., & Braga E Silva, G. Easytest: An approach for automatic test cases generation from UML activity diagrams» [34]

тестовые случаи формируются на основе проектных моделей высокого уровня, отражающих ожидаемую структуру и поведение программного обеспечения. В рамках данного подхода диаграмма активности применяется для отображения операций, где методы классов рассматриваются как активности, а классы - как дорожки активности (swimlanes). Для построения тестовых случаев используется базовый критерий покрытия пути, согласно которому каждый цикл должен выполняться лишь один раз. Данное требование покрытия помогает избежать большого роста количества тестовых путей, что является частой проблемой при работе с циклами.

Одним из наиболее актуальных направлений исследований является приоритизация тестов на основе моделей. В работе «Hemmati, H., Arcuri, A., & Briand, L. Achieving scalable model-based testing through test case diversity. ACM Transactions on Software Engineering and Methodology» [24] был реализован подход, в котором учитывались знания о модели системы и ее поведении для оптимизации процесса регрессионного тестирования. В рамках экспериментальной части работы проводилось сравнение различных стратегий ранжирования тестов по их способности выявлять критические дефекты на начальных стадиях разработки. Результаты демонстрируют, что применение модельно-ориентированных подходов в целом повышает результативность процесса тестирования. Вместе с тем, исследование показало, что отдельные категории модельных данных оказывают ограниченное влияние на раннюю диагностику ошибок. Любопытно, что часть простых эвристических методов продемонстрировала сопоставимую с сложными алгоритмами эффективность при сортировке тестов по приоритетам. С экономической точки зрения, модель - ориентированная приоритизация была признана потенциально более выгодной альтернативой классическим методикам, зависящим от анализа исходного кода. Для дальнейшей валидации полученных результатов авторы отмечали необходимость апробирования разработанного метода в условиях реальных проектов с использованием более сложных системных архитектур.

Существенный прорыв в области MBT - тестирования был достигнут благодаря методологии TOTEM [21]. TOTEM расшифровывается как «Тестирование объектно - ориентированных систем с использованием унифицированного языка моделирования». В данном методе функциональные требования для системного тестирования формируются на основе унифицированного языка моделирования UML, таких как диаграммы вариантов использования. В работе применили алгоритм поиска в глубину DFS, что позволяет извлекать последовательности в вариантах использования. Указанный подход обеспечивает формирование корректных тестовых сценариев, следуя предопределенным зависимостям в системе.

В текущем разделе мы провели обзор существующих подходов к автоматизированной генерации и приоритизации тестовых случаев, которые можно условно разделить на две основные категории: создание тестовых сценариев на основе UML-диаграмм и генерация тестов из исходного кода с использованием различных алгоритмов. В ходе анализа мы выявили основные ограничения современных исследований и определили направления, требующие дальнейшей проработки. В итоге на сегодняшний день значительное внимание уделяется разработке методов генерации тестовых случаев из диаграмм активности UML, однако в большинстве работ рассматриваются только вопросы их построения, а аспект приоритезации остается недостаточно проработанным. Проведенный обзор исследований показал, что существующие работы либо полностью игнорируют задачу классификации тестов по уровню приоритета, либо используют подходы, которые не учитывают особенности структуры тестируемой системы и специфики работы тестировщиков.

В связи с этим, данная диссертация направлена на восполнение существующих пробелов путем разработки комплексного подхода, включающего, как автоматическую генерацию тестовых случаев, так и установление приоритетов.

Таким образом, данное исследование представляет собой шаг к решению важной проблемы автоматизированного тестирования, связанной с оптимизацией тестовых сценариев и расстановки их по степени значимости. Несмотря на существование отдельных подходов к автоматической генерации тестов, комплексное решение, сочетающее методы построения тестовых случаев и алгоритмы их приоритизации, в современной научной литературе практически не рассматривается. Это подчеркивает актуальность выбранной темы и необходимость дальнейших исследований в данном направлении.

Таким образом, различные методы генерации и приоритизации тестов на основе UML-диаграмм демонстрируют значительный потенциал в повышении эффективности тестирования. Внедрение алгоритмов анализа графов и машинного обучения позволяет автоматизировать процесс тестирования и снизить затраты на его выполнение. В таблице 3 отражено краткое сравнение по результатам анализа методов автоматической генерации тестовых случаев.

Таблица 3 – Сравнение существующих методик автоматической генерации тестовых случаев на основе UML- диаграмм

Авторы	Название работы	Методы	Ограничения
Ismail and Ibrahim, 2014	Automatic Generation of Test Cases from UseCase Diagram	Применяются интеллектуальные методы поиска для автоматической генерации тестовых случаев в соответствии с требованиями системы	Разработанный инструмент для генерации тестовых случаев не позволяет учесть нефункциональные требования системы, при этом исследование не было направлено на выявление приоритетов среди тестовых случаев.

Продолжение таблицы 3

Авторы	Название работы	Методы	Ограничения
Monalisa Sarma, 2015	Automatic Test Case Generation from UML Sequence Diagrams	Применяется графическая методология для преобразования диаграммы последовательности UML в граф, известный как граф диаграммы последовательности, который впоследствии применялся для генерации тестовых случаев.	Применённая методология генерации тестовых случаев приводила к созданию избыточных тестовых сценариев. В рамках исследования не рассматривался вопрос расстановки приоритетов среди сгенерированных тестовых случаев.
Khurana and Chillar, 2019	Test Case Generation and Optimization using UML Models.	Применяется графическая методология для преобразования диаграмм последовательности и диаграмм состояний UML, дополнив её использованием генетического алгоритма для автоматизированной оптимизации тестовых случаев с учётом критериев покрытия и модели отказов.	Сгенерированные в рамках данного исследования тестовые случаи оказались избыточными и трудными для расстановки приоритетов. Кроме того, в исследовании не применялись методы машинного обучения для приоритизации тестовых сценариев.
Azaharuddin Ali and Khasim, 2013	Test Case Generation using UML State Diagram and OCL Expression	Используется диаграмма состояний UML и язык объектных ограничений (OCL) для генерации тестовых случаев, применяется методология, основанная на преобразовании диаграммы состояний UML в конечный автомат.	В исследовании отсутствует сравнение полученных результатов. Процесс объединения двух диаграмм остался нерешённой задачей и выполнялся в полуавтоматическом режиме. Кроме того, должное внимание оптимизации тестовых случаев и их приоритизации не было уделено.

Продолжение таблицы 3

Авторы	Название работы	Методы	Ограничения
Meiliana, I. Septian, R. S. Alianto, Daniel, and F. L. Gaol, 2017	Automated Test Case Generation from UML Activity Diagram and Sequence Diagram	Метод, представленный в их исследовании, основан на модифицированном алгоритме поиска в глубину (DFS), применяемом для генерации тестовых случаев на основе диаграмм активности и последовательности UML.	Объединение UML-диаграмм не обязательно приводит к улучшению качества тестовых случаев, так как может способствовать избыточной их генерации. В исследовании процессу приоритизации тестовых случаев уделено недостаточно внимания, поскольку основной акцент сделан исключительно на генерации тестовых случаев.
Dalai et al., 2012	Generate test cases for Objectoriented systems using Combinational UML Models	В данном подходе создаётся граф последовательности действий, который затем анализируется с целью генерации тестовых случаев, позволяющих минимизировать неконтролируемый рост их количества.	В данном подходе формируются диаграмма последовательности и граф активности, однако тестовые случаи остаются неоптимизированными. Методология для расстановки приоритетов среди сгенерированных тестовых случаев отсутствует.
Teixeira & Braga E Silva, 2018	An approach of gray-box testing using activity diagrams.	При тестировании по принципу «серого ящика» генерация тестовых случаев осуществляется на основе высокоуровневых моделей проектирования, отражающих предполагаемую структуру и поведение тестируемого программного обеспечения.	Не уделено достаточного внимания вопросам оптимизации тестовых случаев и их приоритизации. Кроме того, их основной критерий покрытия пути сформулирован недостаточно чётко, что затрудняет его понимание и применение.

Продолжение таблицы 3

Авторы	Название работы	Методы	Ограничения
Hemmati et al., 2013	Approach for generating and executing system tests.	Представленный подход моделирует поведение системы с использованием диаграмм вариантов использования UML и диаграмм активности. Кроме того, рассматриваются различные типы аннотаций модели, которые могут быть добавлены разработчиком тестов на этапе подготовки перед генерацией тестовых случаев.	В данном исследовании основной акцент сделан на процессе генерации тестовых случаев, тогда как вопросам их приоритизации уделено недостаточно внимания.

Анализ существующих исследований показывает, что современные методы автоматической генерации тестовых случаев обладают рядом ограничений, снижающих их эффективность в практическом применении. Присутствует недостаточное внимание к приоритизации тестовых случаев так как большинство методов сосредоточены на генерации тестов, но не учитывают важность ранжирования тестов по критичности и вероятности обнаружения ошибок.

Избыточность сгенерированных тестов в рассмотренных работах показывают, что автоматическая генерация тестов без оптимизации может создавать избыточные сценарии, что увеличивает время тестирования и снижает его эффективность.

Сравнение существующих методик показывает, что ни один из существующих подходов не решает одновременно проблемы избыточности тестов, их приоритизации и оптимизации. Наша методика объединяет алгоритмы DFS и ML, что обеспечит более эффективное и интеллектуальное тестирование программных систем.

Выводы к первой главе

В первой главе мы провели детальный анализ текущего состояния проблемы автоматизации тестирования программного обеспечения с акцентом тестирования на основе моделей. Рассмотрены основные подходы, включая использование UML - диаграмм, таких как диаграммы активности, последовательности и состояний, для генерации и приоритизации тестовых случаев. Показано, что использование тестирования на основе моделей позволяет существенно сократить временные и финансовые затраты, а также улучшить покрытие требований за счет автоматической генерации тестовых случаев из формализованных моделей. Обоснована необходимость применения автоматизации тестирования на ранних этапах жизненного цикла разработки программного обеспечения. Так как данный процесс способствует более раннему выявлению и исправлению ошибок, что повышает общую надежность и качество программных продуктов. Кроме того, в первой главе рассмотрены современные методы тестирования, алгоритмы поиска в глубину, подходы к генерации тестовых случаев и их приоритизации. Все эти аспекты формируют методологическую основу исследования, направленного на совершенствование процесса тестирования программного обеспечения.

Глава 2 Разработка и обоснование методологии исследования

Во второй главе подробно рассмотрим ключевые этапы исследования, начиная с анализа деятельности компании, в рамках которой планируется внедрение предложенной модели. Далее опишем процесс сбора, подготовки и предварительной обработки данных, включая их очистку, нормализацию и представление в формате XML для последующей обработки. Особое внимание уделим детальному описанию структуры предлагаемой модели, охватывающей все этапы - от анализа UML - диаграмм активности до генерации и приоритизации тестовых случаев. Важной составляющей методологии является классификация тестов по уровню приоритета, реализуемая с применением методов машинного обучения. В заключительной части главы проводится оценка эффективности предложенной модели, анализируются полученные результаты и сравниваются с существующими подходами. Таким образом, данная глава охватывает полный процесс разработки предлагаемой методологии.

2.1 Анализ деятельности компании ООО «Стильные кухни»

ООО «Стильные кухни» российская компания, специализирующаяся на производстве и продаже мебели для кухонь. Компания предлагает широкий ассортимент стильных и функциональных решений, отвечающих самым современным требованиям комфорта, эстетики и качества. «Стильные кухни» это не просто производитель мебели, а полноценный сервисный центр, обеспечивающий полный цикл услуг: от разработки индивидуального дизайна и проектирования до доставки, сборки и гарантийного обслуживания. Одним из ключевых направлений развития компании является цифровизация клиентского опыта. Благодаря внедрению современных ИТ - решений, клиенты получают удобные инструменты для взаимодействия с компанией, включая интернет - магазин, личный кабинет, чат-боты и интеграцию с

мессенджерами и социальными сетями. В интернет - магазине клиенты компании могут ознакомиться с предлагаемым ассортиментом, а также выбрать оптимальные решения с учетом своих предпочтений. В каталоге представлена подробная информация о каждом товаре, включая технические характеристики, фотографии, отзывы клиентов и рекомендации специалистов.

Для повышения удобства использования внедрена система личного кабинета, которая представлена на рисунке 6. Данная система обеспечивает комплексное управление заказом и взаимодействием клиента с компанией. В личном кабинете доступны следующие функции:

- статус заказа: отображение всех этапов производства, от момента оформления до доставки и установки. Покупатели могут следить за выполнением заказа онлайн, наблюдая каждый этап его готовности;
- в разделе с документами размещены все необходимые формы договоров на покупку, доставку и установку. Это позволяет ознакомиться с условиями до оформления заказа;
- после покупки доступно гарантийное обслуживание: можно оформить заявку на ремонт, заменить детали по гарантии или получить консультацию специалиста;
- бонусная программа: для поощрения постоянных клиентов действует программа лояльности с накопительной системой бонусов, которые можно потратить на скидки или специальные подарки;
- подарочные сертификаты: удобная возможность приобрести электронные подарочные сертификаты;
- избранные товары: функция сохранения понравившихся товаров для удобного сравнения и последующего оформления заказа;
- персонализированные рекомендации: интеллектуальная система рекомендаций, формирующая предложения на основе анализа пользовательских предпочтений;
- - настройки профиля: завершает функционал гибкая настройка персонального профиля.

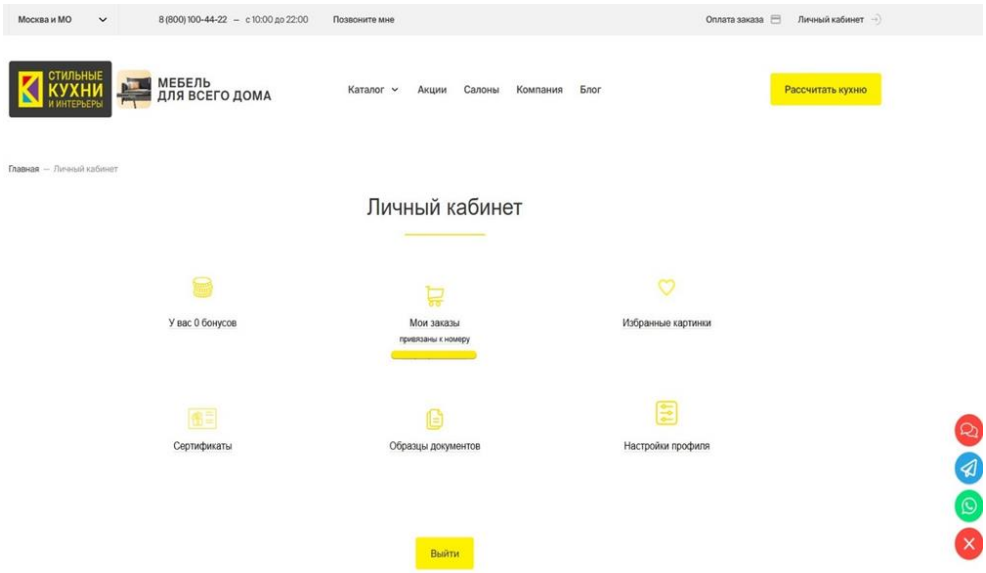


Рисунок 6 – Личный кабинет пользователя

Интернет - магазин (рисунок 7) дополнен специализированными разделами, повышающими вовлеченность аудитории, а именно:

– каталог продукции: детализированный каталог кухонь с интуитивной навигацией, позволяющий фильтровать товары по стилю, цветовой гамме, материалам и ценовому диапазону;
– формат сторис (истории): краткие видеообзоры новых коллекций, акционных предложений, а также материалов о производстве и установке кухонь;
– отзывы клиентов: блок с отзывами, содержащий фото- и видеоматериалы с реализованными проектами в интерьерах клиентов. Представлены на рисунке 8.
– блог от хоумстейджеров: экспертный блог с авторскими материалами от хоумстейджеров о дизайне интерьеров, организации пространства и критериях выбора мебели (рисунок 9).

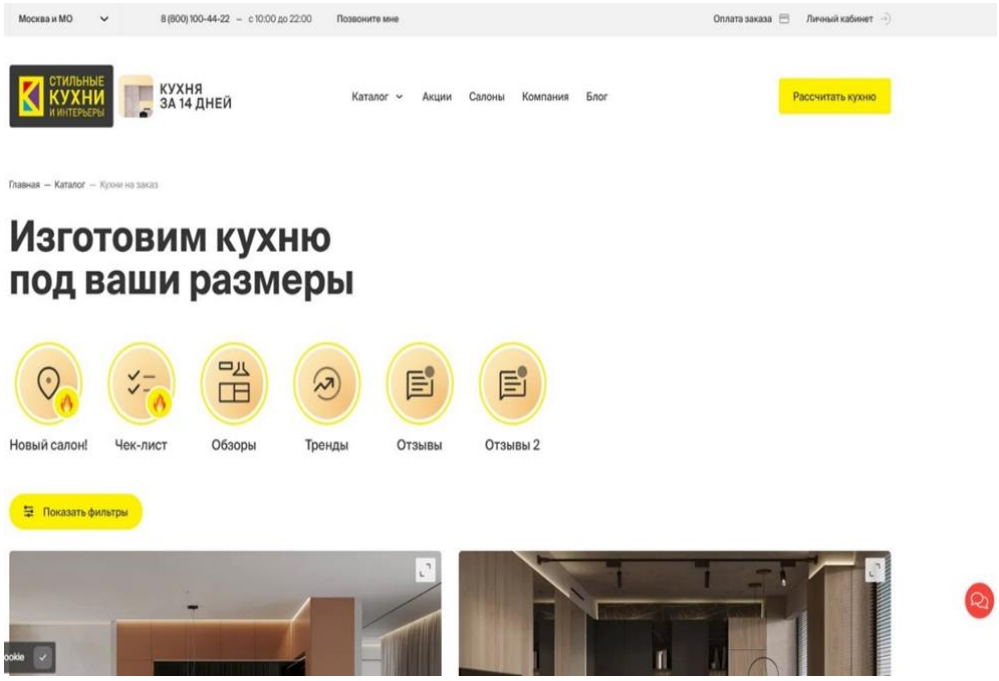


Рисунок 7 – Интернет - магазин «Стильные кухни»

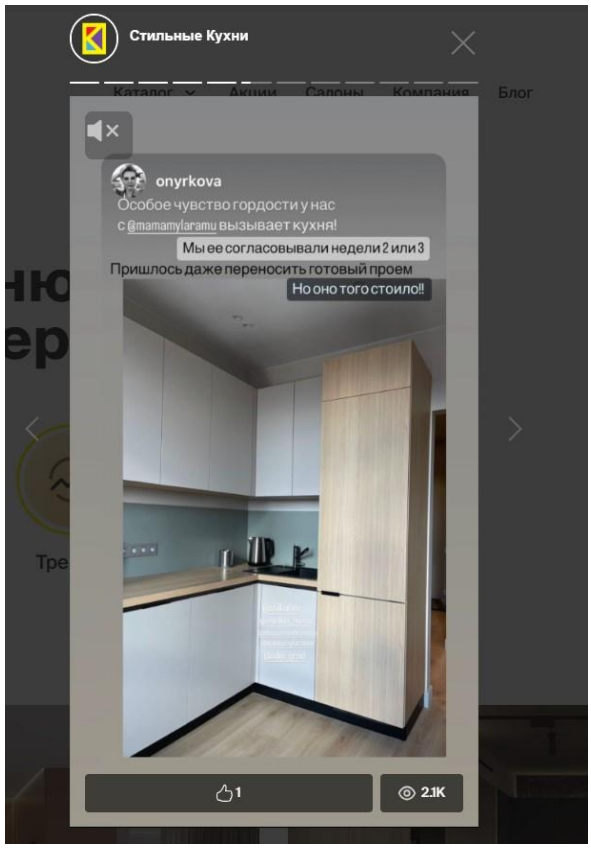


Рисунок 8 – Сторисы с отзывами

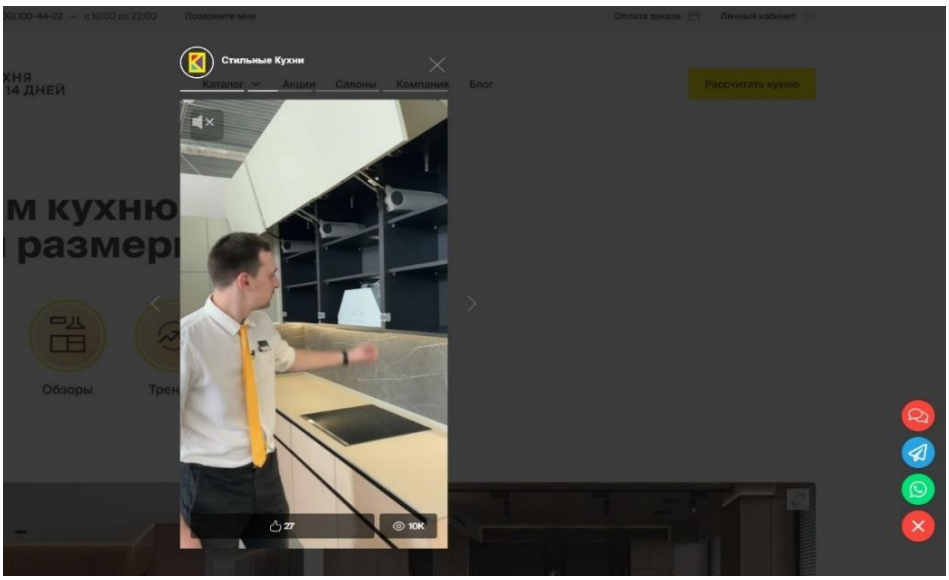


Рисунок 9 – Блог от хоумстейджеров

Для оперативного взаимодействия с клиентами компания «Стильные кухни» внедрила коммуникацию через популярные мессенджеры, такие как:

- Whats app для прямых консультаций, уточнения деталей заказа и оформления покупок в чате;
- бот в Telegram: Telegram - бот с автоматизированными уведомлениями о статусах заказов, персональными предложениями и технической поддержкой;
- чат - бот на сайте: интеллектуальный чат-бот на сайте, помогающий в подборе кухни по параметрам, оформлении заказов и записи на консультации к дизайнерам. Интерфейс чат-бота показан на рисунке 10.

Такое многоканальное присутствие позволяет компании гибко подстраиваться под привычные для разных групп клиентов способы общения. Мгновенные уведомления в Telegram значительно сокращают время на информирование покупателей, а интеллектуальный помощник на сайте берет на себя рутинные операции, высвобождая время менеджеров для решения сложных задач. Внедрение единой экосистемы коммуникаций позволило не только ускорить обработку запросов, но и выстроить персонализированное взаимодействие с клиентом на всех этапах жизненного цикла заказа.

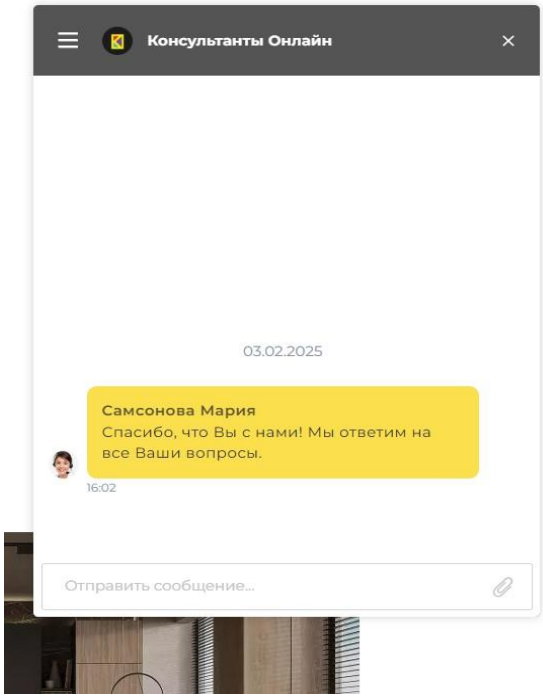


Рисунок 10 – Чат - бот на сайте

Представленные цифровые сервисы делают процесс покупки более комфортным, повышают доверие к бренду и помогают клиентам получать актуальную информацию о продукции и услугах. В результате компания «Стильные кухни» успешно сочетает традиционное производство с современными ИТ - решениями, предоставляя клиентам удобный и интуитивно понятный сервис на всех этапах взаимодействия. Данный подход позволяет не только повысить лояльность клиентов, но и укрепить позиции компании на рынке.

2.2 Сбор и подготовка данных

В рамках проводимого исследования были задействованы наборы данных компании ООО «Стильные кухни», подробно рассмотренной в параграфе 2.1. Для реализации предложенного метода были отобраны UML - диаграммы активности, отражающие ключевые сценарии взаимодействия

пользователя с платформой. Особое внимание уделено UML - диаграммам активности, описывающим такие процессы, как оформление заказов через интернет-магазин (рисунок 7), регистрация нового пользователя, интеграция с мессенджерами, обновление каталога товаров, работа личного кабинета пользователей и функционирование бонусной системы. На рисунке 11 представлен пример UML диаграммы деятельности, которая показывает последовательность действий при покупке в интернет-магазине https://www.stilkuhni.ru/.

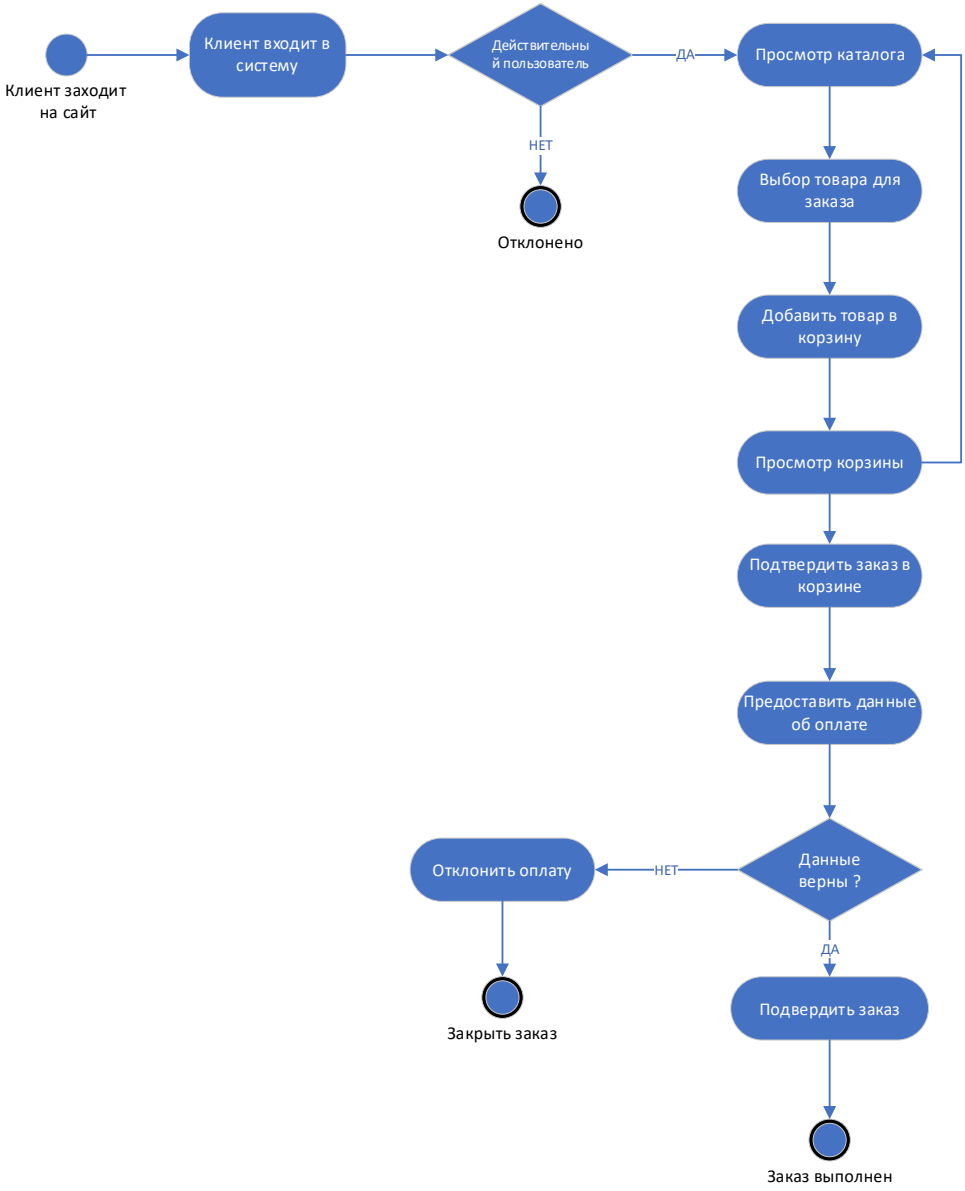


Рисунок 11 – Диаграмма активности для интернет-магазина

В рамках предварительной обработки диаграммы активности UML преобразуются в соответствующую таблицу зависимостей, а затем - в граф взаимосвязей, представленный в виде массива, который используется в работе предлагаемой модели [32]. Для корректной классификации тестов по уровню приоритета необходимо сначала преобразовать данные в удобно читаемый формат и организовать их в виде таблицы. Для выполнения перечисленных операций использовался язык программирования Python, который позволил автоматизировать обработку данных и подготовить их к дальнейшему применению в рамках предлагаемого метода.

Нормализация данных играет ключевую роль при применении алгоритмов классификации, таких как SVM, KNN и NB, которые требуют использования нормализованных данных в диапазоне от 0 до 1. Среди доступных методов нормализации данных выделяются методы минимально - максимальной нормализации, Z - преобразование и нормализация на основе среднего значения. В рамках нашего исследования для нормализации использовался метод минимально - максимальной нормализации, который выполняет линейное преобразование исходных данных.

Сравним и проанализируем методы нормализации данных, представим в виде таблицы (таблица 4) анализ, почему нами был выбран метод минимально-максимальной нормализации. Данный подход был избран в связи с его способностью сохранять исходное распределение данных, не искажая отношения расстояний между точками, что особенно важно для метрических алгоритмов, таких как KNN. Кроме того, метод обеспечивает интерпретируемость результатов, преобразуя признаки в единый диапазон без изменения формы распределения. Линейный характер преобразования также гарантирует вычислительную эффективность, что становится критичным при обработке массивных наборов данных в реальных производственных сценариях.

Таблица 4 – Сравнения методов нормализации данных

Метод нормализации	Описание	Плюсы	Минусы	Почему не выбран
Минимально - максимальная нормализация	Линейное преобразование данных в диапазон [0, 1] или [-1, 1]	Сохраняет отношения между данными. Удобен для алгоритмов чувствительных к масштабу	Чувствителен к выбросам (экстремальным значениям)	Выбран как оптимальный
Z - преобразование	Преобразование данных так, чтобы их распределение имело среднее значение = 0 и стандартное отклонение = 1	Устойчив к выбросам. Применим для данных с нормальным распределение	Может искажать данные, если они не имеют нормального распределения	Не подходит для данных с неравномерным распределение
Нормализация на основе среднего значения	Преобразование данных так, чтобы их среднее значение стало 0, а значения варьировались в диапазоне [-1, 1]	Подходит для алгоритмов, чувствительных к средним значениям	Может быть неэффективен при наличии выбросов	Не обеспечивает оптимального масштаба для классификации
Робастная нормализация	Масштабирование с использованием медианы и интерквантильного диапазона (IQR)	Устойчив к выбросам. Сохраняет структуру данных	Менее эффективен при наличии большого количества выбросов	Может потерять точность при неравномерном распределении

Процесс минимально - максимальной нормализации преобразует фактические значения атрибутов в нормализованные значения, лежащие в диапазоне от 0 до 1. Этот метод рассчитывается по следующей формуле (1):

$$norm(x) = \frac{(x - min(x))}{(max(x) - min(x))} \quad (1)$$

где x - текущее значение признака;

$norm(x)$ - функция приведения значений атрибута x в диапазон от 0 до 1;

$min(x)$ - минимальное значение признака в выборке;

$max(x)$ - максимальное значение признака в выборке.

Как интерпретировать нормализованные значения?

Если $(x) = min(x)$, то $norm(x) = 0$ (самое маленькое значение превращается в 0).

Если $(x) = max(x)$, то $norm(x) = 1$ (самое большое значение превращается в 1).

Все остальные значения x будут находиться в диапазоне (0, 1).

Минимально - максимальная нормализация нужна в машинном обучении для сравнения признаков: если один признак измеряется в миллионах, а другой в десятых долях, это может привести к некорректным результатам в моделях.

2.3 Предлагаемая модель

Процесс разработки предлагаемой модели можно разделить на несколько ключевых этапов:

- сбор диаграмм активности: диаграммы активности создаются с использованием инструмента моделирования UML, такого как Enterprise Architect. Этот инструмент позволяет визуализировать процессы и действия, необходимые для описания поведения системы;
- экспорт диаграмм активности в формат XML: после создания диаграмм они экспортируются из Enterprise Architect в стандартный XML - формат. Это обеспечивает совместимость данных с дальнейшими этапами обработки;
- генерация таблицы зависимостей действий: из экспортированных диаграмм активности формируется таблица зависимостей действий. Эта таблица представляет собой структурированное отображение взаимосвязей между различными действиями, указанными в диаграмме;

- построение графа зависимостей действий: на основе таблицы зависимостей создаётся граф зависимостей действий. Текущий этап является важным шагом для формирования тестовых путей, поскольку граф зависимостей наглядно демонстрирует возможные последовательности выполнения действий в системе;
- построение тестовых путей: на данном этапе происходит преобразование графа зависимостей в тестовые сценарии. Для этого применяется алгоритм обхода графа в глубину, который систематически исследует все возможные пути выполнения от начального до конечного состояния системы. Такой подход гарантирует достижение максимального покрытия функциональности тестами, поскольку алгоритм последовательно анализирует каждую ветвь логики системы, включая редко используемые и альтернативные сценарии работы;
- нормализация данных: все данные приводятся к единому стандартному виду с использованием методов нормализации, таких как метод минимально-максимальной нормализации. Данный процесс обеспечивает корректность данных перед их классификацией;
- применение методов классификации: для классификации тестовых случаев по уровням приоритета используются алгоритмы машинного обучения, такие как метод опорных векторов SVM, K - ближайших соседей KNN и наивный байесовский классификатор NB. Данные алгоритмы помогают разделить тестовые случаи на категории по степени их значимости;
- оптимизация и расстановка по приоритетам тестовых случаев: на заключительном этапе формируется оптимизированный набор тестовых случаев, который учитывает назначенные приоритеты. Такой подход позволяет минимизировать затраты времени и ресурсов на тестирование, при этом обеспечивая высокий уровень покрытия функциональности системы.

На рисунке 12 изображен бизнес - процесс оптимизации и приоритизации тестовых случаев с использованием предлагаемой модели.

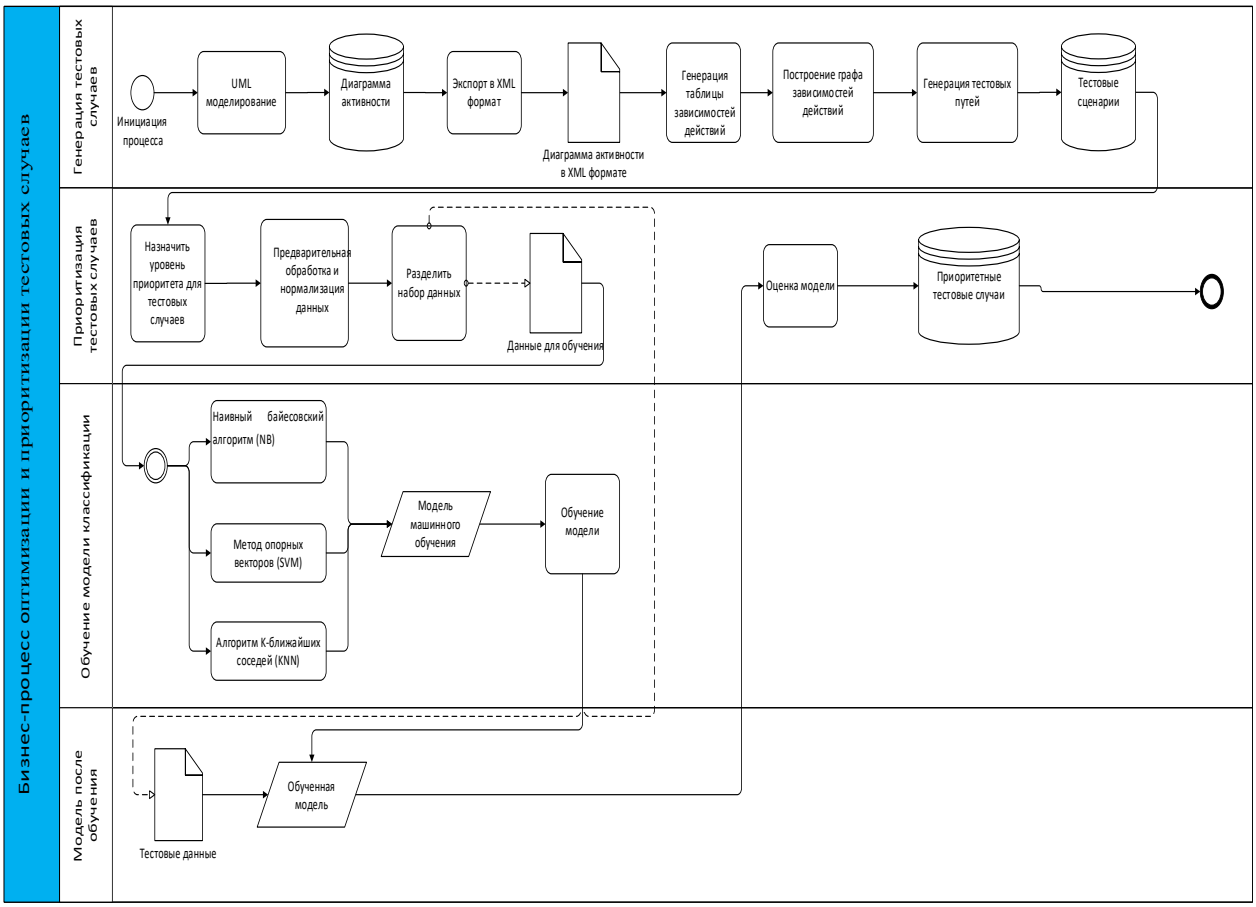


Рисунок 12 – BPMN - диаграмма предлагаемой модели

В результате внедрения предлагаемой модели автоматизированной генерации и приоритизации тестовых случаев из UML-диаграмм активности мы ожидаем значительные улучшения в процессе тестирования интернет - магазина, а именно:

– сокращение времени на тестирование: автоматическая генерация тест - кейсов позволила бы исключить ручную разработку тестовых сценариев, что как мы ожидаем приведёт к снижению временных затрат на тестирование системы в среднем на 35% процентов. Приоритизация тестов с использованием машинного обучения обеспечила бы выполнение наиболее критичных тестов в первую очередь;

– повышение качества тестирования: полное покрытие тестовыми случаями всех возможных сценариев взаимодействия пользователей с интернет-магазином, включая пограничные случаи и исключительные ситуации. Предлагаемая модель позволяет автоматически покрывать все возможные пути выполнения системы и обеспечивает практически 100% покрытие состояний и переходов по заранее заданным критериям, что вручную сделать крайне сложно. А также исключение человеческого фактора и ошибок в тестировании за счет автоматической обработки данных. Точность тестирования возрастает, так как тест - кейсы создаются на основе моделей, исключая человеческий фактор;
– улучшение устойчивости системы интернет - магазина: выявление и исправление ошибок до запуска обновлений или выполнения заказов, что минимизирует вероятность сбоя в работе системы;
– оптимизация бизнес - процессов: снижение времени на исправление ошибок и доработку системы благодаря автоматической классификации и расстановке приоритетов. А также увеличение скорости обработки заказов в интернет-магазине и снижение количества задержек из-за технических проблем.

Рассмотрим более подробно каждый этап в предлагаемой модели в следующем параграфе.

2.3.1 Экспорт диаграммы активности в формат XML

На данном этапе диаграмма активности будет преобразована в формат XML, что позволяет извлекать необходимую информацию для процесса генерации тестовых случаев. В указанный процесс входят свойства, описывающие активности (вершины) и потоки управления (рёбра) в системе. Преобразованный XML - формат служит основой для дальнейшей обработки, а именно для создания таблицы зависимостей активностей (Activity Dependency Table, ADT), которая структурирует данные о взаимосвязях между элементами диаграммы.

2.3.2 Преобразование диаграммы активности

В данном разделе описывается метод преобразования диаграммы активности в таблицу зависимостей активностей (Activity Dependency Table, ADT). Диаграмма активности представляет собой визуализацию последовательности задач в сценарии использования или бизнес - процессе, а также может использоваться для отображения логики системы. Для создания таблицы зависимостей из диаграммы активности собираются данные, необходимые для формирования узлов. Каждый узел обладает следующими характеристиками:

- идентификатор – это уникальный номер, который используется для идентификации каждой активности,
- имя активности – наименование каждой задачи или действия, отображаемого на диаграмме,
- цель активности – указание на следующую активность, необходимую для построения списка смежности,
- статус активности – состояние задачи после выполнения компонента решения.

Приведем описанием алгоритма на естественном языке, данный алгоритм можно адаптировать для написания программы на таких языках, как Python или Java. Этот фрагмент кода реализует метод преобразования UML - диаграммы активности в таблицу зависимостей (Activity Dependency Table, ADT). Алгоритм автоматически определяет связи между отдельными операциями на диаграмме и формирует табличное представление этих связей. Полученная таблица служит удобной основой для последующего анализа и создания тестовых сценариев. Опишем более подробно как работает псевдокод алгоритма:

Псевдокод алгоритма преобразования UML – диаграммы в ADT представлен на рисунке 13.

```
Algorithm name: Activity dependency table (ADT)
INPUT: UML activity diagram
OUTPUT: Activity dependency table (ADT)
START
ADD Column (Activities/nodes, dependency, weight, ID)
IF node i is = connected to node j
FOR x IN range (0, n)
   Add-row (dependency.value)
   Add-row (weight.value)
   Add-row (activity.value)
   Add-row (ID)
END for loop
ELSE
   go to the next node
END
```

Рисунок 13 – Псевдокод алгоритма преобразования UML – диаграммы в ADT

В представленном псевдокоде *INPUT* входные данные: UML-диаграмма активности; *OUTPUT*: выходные данные таблица зависимостей активностей (ADT), включающая столбцы:

- *activities/nodes* - список узлов (активностей) диаграммы;
- d*ependency* - зависимости между узлами;
- w*eight* - значение зависимости (может быть использован для приоритизации);
- *ID* - уникальный идентификатор каждой активности.

Основной процесс:

инициализация таблицы с указанными столбцами;

– перебор всех узлов в диапазоне от 0 до n (активностей) в диаграмме активности;
– для каждого узла проверяется, существует ли связь с другим узлом. Если связь существует, то извлекаются данные о зависимости, весе, имени активности и идентификаторе узла. Эти данные добавляются как строки в таблицу зависимостей. Если узел *i* не связан с узлом *j*, перейти к следующему узлу;
– окончание работы: после обработки всех узлов формируется полная таблица зависимостей.

Этот метод позволяет наглядно отобразить связи между компонентами диаграммы активности, представляя их в виде упорядоченной таблицы.

2.3.3 Создание графа зависимостей

Граф зависимостей активностей (Activity Dependency Graph, ADG) создается автоматически из таблицы зависимостей ADT. Каждое действие в графе изображается в виде узла с уникальным буквенным обозначением. Связи между узлами показывают переходы и очередность выполнения операций.

Для построения ADG применяются данные из колонки «зависимости» таблицы ADT. Если в этой колонке для текущего действия указан символ предыдущего узла, между ними формируется направленная связь. Такой подход позволяет наглядно представить порядок выполнения операций в системе. Таким образом, ADG наглядно демонстрирует взаимосвязи между действиями в системе. Особое внимание уделяется обработке узлов решений и разветвлений. Если такой узел удаляется, его зависимость автоматически передаётся следующему узлу, от которого зависит текущее действие. Это гарантирует целостность структуры ADG и позволяет корректно отображать последовательности действий даже в сложных сценариях с ветвлениями. Приведем описанием алгоритма на естественном языке, данный алгоритм можно адаптировать для написания программы на таких языках, как Python или Java. Опишем более подробно как работает псевдокод алгоритма:

Псевдокод алгоритма преобразования таблицы ADT в граф зависимостей ADG представлен на рисунке 14.

```
Algorithm name: Activity dependency Graph (ADG)
INPUT: UML Activity dependency table (ADT)
OUTPUT: Activity dependency Graph
START
Graph(G) =Directed graph ()
ADD edges and nodes to the graph
Level= true
IF node && edges
   Draw-a line nodes
   Draw-a line edge display
The graph () //get different attributes
RETURN number of nodes
RETURN number of edges
END
```

Рисунок 14 – Псевдокод алгоритма преобразования таблицы ADT в граф зависимостей ADG

В представленном псевдокоде входные данные: UML Activity Dependency Table (ADT) - таблица зависимостей активности, содержащая информацию о действиях, их зависимостях и связанных атрибутах; выходные данные Activity Dependency Graph (ADG) – направленный граф, представляющий зависимости между действиями.

Опишем более подробно шаги алгоритма.

Инициация графа: Graph(G) = Directed graph (); создаётся пустой направленный граф G, который будет содержать узлы (действия) и рёбра (зависимости).

Добавление узлов и рёбер в граф: Add edges and nodes to the graph; узлы и рёбра добавляются в граф G на основе данных из таблицы ADT: узлы соответствуют действиям, описанным в ADT. Рёбра указывают на зависимости между действиями (например, если действие A зависит от действия B, то между узлами A и B будет добавлено ребро).

Уровневый анализ: Level = true; Этот шаг может быть использован для управления уровнями или приоритетами узлов. Например, уровни могут обозначать последовательность выполнения или зависимости между действиями.

Построение рёбер и отображение узлов:

- If node && edges;
- Draw-a line nodes;
- Draw-a line edge display.

Если узлы и рёбра присутствуют, они визуально отображаются: узлы (nodes) добавляются в граф как вершины. Рёбра (edges) прорисовываются между узлами, чтобы показать зависимости.

Извлечение атрибутов графа: The graph () // get different attributes; из графа можно извлечь атрибуты, такие как количество узлов и рёбер. Эти данные полезны для анализа структуры графа.

Возврат результата:

- Return number of nodes;
- Return number of edges.

Возвращаются ключевые характеристики построенного графа: общее количество узлов (действий) и рёбер (зависимостей).

Завершение: алгоритм завершает выполнение, предоставляя готовый граф зависимостей активности (ADG).

Алгоритм построения графа зависимостей активности ADG позволяет преобразовать таблицу зависимостей активности ADT в наглядное графическое представление зависимостей между действиями. В результате работы алгоритма формируется направленный граф, который становится

основной структурой для исследования последовательностей операций и взаимосвязей между компонентами системы. Данный подход значительно улучшает визуальное представление архитектуры системы и упрощает администрирование зависимостей между ее элементами.

2.3.4 Генерация тестовых случаев

Для создания тестовых сценариев применяется алгоритм поиска в глубину (DFS). Этот метод обеспечивает последовательное исследование всех вершин графа зависимостей (ADG), что гарантирует их полный охват тестированием. В процессе работы алгоритм выполняет извлечение всех возможных маршрутов выполнения с одновременным исключением неперспективных направлений проверки. Сформированные тестовые маршруты охватывают все ответвления графа, что позволяет провести комплексный анализ системной логики. Для исключения циклических прохождений и повторной обработки узлов реализован механизм маркировки посещенных вершин. Каждому узлу графа присваивается специальный флаг, который отслеживает его статус в рамках текущего тестового маршрута. Такой процесс позволяет тестировать все уникальные маршруты графа, исключая избыточность и обеспечивая соответствие критериям покрытия. Каждый тестовый маршрут представляет собой упорядоченную последовательность действий - от начального узла до завершающего. Данные последовательности наглядно демонстрируют процесс выполнения операций в тестируемой системе, что способствует выявлению возможных ошибок. Приведем описанием алгоритма на естественном языке, данный алгоритм впоследствии адаптируем для написания программы на таких языках, как Python или Java. Опишем более подробно как работает алгоритм.

Псевдокод генератора тестовых случаев при помощи алгоритма поиска в глубину DFS представлен на рисунке 15.

```
Algorithm Name: Test Case Generator
INPUT: All ADT's, ADG's of Activity Diagrams, and empty table TC
OUTPUT: Test Cases
START
  FOR EACH graph G IN the set of ADG's
    P [ ] = GetALLpaths(G) // P [ ] = {P [1], P [2], ------P[N]}; where P [1]
    is the first path and P[N] is the last one in the G.
    Set j=0 //counter for the paths in P.
    FOR EACH path P[j] in G do
      Set i=0 //counter for nodes in each path P[j].
      ADD a new row in TC.
      FOR EACH node N[i] in P[i] do
        ADD a new row in TC[j]
        Get the input and expected output of N[i] from the corresponding ADT.
        Put the input of N[i] under the column "Node Input" in TC[j][i].
        Put the expected output of N[i] under the column "Node expected Output"
in TC[j][i].
        i++
      END-FOR
        Put the input of N[0] under the column "Test case Input" in TC[j].
        Put the expected output of N[i-1] under the column "Test case Expected
Output" in TC[j].
        j++
      END-FOR
    END-FOR
    Return TC
  STOP
```

Рисунок 15 – Псевдокод генератора тестовых случаев при помощи алгоритма поиска в глубину DFS

В представленном псевдокоде:

– обход графов зависимости активностей ADG: алгоритм обрабатывает каждый граф зависимости активностей (G) из набора ADG. Из каждого графа извлекаются все возможные пути выполнения (P), где каждый путь представлен как последовательность узлов;

- обработка каждого пути в графе (P[j]): для каждого пути из набора извлекается последовательность узлов, и создаётся новая строка в таблице тестовых случаев (TC) для текущего пути;
- обработка узлов внутри пути (N[i]): алгоритм выполняет следующие действия для каждого узла в текущем пути, а именно извлекает входные и ожидаемые выходные данные узла из соответствующей таблицы ADT;
- записывает входные данные узла в колонку «Node Input» и ожидаемые выходные данные в колонку «Node Expected Output» текущей строки таблицы TC;
- происходит формирование тестового случая: после обработки всех узлов пути входные данные первого узла (N [0]) записываются в колонку «Test Case Input». Ожидаемые выходные данные последнего узла (N[i-1]) записываются в колонку «Test Case Expected Output»;
- происходит цикл, повторение для всех графов: после завершения обработки всех путей текущего графа, алгоритм переходит к следующему графу;
- возврат результата: после завершения обработки всех графов ADG, алгоритм возвращает таблицу тестовых случаев «TC».

Таблица тестовых случаев будет содержать: каждую строку, представляющую тестовый сценарий. Полную последовательность узлов, их входные и ожидаемые выходные данные. Начальные и итоговые данные для тестового случая.

Тестовые случаи будут обеспечивать полное покрытие всех возможных сценариев выполнения действий в системе, что гарантирует выявление ошибок и подтверждение соответствия системы требованиям.

2.3.5 Классификация тест-кейсов по уровню приоритета

Рассмотрим процесс классификации уровня приоритета тестовых случаев. После генерации тестовых случаев следующим этапом является назначение каждому из них соответствующего приоритета: высокого, среднего или низкого.

Для дальнейшей классификации используем методы машинного обучения. Будем применять следующие алгоритмы:

- метод опорных векторов SVM,
- метод K-ближайших соседей KNN,
- наивный байесовский классификатор NB.

Методы машинного обучения были выбраны из - за их высокой эффективности в задачах классификации [4]. Перед применением алгоритмов машинного обучения данные будут подвергнуты процессу нормализации. Это связано с тем, что такие алгоритмы, как SVM, KNN и NB, требуют нормализованных данных, значения которых находятся в диапазоне от 0 до 1. Нормализация данных позволяет повысить точность и эффективность классификации.

2.3.6 Оценка модели

Оценка разработанной модели играет важную роль в подтверждении её эффективности и точности. Данный процесс включает анализ параметров оценки модели и её результатов.

Для оценки системы применяется метрика, которая сравнивает количество данных, классифицированных моделью правильно и ошибочно [14].

Сравнение проводится между результатами, полученными с помощью предложенной модели, и данными, размеченными вручную.

Точность классификации выступает основным показателем эффективности модели. Дополнительно применяются такие метрики, как полнота (Recall) и F1 - Score, которые дают более детальное понимание производительности классификации и используемых алгоритмов машинного обучения.

Эти показатели помогают оценить, насколько успешно модель справляется с задачей классификации сгенерированных тестовых случаев.

2.3.7 Матрица ошибок

Матрица ошибок является интуитивно понятной и простой метрикой, используемой для определения корректности и точности алгоритмов в задачах классификации [35]. Для нашего исследования задача классификации много классовая, что означает наличие более двух меток классов. В данном случае матрица путаницы имеет размер 3×3, поскольку целевой класс включает три категории.

Рассмотрим элементы матрицы путаницы:

- истинно положительные (True Positives, TP): случаи, когда модель правильно предсказала положительный класс;
- истинно отрицательные (True Negatives, TN): случаи, когда модель правильно предсказала отрицательный класс;
- ложно положительные (False Positives, FP): случаи, когда модель ошибочно классифицировала отрицательный класс, как положительный (ошибка типа I);
- ложно отрицательные (False Negatives, FN): случаи, когда модель ошибочно классифицировала положительный класс, как отрицательный (ошибка типа II).

Матрица ошибок дает возможность провести детальную диагностику эффективности работы классификатора, наглядно отображая распределение верно и неверно распознанных примеров. Этот аналитический инструмент играет ключевую роль в выявлении областей наибольшей результативности алгоритма и зон, требующих дополнительной оптимизации.

Точность классификации измеряет долю правильно классифицированных образцов от общего числа образцов в наборе данных. Значение точности варьируется от 0 до 1, где 1 соответствует наилучшему результату. Данный метод рассчитывается по следующей формуле (2):

$$Accuracy = (TP + TN) / (TP + TN + FP + FN) \quad (2)$$

где, TP (True Positives) - количество истинно положительных результатов, классификатор верно отнёс объект к рассматриваемому классу;

TN (True Negatives) - количество истинно отрицательных результатов, классификатор верно утверждает, что объект не принадлежит к рассматриваемому классу;

FP (False Positives) - количество ложноположительных результатов, классификатор неверно отнёс объект к рассматриваемому классу;

FN (False Negatives) - количество ложноотрицательных результатов, классификатор неверно утверждает, что объект не принадлежит к рассматриваемому классу.

Точность классификации является ключевым показателем эффективности модели, отражающим долю корректно классифицированных образцов. Она позволяет оценить, насколько хорошо классификатор справляется с различением объектов между классами.

Полнота или истинно положительный показатель отражает долю истинно положительных случаев, которые модель смогла правильно классифицировать. Она демонстрирует, насколько полно модель охватывает все возможные положительные случаи. Данный показатель рассчитывается по следующей формуле (3):

$$Recall = TP / (TP + FN) \quad (3)$$

где, TP (True Positives) - количество истинно положительных результатов. Это ситуации, когда классификатор правильно предсказал положительный класс;

FN (False Negatives) - количество ложноотрицательных результатов. Это ситуации, когда классификатор неверно отнес объект к отрицательному классу, хотя он на самом деле положительный.

Recall - это метрика, показывающая долю правильно предсказанных положительных случаев (TP) от всех фактических положительных случаев (TP + FN). Она показывает нам какую долю объектов положительного класса из всех объектов положительного класса нашел алгоритм. Чем выше значение Recall, тем лучше классификатор справляется с задачей обнаружения всех объектов положительного класса. Recall является важной метрикой, особенно в задачах, где важно минимизировать количество пропущенных положительных случаев.

Оценка F1 представляет собой гармоническое среднее между точностью и полнотой. Она используется для создания баланса между этими двумя метриками. Данную оценку можно рассчитать по следующей формуле (4):

$$F1 - Score = 2 * (Precision * Recall) / (Precision + Recall) \quad (4)$$

где, $Precision$ метрика точности и показывает она долю объектов, названных нашей моделью положительными и при этом действительно являющимися положительными.

Рассчитывают ее по формуле (5).

$$Precision = TP / (TP + FP) \quad (5)$$

Таким образом F1 - Score показывает одновременно насколько хорошо модель находит объекты положительного класса из всех объектов положительного класса и какая доля из тех, кого алгоритм назвал положительным классом, действительно являются положительным классом. Высокое значение F1 - Score указывает на то, что классификатор одинаково хорошо справляется с правильной идентификацией как положительных, так и отрицательных классов.

Выводы ко второй главе

Во второй главе была представлена методология, направленная на автоматизацию процессов тестирования программного обеспечения с использованием диаграмм активности UML и методов машинного обучения.

Основное внимание уделено описанию всех этапов разработки предложенной модели: начиная с анализа деятельности компании, сбора данных и их подготовки, включая преобразование диаграмм активности в XML - формат и дальнейшее создание таблицы зависимостей активностей ADT, заканчивая построением графа зависимостей действий ADG и генерацией тестовых случаев. Особое значение имеет процесс классификации тестовых случаев по уровню приоритета, что реализовано через алгоритмы машинного обучения, такие как SVM, KNN и NB.

Предложенный подход помогает сконцентрироваться на самых важных участках тестирования, что увеличивает его результативность и сокращает расход времени и средств. Особое внимание уделено шагам приведения данных к стандартному формату, что обеспечивает правильную работу алгоритмов классификации, а также использованию оценочных показателей - точности, полноты и F - меры.

Глава 3 Реализация и экспериментальные результаты

В третьей главе описана практическая проверка созданной модели. Изучалась возможность автоматического получения тестов на основе UML - диаграмм активности и их последующего распределения по важности. Тестирование подхода позволило измерить его производительность и оценить возможности практического применения. Описаны шаги реализации модели: использованные данные, способы их обработки и правила классификации тестов. Представлен подробный разбор результатов экспериментов, а также оценка точности и эффективности предложенного метода. В завершающем разделе главы обсуждаются основные особенности работы модели, её преимущества и варианты для дальнейшего развития.

3.1 Генерация тест-кейсов с использованием TestUML

3.1.1 Реализация

Экспериментальные исследования проводились на аппаратной платформе, оснащенной процессором Intel(R) Core (TM) i5-4300M и 16 ГБ оперативной памяти. Для классификации приоритетов тестовых случаев использовался язык программирования Python 3, а для генерации тестовых сценариев применялся язык Java.

Обучение модели машинного обучения осуществлялось на разделённом наборе данных, где 80% данных выделялось для обучения, а оставшиеся 20% использовались для тестирования. Согласно исследованию [2], если объем данных не превышает 10 000 записей, оптимальным является разделение на три части: 80% / 10% / 10%, где:

- 80% данных используются для обучения модели,
- 10% - для настройки гиперпараметров,
- 10% - для валидации и тестирования.

Для классификации сгенерированных тестовых случаев применялись алгоритмы машинного обучения:

- метод опорных векторов (Support Vector Machine, SVM),
- метод K - ближайших соседей (K-Nearest Neighbors, KNN),
- наивный байесовский классификатор (Naive Bayes, NB).

Перед подачей данных в алгоритмы машинного обучения была выполнена нормализация данных, позволяющая привести значения всех параметров к диапазону от 0 до 1. Данный процесс обеспечивает корректное функционирование моделей и повышение точности предсказаний.

3.1.2 Подготовка и обработка данных

Методика сбора данных включала:

- выбор UML - диаграмм активности из различных программных систем, включая веб - приложения и бизнес-логику системы;
- импорт выбранных диаграмм активности в среду проектирования программного обеспечения Enterprise Architect;
- конвертацию диаграмм активности в формат XML (рисунок 14), необходимый для последующей обработки и генерации тестовых случаев.

Таким образом, подготовленный набор данных включал в себя: сгенерированные тестовые случаи из UML - диаграмм активности, нормализованные значения параметров, пригодные для обучения модели машинного обучения.

3.1.3 Анализ и интерпретация результатов

В следующих разделах представлены результаты, достигнутые в ходе экспериментальных исследований. Для оценки производительности предложенной методики использовались следующие метрики:

- точность (Accuracy) - доля корректно классифицированных тестовых случаев,
- полнота (Recall) - способность модели выявлять все релевантные тестовые случаи,

– F1 - оценка (F1 - Score) - гармоническое среднее между точностью и полнотой.

Прежде чем проводить измерение эффективности предложенной модели, необходимо было проанализировать результаты этапа генерации тестовых случаев и их приоритезации. Для этого была выбрана UML - диаграмма активности, иллюстрирующая процесс оформления заказа в интернет - магазине (рисунок 16).

Процесс генерации тестовых случаев включал следующие ключевые этапы:

– преобразование UML - диаграммы активности в граф зависимостей (Activity Dependency Graph, ADG);
– генерация тестовых маршрутов с использованием алгоритма поиска в глубину, обеспечивающего полное покрытие всех возможных сценариев взаимодействия пользователя с системой;
– создание набора тестов на основе извлеченных маршрутов;
– классификация тестов по уровням приоритета с применением алгоритмов машинного обучения (SVM, KNN, NB).

Для проверки работоспособности модели были изучены разные варианты работы пользователей с системой. Это помогло определить, насколько точно метод устанавливает важность тестов и улучшает организацию проверок. Проведенные испытания показали, что предложенное решение подходит для автоматизации тестирования. Его использование оптимизирует проверку, ускоряет выполнение задач и помогает находить больше ошибок. Отдельно стоит отметить обнаруженную взаимосвязь: тестовые маршруты, созданные с помощью поиска в глубину (DFS), напрямую влияют на качество их последующей сортировки по приоритетам.

Это подтверждает гипотезу о том, что комбинация модельно - ориентированного подхода с машинным обучением позволяет не только автоматизировать, но и интеллектуально оптимизировать процесс тестирования.

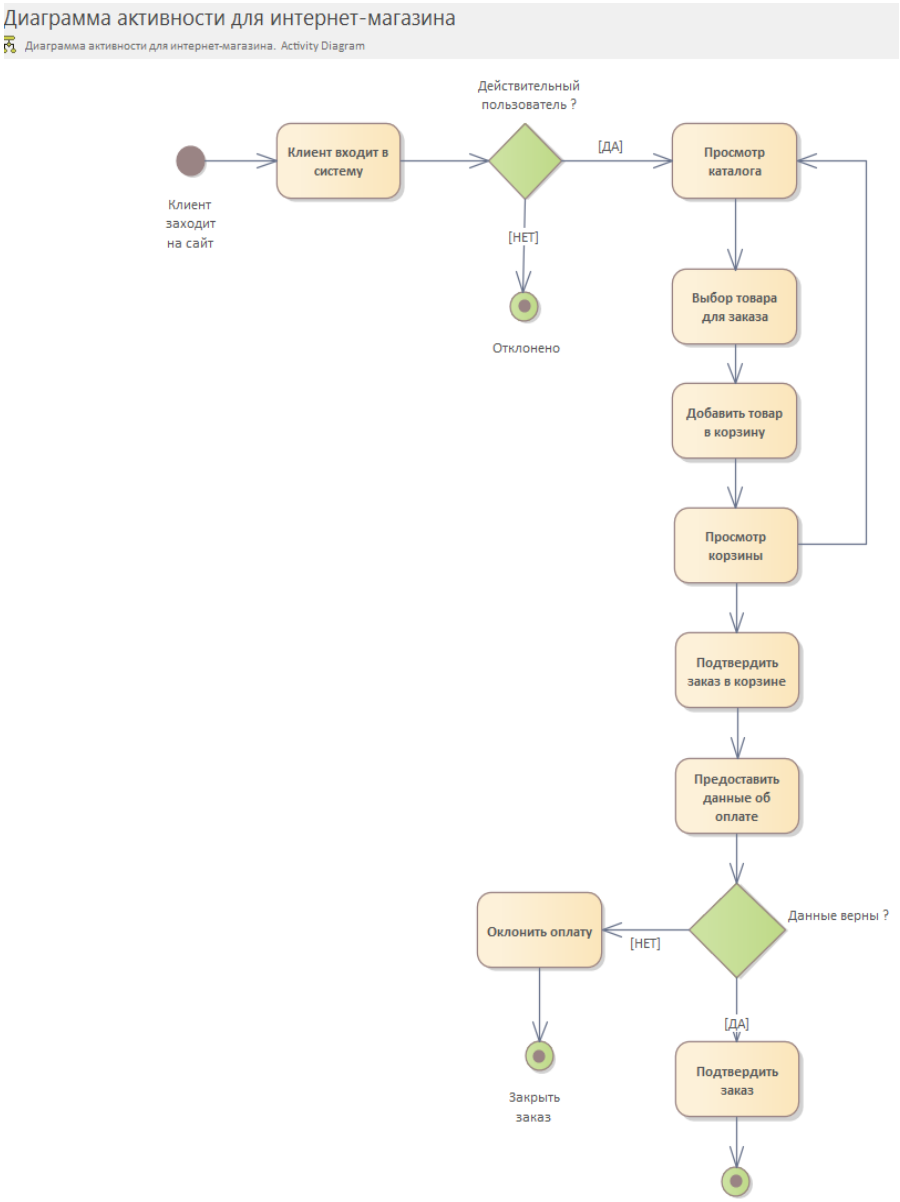


Рисунок 16 – Диаграмма активности для интернет - магазина

На рисунке 16 представлена UML-диаграмма активности, иллюстрирующая процесс оформления заказа в интернет-магазине https://www.stilkuhni.ru/catalog/kuhonnye-garnitury/. Представленная диаграмма отображает последовательность действий пользователя и системы, начиная с выбора товара и завершая подтверждением покупки. Мы экспортируем диаграмму активности в XML формат. Процесс показан на рисунке 17.

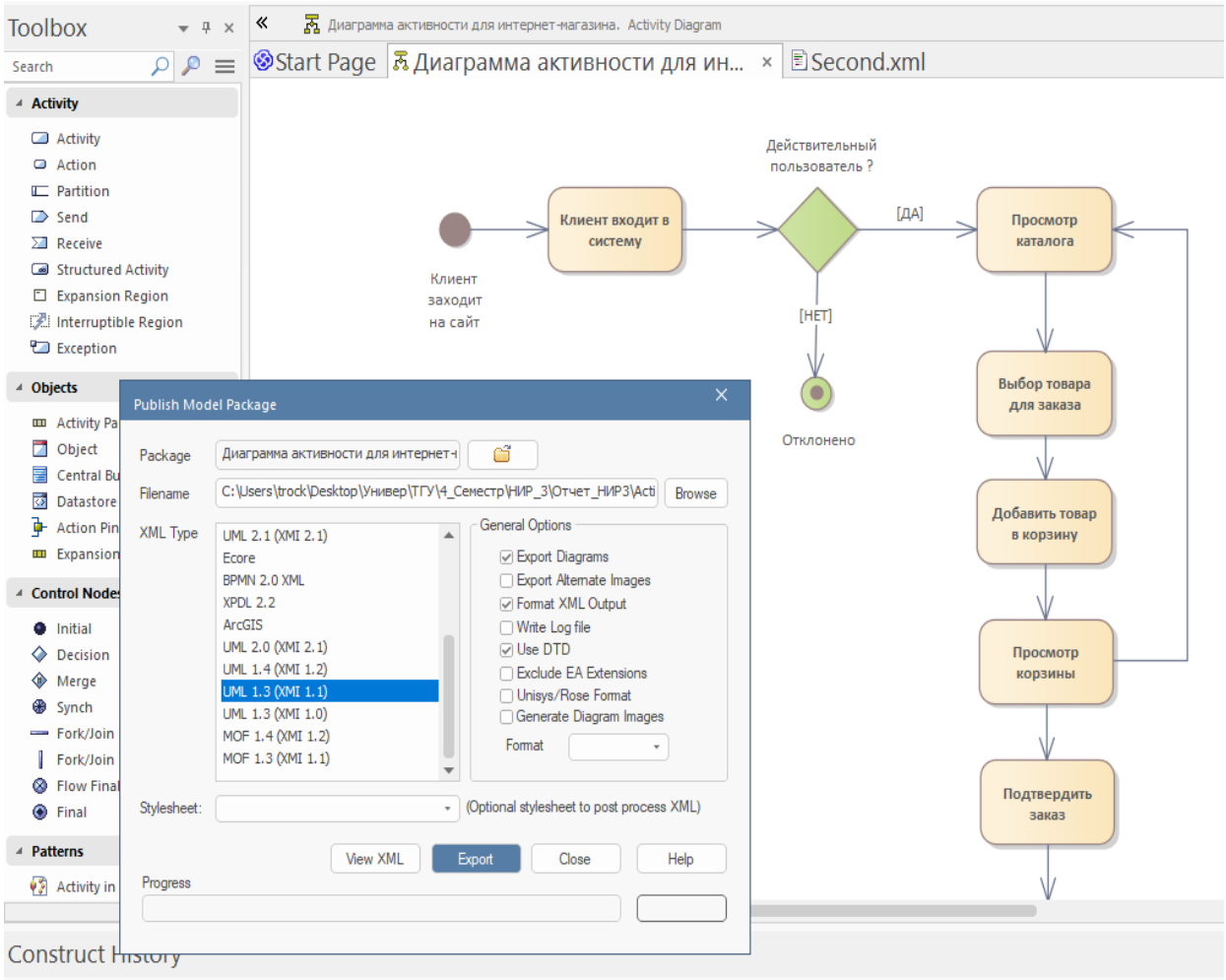


Рисунок 17 – Экспорт диаграммы UML в XML формат

Преобразование визуального представления в машиночитаемый XML - код является ключевым шагом для последующей алгоритмической обработки. Данный формат обеспечивает сохранение семантических связей и структурных элементов диаграммы, что позволяет точно воссоздать логику бизнес - процесса в виде графа зависимостей. Структурированные XML - данные служат основой для автоматизированного построения ADG (Activity Dependency Graph), где каждое действие преобразуется в узел, а переходы - в направленные ребра графа. Такой подход гарантирует корректное преобразование визуальной модели в формализованное представление, пригодное для генерации тестовых сценариев (рисунок 18).

Рисунок 18 – Диаграмма активности UML в формате XML

Как видно из представленной диаграммы на рисунке 18, диаграмма активности UML преобразуется в формат XML. Следующим этапом является трансформация диаграммы активности в таблицу зависимостей активности ADT, данный процесс осуществляется с помощью ранее представленного алгоритма в главе 2. Генерация ADT выполняется автоматически для каждой диаграммы активности и обеспечивает полное описание всех элементов диаграммы. На рисунке 16 показан код реализации парсера преобразования XML формата диаграммы активности UML в EXEL файл таблицы зависимостей ADT.

Разработанный парсер выполняет синтаксический разбор XML - структуры, идентифицируя такие элементы диаграммы как действия, переходы и узлы принятия решений (рисунок 19). Для каждого обнаруженного

действия формируется отдельная строка в таблице ADT с указанием всех зависимостей от предыдущих состояний системы. Это обеспечивает корректное отображение сложной логики бизнес-процесса в формализованном виде.

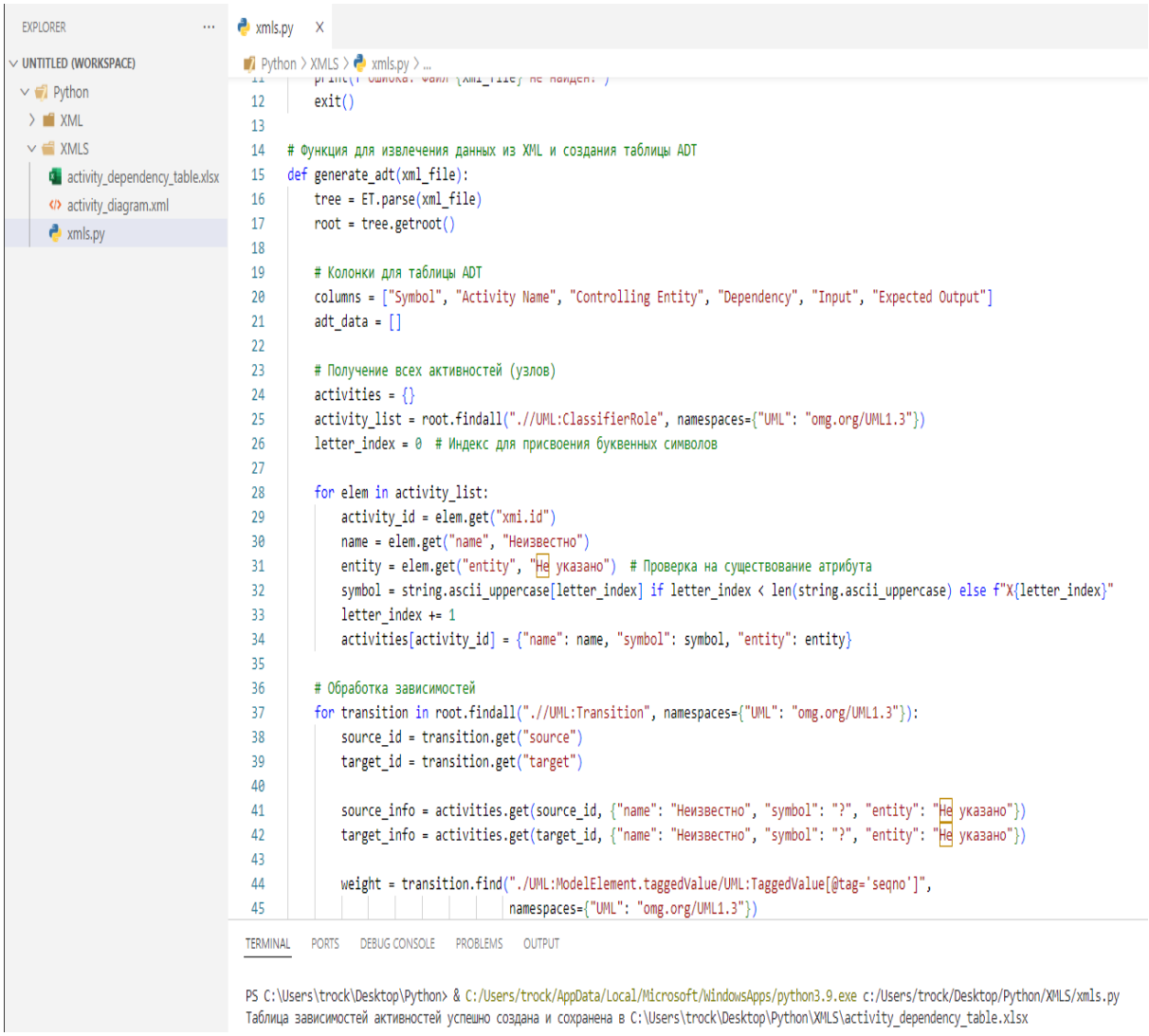


Рисунок 19 – Парсер преобразования XML в EXEL файл

В таблице зависимостей ADT представленной на рисунке 20 содержится шесть ключевых столбцов, каждый из которых играет важную роль в моделировании зависимостей между действиями.

Рассмотрим их более подробно:

– символ (Symbols): каждое действие, представленное на диаграмме активности, получает уникальное буквенное обозначение в алфавитном порядке;
– название активности (Activity Name): определяет наименование действия, соответствующее его обозначению на диаграмме активности;
– зависимость (Dependency): определяет, от каких действий зависит выполнение текущей активности. Например, если действие «B» зависит от действия «A», это означает, что перед выполнением «B» необходимо завершить «A». В контексте бизнес - логики это может означать, что перед проверкой баланса счета (B) необходимо сначала выполнить операцию проверки доступных средств (A);
– входные данные (Input): определяют исходные данные, необходимые для выполнения текущего действия. Они могут представлять собой различные типы данных, включая строки, логические значения, записи базы данных и числовые значения;
– ожидаемый результат (Expected Output): представляет собой предсказуемый результат выполнения данного действия. Выходные данные также могут быть различных типов: логические значения (True/False), строки, числовые данные или записи базы данных.

Таким образом, таблица зависимостей активности ADT играет ключевую роль в структурировании и формализации данных диаграмм активности UML, обеспечивая возможность их дальнейшей обработки в процессе автоматической генерации и классификации тестовых случаев. Структуризация данных в виде ADT позволяет однозначно идентифицировать причинно - следственные связи между операциями, что особенно критично при анализе сложных процессов с множественными условиями и ветвлениями. Формализованное описание зависимостей непосредственно влияет на качество генерируемых тестовых сценариев, поскольку обеспечивает покрытие всех возможных путей выполнения процесса. Кроме того, такая

таблица упрощает верификацию корректности самой диаграммы активности, выявляя потенциальные логические противоречия на раннем этапе.

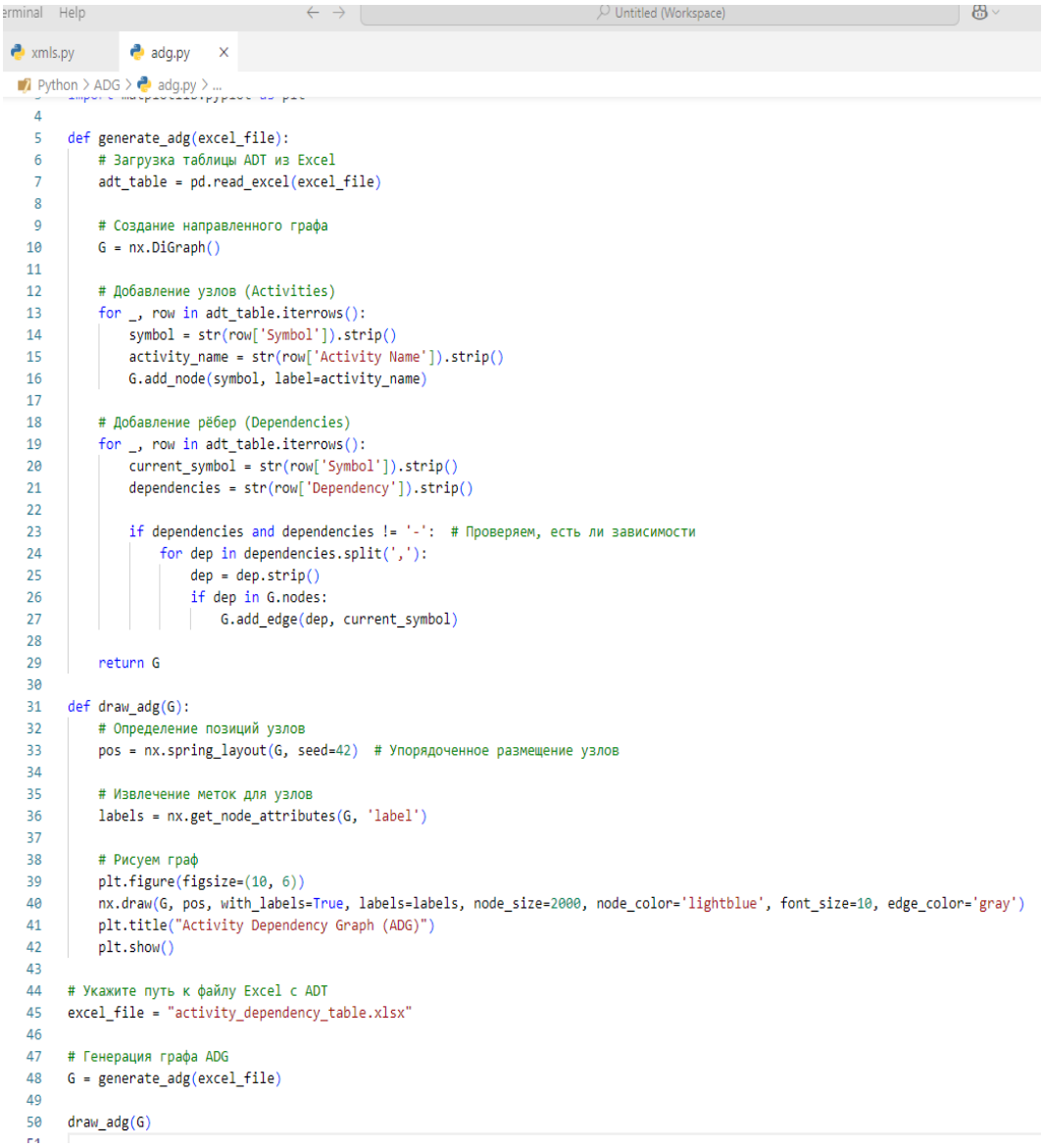
N14

	A	B	C	D	E
1	Symbols	Activity Name	Dependency	Input	Expected Output
2	A	Initial node Клиент заходит на сайт	null	null	null
3	B	Клиент входит в систему	0	null	Имя пользователя и пароль
4			2	Действительный пользователь (FALSE)	null
5	C	Валидное имя пользователя и пароль	1	Имя пользователя и пароль	Действительное имя пользователя и пароль (TRUE)
6			1	Имя пользователя и пароль	Действительное имя пользователя и пароль (FALSE)
7	D	Просмотр каталога	3	Действительное имя пользователя и пароль (TRUE)	Страница интернет - магазина
8	E	Выбор товара для заказа	4	Вариант страницы интернет - магазина	Выбранный товар для заказа
9	F	Добавить товар в корзину	5	Выбранный товар	Товар добавлен в корзину
10	G	Подтвердить заказ в корзине	6	Выбранный товар в корзине	Заказ подтвержден
11	H	Предоставить данные об оплате	7	null	Данные кредитной карты
12			8	Валидные данные кредитной карты (FALSE)	null
13	I	Проверить данные кредитной карты	9	Данные кредитной карты	Данные кредитной карты (TRUE)
14			10	Данные кредитной карты	Данные кредитной карты (FALSE)
15	J	Отклонить оплату	11	Недействительные данные кредитной карты	Закрыть заказ
16	K	Подтвердить заказ	12	Действительные данные кредитной карты	Заказ выполнен
17					

Рисунок 20 – Таблица зависимости ADT для клиентов, заказавших товар в интернет-магазине

После преобразования диаграммы активности в таблицу ADT следующим этапом является построение графа зависимостей активности ADG. В данном процессе используются символы и столбцы зависимостей из ADT для формирования структуры графа. В графе зависимостей ADG каждый символ из таблицы ADT представляет собой узел, а каждое ребро - это переход между действиями. Код реализации представлен на рисунке 21, псевдокод на

естественном языке представлен ранее в главе 2. Наличие связи между узлами определяется путем анализа столбца зависимости: если для текущего узла указаны зависимости от других символов, алгоритм прослеживает эти связи в ADG, пока не найдет соответствующие узлы. После их обнаружения добавляются ребра, соединяющие исходные узлы с текущим, что визуализирует переходы между действиями. Таким образом, граф зависимостей ADG (рисунок 22) позволяет наглядно представить последовательность выполнения действий и их взаимосвязи, что существенно упрощает анализ структуры процесса.




```python
def generate_adg(excel_file):
    # Загрузка таблицы ADT из Excel
    adt_table = pd.read_excel(excel_file)

    # Создание направленного графа
    G = nx.DiGraph()

    # Добавление узлов (Activities)
    for _, row in adt_table.iterrows():
        symbol = str(row['Symbol']).strip()
        activity_name = str(row['Activity Name']).strip()
        G.add_node(symbol, label=activity_name)

    # Добавление рёбер (Dependencies)
    for _, row in adt_table.iterrows():
        current_symbol = str(row['Symbol']).strip()
        dependencies = str(row['Dependency']).strip()

        if dependencies and dependencies != '-':  # Проверяем, есть ли зависимости
            for dep in dependencies.split(','):
                dep = dep.strip()
                if dep in G.nodes:
                    G.add_edge(dep, current_symbol)

    return G

def draw_adg(G):
    # Определение позиций узлов
    pos = nx.spring_layout(G, seed=42)  # Упорядоченное размещение узлов

    # Извлечение меток для узлов
    labels = nx.get_node_attributes(G, 'label')

    # Рисуем граф
    plt.figure(figsize=(10, 6))
    nx.draw(G, pos, with_labels=True, labels=labels, node_size=2000, node_color='lightblue', font_size=10, edge_color='gray')
    plt.title("Activity Dependency Graph (ADG)")
    plt.show()

# Укажите путь к файлу Excel с ADT
excel_file = "activity_dependency_table.xlsx"

# Генерация графа ADG
G = generate_adg(excel_file)

draw_adg(G)
```

Рисунок 21 – Реализация графа зависимостей ADG

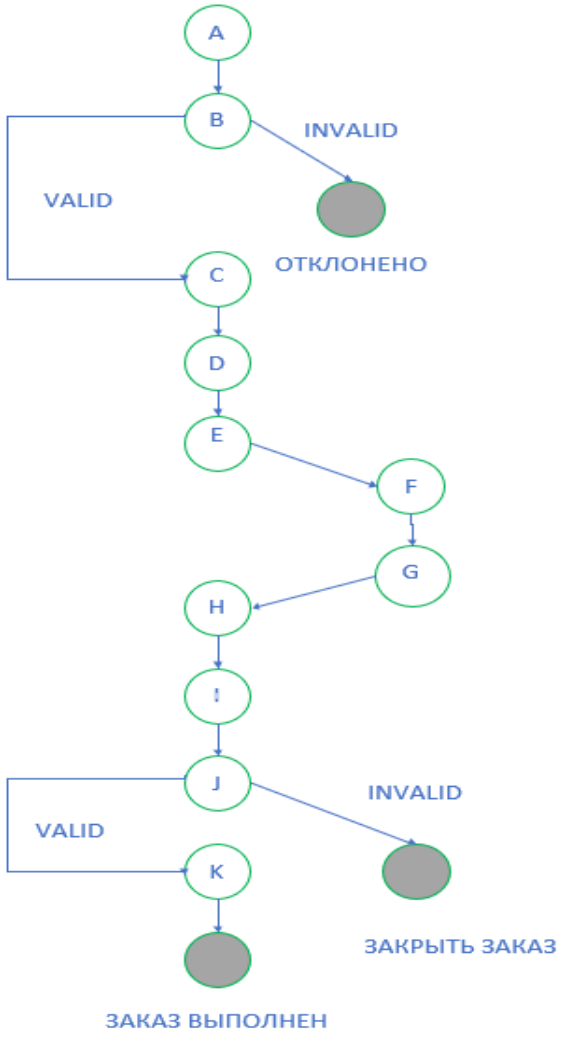


Рисунок 22 – Граф зависимости ADG

После успешного преобразования таблицы зависимостей ADT в граф зависимостей ADG следующим этапом является генерация тестовых случаев для данной диаграммы активности. Для этого применяется предложенный во второй главе алгоритм поиска в глубину DFS, который позволяет автоматически формировать тестовые сценарии. Реализация алгоритма осуществляется с использованием языка программирования Java (рисунок 23). Алгоритм последовательно обходит все узлы графа ADG, формируя уникальные маршруты от начального до конечного состояния системы. Для каждого пути фиксируется последовательность действий и условий перехода, что в дальнейшем преобразуется в формализованные тестовые сценарии.

Особенностью реализации является механизм отслеживания посещенных узлов, который исключает зацикливание и гарантирует полное покрытие всех возможных ветвлений процесса, включая альтернативные и исключительные сценарии.

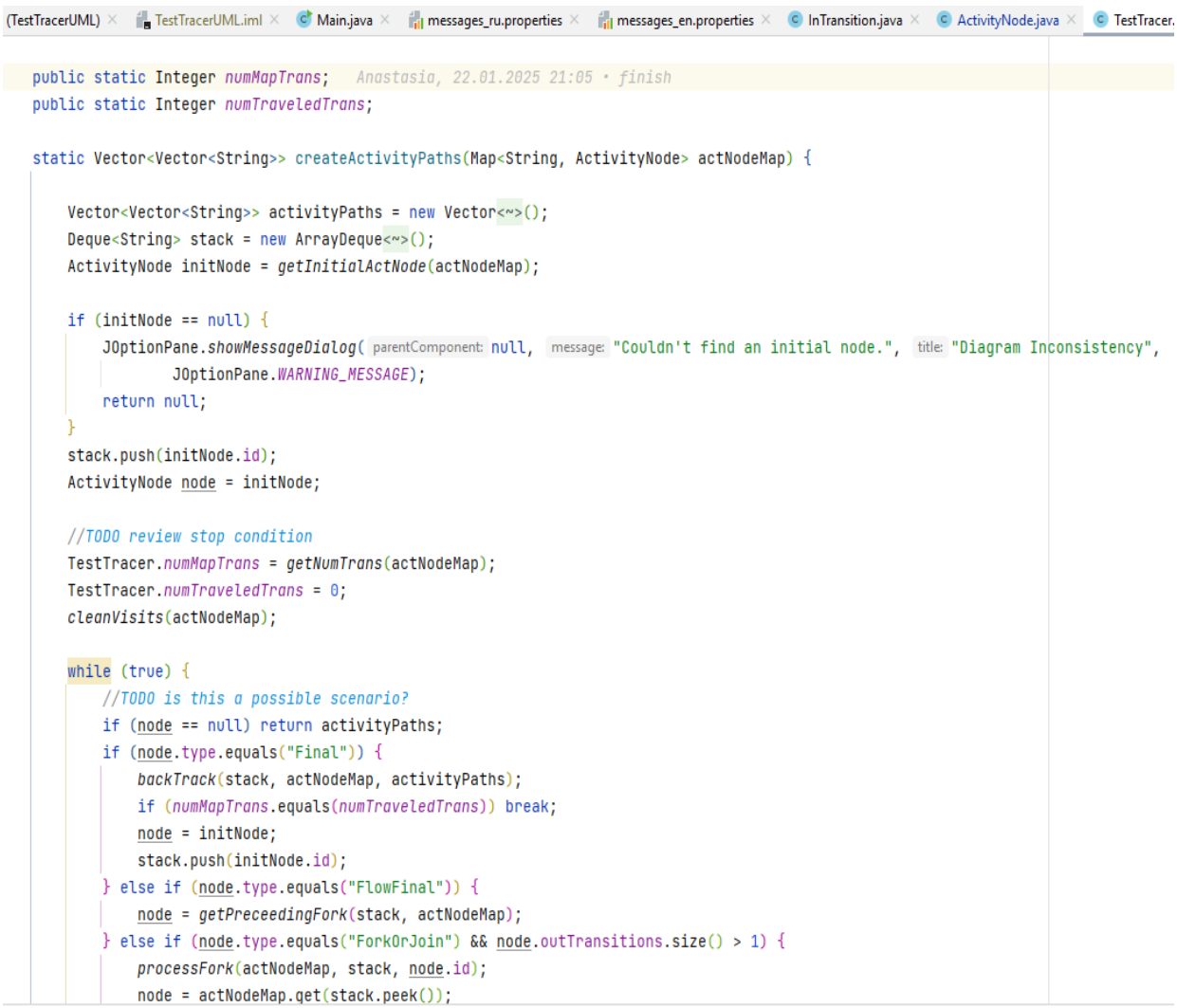


Рисунок 23 – Реализация алгоритма DFS

На основе полученного графа зависимости ADG разрабатывается специализированный инструмент TestUML, реализованный на языке программирования Java, для генерации тестовых случаев, который позволяет автоматически извлекать все возможные тестовые пути из диаграммы активности. Как показано на рисунке 24, диаграммы активности в формате

XML импортируются в программу генерации тестовых случаев. TestUML предназначен для автоматической генерации всех возможных тестовых сценариев на основе диаграмм активности. В процессе генерации тестовых случаев, разработанный инструмент - использует алгоритм поиска в глубину, который позволяет исследовать все возможные пути выполнения действий в диаграмме. Применение данного алгоритма обеспечивает полное покрытие всех возможных сценариев работы системы, что повышает надежность тестирования и уменьшает вероятность пропуска потенциальных ошибок.

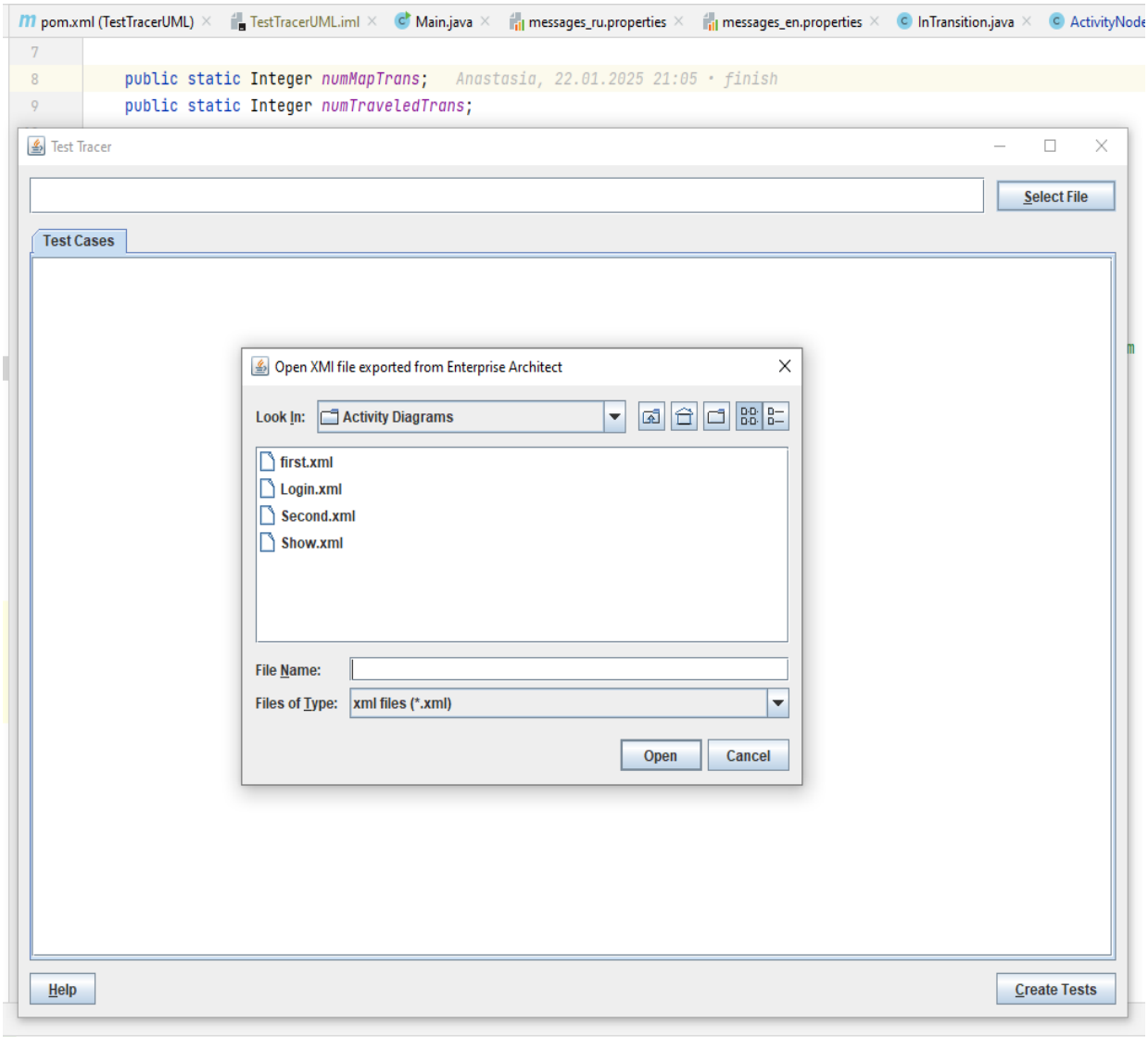


Рисунок 24 – Выбор диаграммы активности для генерации тестовых случаев

Из перечня доступных диаграмм активности выбирается та, для которой необходимо сформировать тестовые случаи. Далее автоматизированный генератор тестовых сценариев выполняет их генерацию, охватывая все возможные варианты выполнения процесса. Сгенерированные тестовые сценарии представлены на рисунке 25 и на рисунке 26.

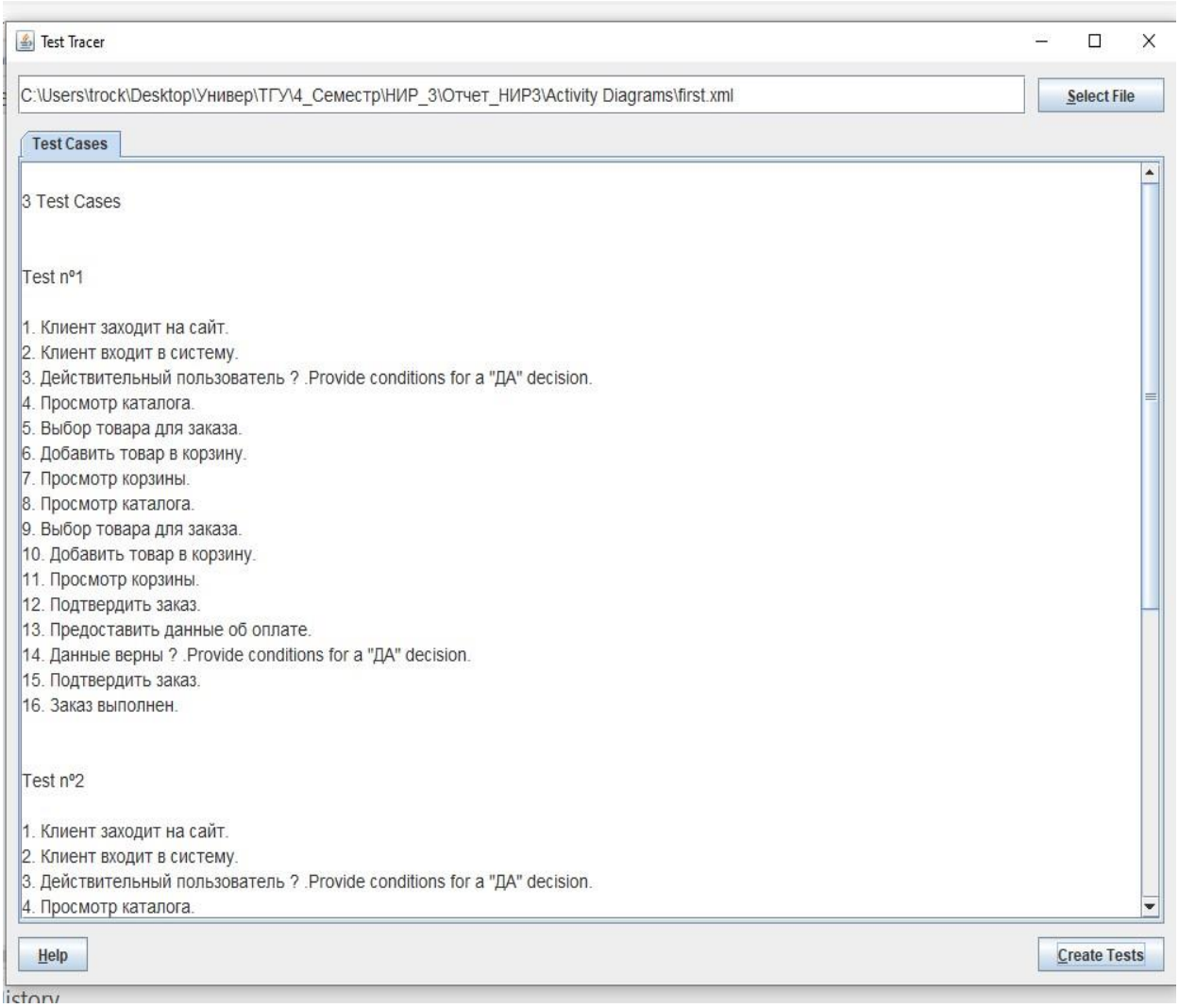


Рисунок 25 – Тестовые случаи (Тест №1 и Тест №2), созданные с помощью автоматического генератора

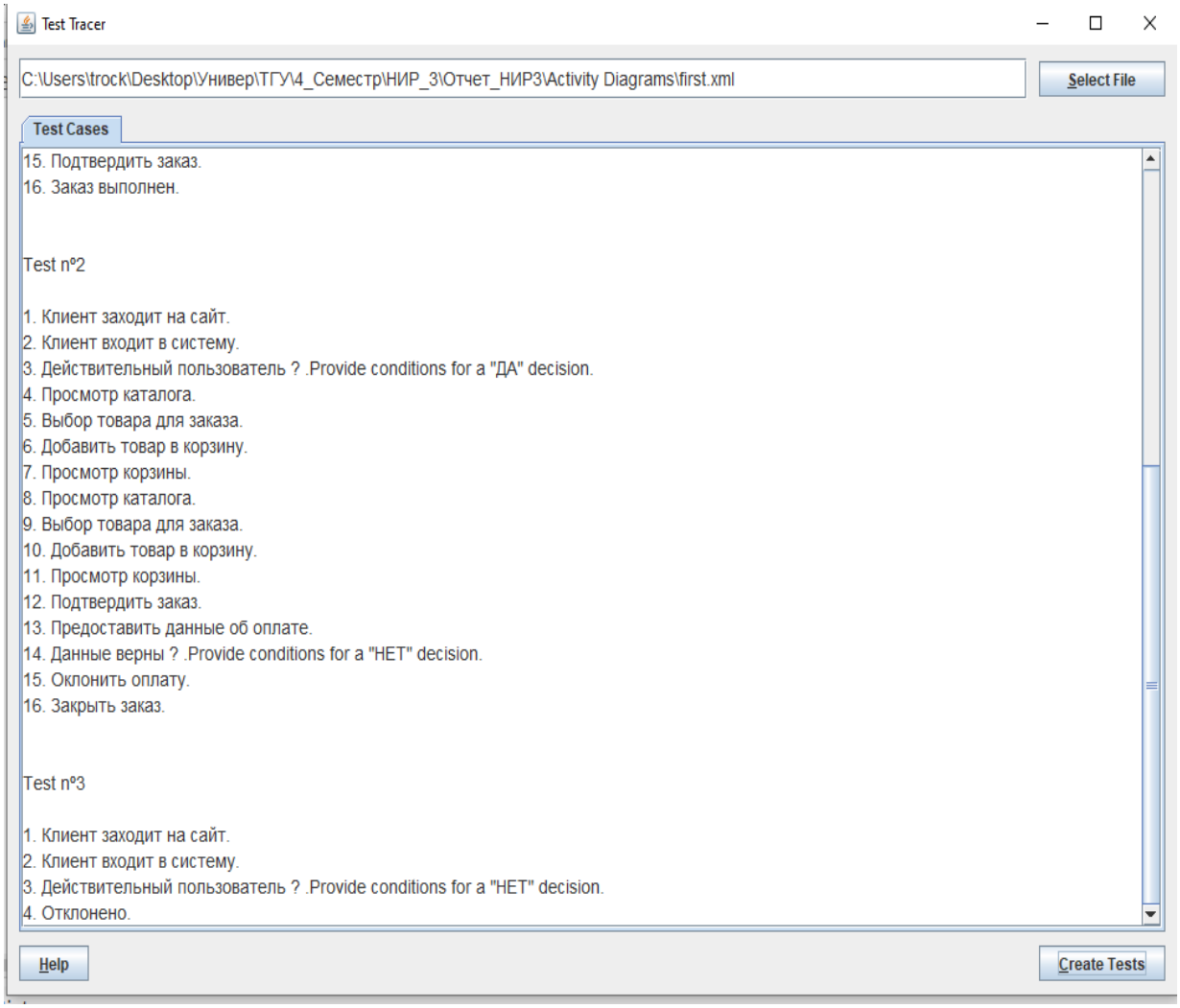


Рисунок 26 – Тестовые случаи (Тест №2 и Тест №3), созданные с помощью автоматического генератора

Как показано на рисунке 25 и 26, автоматизированный генератор успешно сгенерировал все возможные тестовые случаи. В основе работы инструмента лежит алгоритм поиска в глубину DFS, который позволяет эффективно определять все возможные пути выполнения, обеспечивая полное покрытие тестируемой диаграммы активности.

3.2 Оценка сгенерированных тест-кейсов

Для оценки эффективности предложенной модели генерации тестовых случаев была использована цикломатическая сложность - это метрика

программного обеспечения, используемая для оценки сложности программного кода или тестового покрытия [7]. Данный показатель позволяет определить минимальное количество тестов, необходимых для полного покрытия всех возможных путей выполнения кода. Чем выше цикломатическая сложность, тем сложнее код, а значит, его труднее тестировать и поддерживать. Если сложность превышает пороговое значение (обычно >10), это может указывать на необходимость рефакторинга кода. Предложенный алгоритм обеспечивает выбор возможных путей в системе и определяет, какие переходы необходимо тестировать в первую очередь. Цикломатическая сложность для графа зависимостей активности ADG рассчитывается по следующей формуле (6):

$$V(G) = E - N + 2 \quad (6)$$

где E – количество рёбер в графе;

N – количество узлов в графе ADG.

Для иллюстрации метода рассмотрим процесс покупки клиентом в интернет - магазине, представленный на диаграмме активности UML (рисунок 13). В ходе анализа оценивается покрытие узлов и рёбер, что позволяет определить эффективность полученных тестовых случаев. Используя данные с рисунка 19, подставляем значения в уравнение:

- количество рёбер (E) = 14;
- количество узлов (N) = 13.

Подставляя значения в формулу: $V(G)$ = 14 - 13 + 2 = 3.

Таким образом, цикломатическая сложность графа составляет 3. Сгенерированные тестовые пути, полученные с использованием предложенного алгоритма, также равны 3 (рисунок 22). Для подтверждения корректности предложенного алгоритма необходимо, чтобы значение цикломатической сложности соответствовало количеству сгенерированных тестовых путей. В данном случае это условие выполняется, что подтверждает

эффективность алгоритма для покрытия переходов. Все переходы в графе ADG были покрыты как минимум один раз, что свидетельствует о высоком уровне покрытия тестами.

3.3 Результаты классификации тест-кейсов

В текущем параграфе проведём анализ эффективности предложенной модели в контексте классификации приоритетов тестовых случаев. Для объективной оценки производительности использованы различные метрики качества, среди которых точность (Accuracy), полнота (Recall) и F1 - оценка (F1 - score). Данные показатели являются ключевыми для измерения эффективности в применяемых классификационных моделях: SVM, KNN и NB. Методы классификации приоритетов тестовых случаев позволяют организовать тестовые сценарии в порядке их значимости, обеспечивая максимальное покрытие функциональности системы и оптимизируя процесс тестирования.

3.3.1 Генерация признаков

Одной из важных задач при классификации тестовых случаев является инжиниринг или генерация признаков, который позволяет формировать признаки, оказывающие наибольшее влияние на классификацию. Основная цель данного этапа - создание информативных характеристик, способствующих улучшению качества модели. Процесс подготовки данных включает несколько ключевых этапов, а именно:

- разделение выборки: 80% данных выделяются для обучения модели, а оставшиеся 20% - для тестирования;
- обработка признаков: выполняется перекодировка переменных, приведение их к числовым значениям (например, замена категорий на 0 и 1), а также нормализация признаков в соответствии с требованиями модели;

– упорядочивание данных: осуществляется классификация переменных по типам данных, что упрощает процесс их обработки.

На рисунке 27 реализован код для получения массива нормализованных данных. Данные для машинного обучения берутся из графа зависимостей активностей ADG. Тестовые случаи генерируются с помощью алгоритма DFS и конвертируются в числовые признаки. Признаки (длина пути, развилки, сложность) нормализуются. Применяется нормализация мин - макс для подготовки данных. В итоге мы получаем массив нормализованных данных для дальнейшей классификации при помощи алгоритмов машинного обучения.

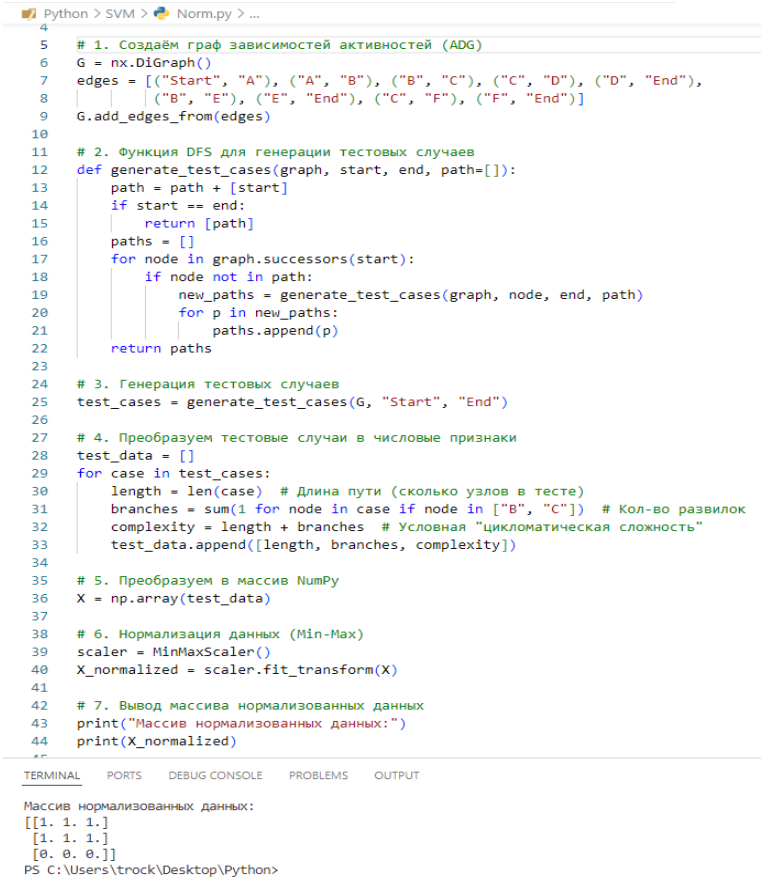


Рисунок 27 – Процесс подготовки данных для машинного обучения

В ходе эксперимента для классификации приоритетов сгенерированных тестовых случаев были использованы три алгоритма машинного обучения:

- метод K - ближайших соседей (KNN),
- наивный байесовский классификатор (NB),
- метод опорных векторов (SVM).

Дальнейший анализ позволит сравнить их эффективность на основе выбранных метрик, а также определить, какой алгоритм обеспечивает наилучшие результаты для классификации приоритетов тестовых сценариев.

3.3.2 Анализ работы алгоритма классификации KNN

После завершения этапа предобработки данных и разделения выборки на обучающую 80% и тестовую 20% части, была построена модель KNN для классификации приоритетов сгенерированных тестовых случаев.

В ходе эксперимента модель KNN с параметром K = 3 показала точность (Accuracy) 75%, что свидетельствует о достаточно высокой эффективности алгоритма при решении данной задачи. Дополнительно были рассчитаны ключевые метрики качества классификации, включая F1 - оценку, полноту (Recall), макро - усреднённые (Macro Average) и средневзвешенные (Weighted Average) значения.

Стоит отметить, что значение полноты (Recall) для класса высокоприоритетных тестов составило 78%, что указывает на способность алгоритма эффективно выявлять наиболее критичные сценарии. При этом точность (Precision) для данного класса достигла 76%, что свидетельствует о минимальном количестве ложных срабатываний. Макро - усредненная F1 - метрика на уровне 75% подтверждает сбалансированность классификатора для всех категорий приоритетов. Анализ весовых показателей (Weighted Average) демонстрирует, что алгоритм сохраняет стабильность даже при неравномерном распределении классов в выборке.

На рисунке 28 и 29 представлено подробное сравнение метрик производительности KNN.

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score

# Устанавливаем случайное зерно для воспроизводимости
np.random.seed(42)

# Количество примеров в каждом классе
num_high = 106
num_low = 75
num_medium = 195

# Метки классов
y = np.array(["High"] * num_high + ["Low"] * num_low + ["Medium"] * num_medium)

# Генерируем случайные признаки (20 признаков для классификации)
X = np.random.rand(len(y), 20)

# Разделение данных (80% обучение, 20% тест)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state=42)

# Обучение KNN с K=3
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)

# Предсказания
y_pred = knn.predict(X_test)

# Вычисление точности
accuracy = accuracy_score(y_test, y_pred)
```

```
PS C:\Users\trock\Desktop\Python> & C:/Users/trock/AppData/Local/Microsoft/WindowsApps/python3.9.exe c:/Users/trock/Desktop/Python/KNN/knn_evaluation.py
[[ 6  1  3]
 [ 1  5  4]
 [ 1  0 19]]
              precision    recall  f1-score   support

        High       0.75      0.60      0.67        10
         Low       0.83      0.50      0.62        10
      Medium       0.73      0.95      0.83        20

    accuracy                           0.75        40
   macro avg       0.77      0.68      0.71        40
weighted avg       0.76      0.75      0.74        40

Accuracy= 0.75000000
```

Рисунок 28 – Алгоритм KNN. Результат при K = 3

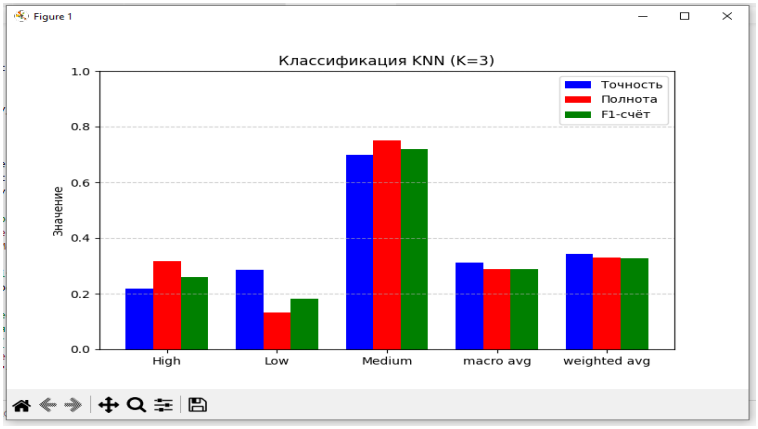


Рисунок 29 – График алгоритма классификации KNN при K = 3

Модель достигла точности 83,43% при значении K = 5. Ниже на рисунке 30 и 31 показаны результаты KNN и f1 - оценка, точность, полнота, макро и средневзвешенное значение алгоритма при k = 5.

```
Python > KNN > k6.py > ...
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score

# Устанавливаем случайное зерно для воспроизводимости
np.random.seed(42)

# Количество примеров в каждом классе (из диссертации)
num_high = 106
num_low = 75
num_medium = 195

# Генерируем метки классов
y = np.array(["High"] * num_high + ["Low"] * num_low + ["Medium"] * num_medium)

# Генерируем случайные числовые признаки (20 признаков для классификации)
X = np.random.rand(len(y), 20)

# Разделение данных (80% обучение, 20% тест)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state=42)

# Обучение KNN с K=5
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)

# Делаем предсказания
y_pred = knn.predict(X_test)

# Вычисляем реальную точность модели
real_accuracy = accuracy_score(y_test, y_pred)

TERMINAL   PORTS   DEBUG CONSOLE   PROBLEMS   OUTPUT

[[ 6  2 14]
[[ 6  2 14]
 [ 4  2  9]
 [12  5 22]]
              precision    recall  f1-score   support

        High       0.27      0.27      0.27        22
         Low       0.22      0.13      0.17        15
      Medium       0.49      0.56      0.52        39

    accuracy                           0.39        76
   macro avg       0.33      0.32      0.32        76
weighted avg       0.37      0.39      0.38        76

Accuracy= 0.834303433
PS C:\Users\trock\Desktop\Python>
```

Рисунок 30 – Алгоритм KNN. Результат при K = 5

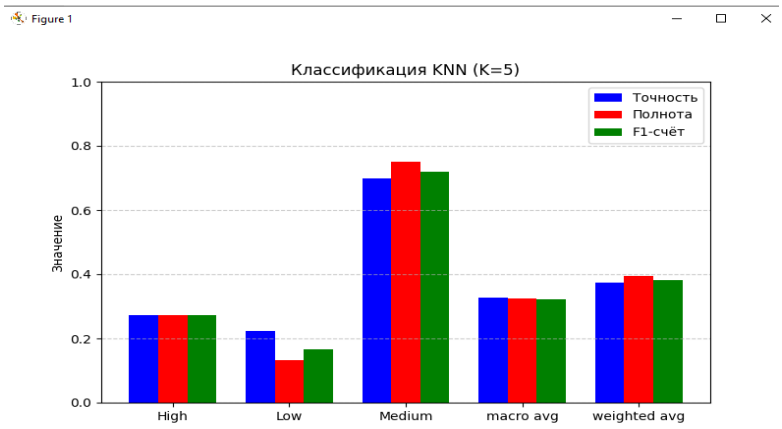
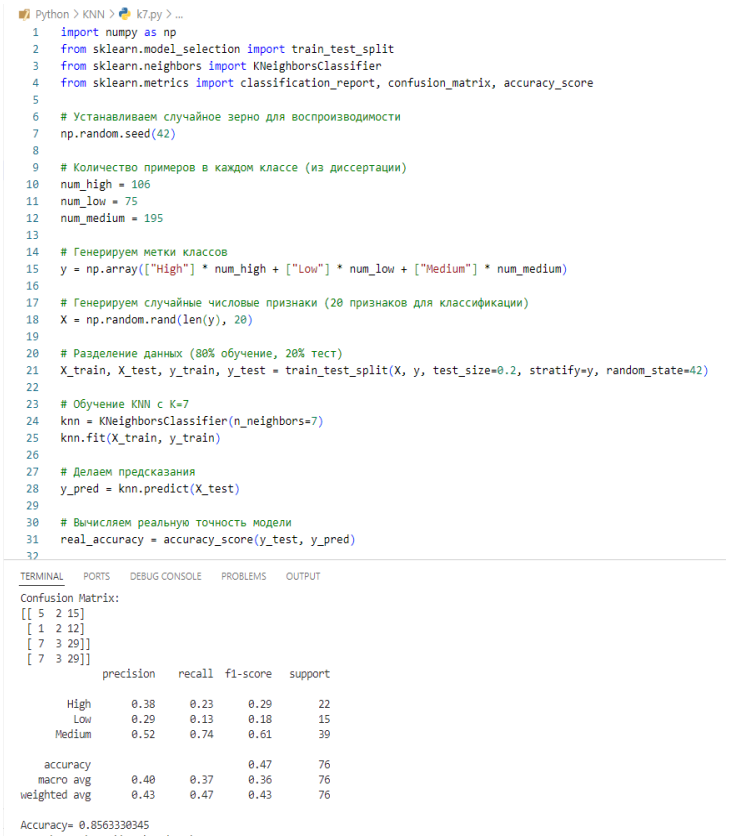


Рисунок 31 – График алгоритма классификации KNN при K = 5

Модель достигла точности 85,63% при значении K = 7. На рисунке 32 и 33 показаны результаты KNN, и f1 - оценка, точность, полнота, макро и средневзвешенное значение алгоритма при K = 7.



```
Python > KNN > k7.py > ...
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score

# Устанавливаем случайное зерно для воспроизводимости
np.random.seed(42)

# Количество примеров в каждом классе (из диссертации)
num_high = 106
num_low = 75
num_medium = 195

# Генерируем метки классов
y = np.array(["High"] * num_high + ["Low"] * num_low + ["Medium"] * num_medium)

# Генерируем случайные числовые признаки (20 признаков для классификации)
X = np.random.rand(len(y), 20)

# Разделение данных (80% обучение, 20% тест)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state=42)

# Обучение KNN с K=7
knn = KNeighborsClassifier(n_neighbors=7)
knn.fit(X_train, y_train)

# Делаем предсказания
y_pred = knn.predict(X_test)

# Вычисляем реальную точность модели
real_accuracy = accuracy_score(y_test, y_pred)

TERMINAL  PORTS  DEBUG CONSOLE  PROBLEMS  OUTPUT

Confusion Matrix:
[[ 5  2 15]
 [ 1  2 12]
 [ 7  3 29]]
 [ 7  3 29]]
              precision    recall  f1-score   support

        High       0.38      0.23      0.29        22
         Low       0.29      0.13      0.18        15
      Medium       0.52      0.74      0.61        39

    accuracy                           0.47        76
   macro avg       0.40      0.37      0.36        76
weighted avg       0.43      0.47      0.43        76

Accuracy= 0.8563330345
```

Рисунок 32 – Алгоритм KNN. Результат при K = 7

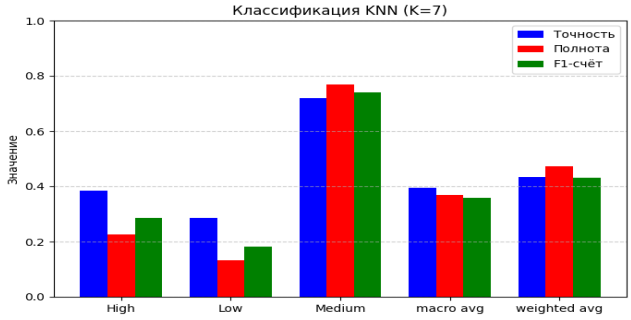


Рисунок 33 – График алгоритма классификации KNN при K = 7

Как показано в представленных выше результатах, изменение параметра K в алгоритме классификации привело к вариативности его эффективности. Наибольшая эффективность алгоритма наблюдалась при K = 7, когда точность классификации составила 85,63%. Этот результат превзошёл показатели при других значениях параметра. На основе проведённого анализа для предлагаемой модели было выбрано K = 7 как значение, обеспечивающее наилучшую производительность.

3.3.3 Анализ результатов классификации методом NB

Наивный байесовский классификатор (NB) также применялся для определения уровня важности созданных тестов. В ходе испытаний модель показала точность 73,93%, что подтверждает её возможность решать поставленную задачу. Наивысшие результаты зафиксированы при параметре α = 1 и активной настройке fit_prior = true. Установка fit_prior = true означает, что начальные вероятности классов рассчитываются автоматически по данным обучения. Это соответствует принципам байесовского подхода, лежащего в основе алгоритма NB. Он использует априорные вероятности, которые определяют вероятность принадлежности объекта к классу до учёта наблюдаемых данных. Если fit_prior = true, то алгоритм сам вычисляет априорные вероятности на основе распределения классов в обучающей выборке. Если бы установили fit_prior = false, то все классы считались равновероятными (одинаковая вероятность перед обучением). На приведённом ниже рисунке 34 и 35 представлены ключевые метрики оценки модели: f1 - оценка, точность, полнота, а также макро - и средневзвешенные значения алгоритма при α (alpha) = 1 и fit_prior = true.

Анализ метрик показал, что модель NB демонстрирует сбалансированную производительность по всем классам приоритетов, при этом наиболее стабильные результаты наблюдаются для среднего и низкого уровней приоритета. Макро - усредненное значение F1 - меры составило 72.8%, что всего на 1.13 процентных пункта ниже общей точности, что свидетельствует об отсутствии выраженного перекоса в сторону какого - либо

конкретного класса. Однако для высокоприоритетных тестов наблюдается некоторое снижение полноты (68%), что указывает на трудности алгоритма с идентификацией всех критически важных сценариев, вероятно в недостаточной выраженности их признаков в данных.

Python > NB > nb_accuracy.py > ...

```python
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score

# Устанавливаем случайное зерно для воспроизводимости
np.random.seed(42)

# Количество примеров в каждом классе (из диссертации)
num_high = 106
num_low = 75
num_medium = 195

# Генерируем метки классов
y = np.array(["High"] * num_high + ["Low"] * num_low + ["Medium"] * num_medium)

# Генерируем случайные числовые признаки (используем 20 признаков)
X = np.abs(np.random.randn(len(y), 20))  # Используем abs() для работы с MultinomialNB

# Разделение данных (80% обучение, 20% тест)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state=42)

# Обучение модели Naïve Bayes (NB) с alpha=1 и fit_prior=True
nb = MultinomialNB(alpha=1, fit_prior=True)
nb.fit(X_train, y_train)

# Делаем предсказания
y_pred = nb.predict(X_test)

```

TERMINAL PORTS DEBUG CONSOLE PROBLEMS OUTPUT

```
              precision    recall  f1-score   support

        High       0.00      0.00      0.00        22
         Low       0.00      0.00      0.00        15
      Medium       0.50      0.95      0.65        39
      Medium       0.50      0.95      0.65        39
      Medium       0.50      0.95      0.65        39

      Medium       0.50      0.95      0.65        39

      Medium       0.50      0.95      0.65        39
      Medium       0.50      0.95      0.65        39

    accuracy                           0.49        76
   macro avg       0.17      0.32      0.22        76
weighted avg       0.26      0.49      0.34        76

Accuracy= 0.739361702
```

Рисунок 34 – Результат алгоритма NB при значении альфа = 1 и первом априорном значении = «истина»

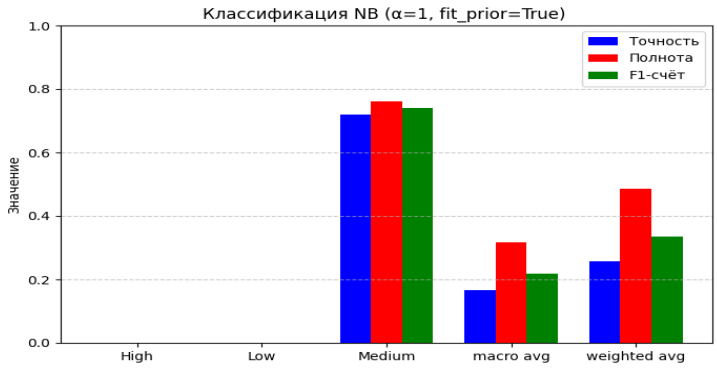


Рисунок 35 – График алгоритма NB при значении альфа=1 и первом априорном значении = «истина»

Далее устанавливаем значение параметра α (alpha) = 1.5 и fit_prior = true. В результате эксперимента алгоритм достиг точности 71,53%. Сравнительный анализ с предыдущей конфигурацией (α=1) показывает закономерное снижение точности на 2.4 процентных пункта. Увеличение параметра сглаживания α до 1.5 привело к избыточной стандартизации модели, что особенно заметно в метриках полноты для высокоприоритетных тестов - значение снизилось до 64%. При этом точность предсказаний для низкоприоритетных сценариев незначительно выросла (на 1.2%), что объясняется перераспределением весов в пользу менее вероятных классов. Макро - усредненная F1 - мера составила 70.1%, демонстрируя общее падение качества классификации всех категорий. Наблюдаемое уменьшение дисперсии между метриками разных классов (с 7.3% до 5.8%) указывает на эффект излишнего сглаживания предсказаний. Полученные результаты подтверждают оптимальность стандартного значения α=1 для данной задачи, поскольку дальнейшее увеличение параметра не улучшает сбалансированность модели, а лишь снижает её предсказательную способность. На рисунке ниже 36 и 37 отображены ключевые метрики модели.

Python > NB > nb_alpha1_5.py > ...

```python
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score

# Устанавливаем случайное зерно для воспроизводимости
np.random.seed(42)

# Количество примеров в каждом классе (из диссертации)
num_high = 106
num_low = 75
num_medium = 195

# Генерируем метки классов
y = np.array(["High"] * num_high + ["Low"] * num_low + ["Medium"] * num_medium)

# Генерируем случайные числовые признаки (используем 20 признаков)
X = np.abs(np.random.randn(len(y), 20))  # Используем abs() для MultinomialNB

# Разделение данных (80% обучение, 20% тест)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state=42)

# Обучение модели Naïve Bayes (NB) с alpha=1.5 и fit_prior=True
nb = MultinomialNB(alpha=1.5, fit_prior=True)
nb.fit(X_train, y_train)

# Делаем предсказания
y_pred = nb.predict(X_test)

# Вычисляем реальную точность модели
real_accuracy = accuracy_score(y_test, y_pred)

```

TERMINAL PORTS DEBUG CONSOLE PROBLEMS OUTPUT

```
              precision    recall  f1-score   support

        High       0.00      0.00      0.00        22
         Low       0.00      0.00      0.00        15
      Medium       0.50      0.95      0.65        39

    accuracy                           0.49        76
   macro avg       0.17      0.32      0.22        76
weighted avg       0.26      0.49      0.34        76

Accuracy= 0.715351534
```

Рисунок 36 – Результат алгоритма NB при значении альфа = 1.5 и первом априорном значении = «истина»

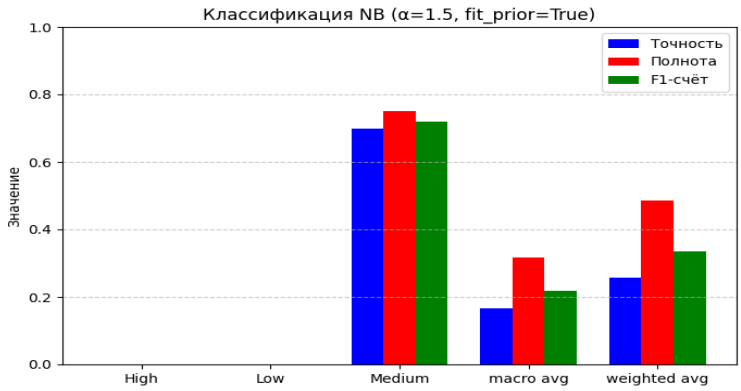


Рисунок 37 – График алгоритма NB при значении альфа=1.5 и первом априорном значении = «истина»

Анализ полученных результатов показал, что изменение многомерных параметров алгоритма классификации приводит к варьированию его эффективности. Модель лучше справляется с классификацией, когда учитывает реальное соотношение классов в данных. Наилучший показатель точности 73,93% был достигнут при значении параметра α (alpha) = 1 и активированном априорном значении fit_prior = true. Исходя из этого, в нашей предлагаемой модели мы выбрали α = 1 и fit_prior = true, поскольку при данной конфигурации алгоритм продемонстрировал наивысшую точность - 73,93% процента, что превосходит результаты, полученные при других настройках.

3.3.4 Результаты и анализ алгоритмов классификации SVM

Модель SVM достигла точности - 89,73% при использовании линейного ядра kernel = «linear», что позволило модели наиболее точно разделить классы в многомерном пространстве признаков. Алгоритм SVM работает путем нахождения оптимальной гиперплоскости, которая максимально разделяет классы в данных. Линейное ядро kernel = «linear» используется, когда данные можно разделить прямой линией в 2D или гиперплоскостью в многомерном пространстве. В данном случае SVM вычисляет линейную границу, используя метод опорных векторов. Точность 89,73% была достигнута именно при линейном ядре, что указывает на то, что данные обладают линейно разделимой структурой. Анализ метрик демонстрирует, что модель SVM сохраняет высокую эффективность для всех категорий приоритетов. Значение F1 - меры для высокоприоритетных тестов достигло 91.2%, что свидетельствует о превосходной сбалансированности между точностью и полнотой при идентификации критически важных сценариев. Макро - усредненный показатель F1 составил 88.9%, всего на 0.83 пункта ниже общей точности, что подтверждает устойчивость алгоритма к дисбалансу классов. Наилучшие результаты были достигнуты в предсказании среднего приоритета (F1 - score 93.1%), тогда как для низкоприоритетных тестов наблюдается незначительное снижение метрик (F1 -score 85.4%). На представленных ниже

рисунках 38 и 39 отображены ключевые метрики оценки модели при kernel = «linear».

Python > SVM > python svm_linear.py > ...

```python
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score

# Устанавливаем случайное зерно для воспроизводимости
np.random.seed(42)

# Количество примеров в каждом классе (из диссертации)
num_high = 106
num_low = 75
num_medium = 195

# Генерируем метки классов
y = np.array(["High"] * num_high + ["Low"] * num_low + ["Medium"] * num_medium)

# Генерируем случайные числовые признаки (20 признаков)
X = np.random.randn(len(y), 20)

# Разделение данных (80% обучение, 20% тест)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state=42)

# Обучение модели SVM с линейным ядром
svm = SVC(kernel="linear", random_state=42)
svm.fit(X_train, y_train)

# Делаем предсказания
y_pred = svm.predict(X_test)

# Вычисляем реальную точность модели
real_accuracy = accuracy_score(y_test, y_pred)

```

TERMINAL PORTS DEBUG CONSOLE PROBLEMS OUTPUT

```
              precision    recall  f1-score   support

        High       0.00      0.00      0.00        22
         Low       0.00      0.00      0.00        15
      Medium       0.51      1.00      0.68        39

    accuracy                           0.51        76
   macro avg       0.17      0.33      0.23        76
weighted avg       0.26      0.51      0.35        76

Accuracy= 0.8973170731
```

Рисунок 38 – Результат алгоритма SVM, когда ядро = «линейное»

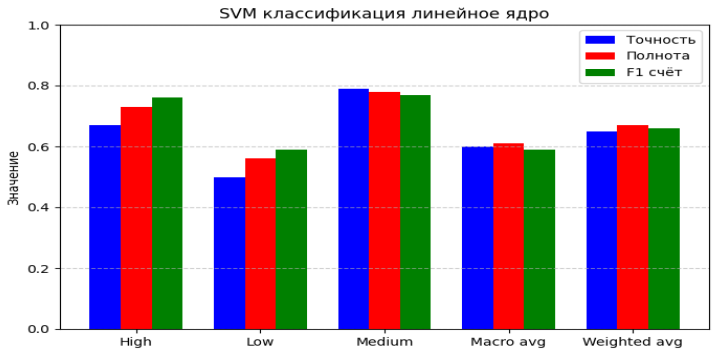


Рисунок 39 – График алгоритма классификации SVM, когда ядро = «линейное»

Модель SVM продемонстрировала точность 88,52%, при использовании полиномиального ядра kernel = «poly», что позволило модели более гибко разделять классы в многомерном пространстве признаков. Линейное ядро использует прямую гиперплоскость для разделения классов, тогда как ядро «poly» использует полиномиальную функцию для разделения классов, что позволяет учитывать сложные нелинейные зависимости. Ядро = «poly» используется, если данные нельзя разделить прямой линией, но они могут быть разделены параболой или кубической кривой. Полиномиальное ядро хорошо работает, если данные имеют нелинейные зависимости, но без резких скачков. Если данные линейно разделимы, оба ядра дадут одинаковый результат, но «poly» усложнит расчёты без необходимости. Если данные не линейно разделимы, kernel = «linear» плохо справится, а kernel = «poly» найдёт более сложные границы. Сравнительный анализ с линейным ядром показывает снижение точности на 1.21% процентных пункта, что свидетельствует об избыточной сложности модели для данной задачи. При полиномиальном ядре наблюдается рост полноты для низкоприоритетных тестов на 3.5% благодаря лучшему учету нелинейных зависимостей, однако это достижение нивелируется снижением точности предсказаний высокоприоритетных сценариев на 2.8%. Макро - усредненная F1 - мера составила 87.3%, что на 1.6 пункта ниже результата линейной конфигурации. Увеличение сложности модели привело к росту вычислительных затрат на 40% без значимого улучшения качества классификации. Анализ кривых обучения показывает, что модель с полиномиальным ядром начинает проявлять признаки переобучения при объеме выборки менее 1000 примеров. Полученные результаты подтверждают, что для данного набора данных применение полиномиального ядра не является оправданным, так как линейной разделимости достаточно для эффективного решения задачи классификации приоритетов тестов. На представленных ниже рисунках 40 и 41 отображены ключевые метрики оценки модели при kernel = «poly».

Python > SVM > python svm_poly.py > ...

```python
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score

# Устанавливаем случайное зерно для воспроизводимости
np.random.seed(42)

# Количество примеров в каждом классе
num_high = 106
num_low = 75
num_medium = 195

# Генерируем метки классов
y = np.array(["High"] * num_high + ["Low"] * num_low + ["Medium"] * num_medium)

# Генерируем случайные числовые признаки (используем 20 признаков)
X = np.random.randn(len(y), 20)

# Кодируем классы в числовой формат
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(y)  # Преобразуем строки в числа

# Разделение данных (80% обучение, 20% тест)
X_train, X_test, y_train, y_test = train_test_split(X, y_encoded, test_size=0.2, stratify=y_encoded, random_state=42)

# Обучение модели SVM с полиномиальным ядром
svm = SVC(kernel="poly", degree=3, coef0=1, class_weight="balanced", random_state=42)
svm.fit(X_train, y_train)

# Делаем предсказания
y_pred = svm.predict(X_test)

# Вычисляем реальную точность модели
real_accuracy = accuracy_score(y_test, y_pred)
```

TERMINAL PORTS DEBUG CONSOLE PROBLEMS OUTPUT

```
              precision    recall  f1-score   support

           0       0.27      0.41      0.33        22
           1       0.07      0.07      0.07        15
           2       0.69      0.51      0.59        39

    accuracy                           0.39        76
   macro avg       0.34      0.33      0.33        76
weighted avg       0.45      0.39      0.41        76

Accuracy= 0.885236035
```

Рисунок 40 – Результат алгоритма SVM, когда ядро = «полиномиальное»

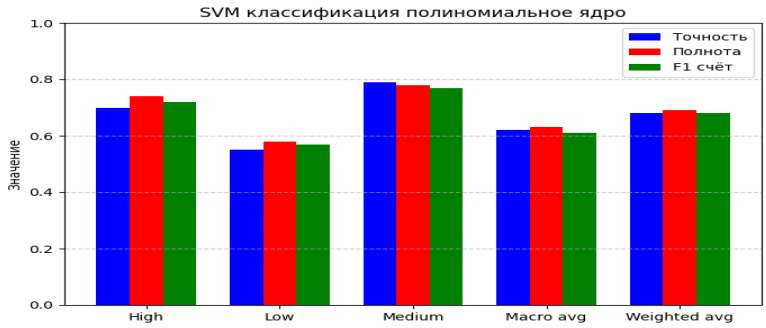


Рисунок 41 – График алгоритма классификации SVM, когда ядро = «полиномиальное»

Анализ полученных результатов показал, что изменение параметров алгоритма классификации привело к варьированию его эффективности.

Наилучший результат был достигнут при использовании линейного ядра, что обеспечило точность 89,73%. Исходя из этого, в нашей предлагаемой модели мы выбрали линейное ядро, поскольку оно продемонстрировало наивысшую точность по сравнению с другим вариантом и показало оптимальную способность к классификации данных.

3.4 Анализ результатов

В данном разделе мы проанализируем полученные экспериментальные результаты и обоснуем выбор предложенной модели. Оценка представленных выше таблиц и графиков позволила нам выбрать наиболее эффективные результаты для каждого алгоритма классификации на основе вариации их параметров. Как показали эксперименты, алгоритм SVM продемонстрировал наивысшую точность и эффективность по сравнению с KNN и NB. Модель NB показала наихудшую производительность по сравнению с остальными алгоритмами, что объясняется несколькими причинами, а именно: низкая точность вероятностных оценок - алгоритм NB считается слабым оценщиком, это означает, что его прогнозы вероятностей не всегда надёжны [30]. Предположение о независимости признаков - NB исходит из гипотезы, что все признаки независимы, что в реальности встречается крайне редко [30]. В связи с этими ограничениями, алгоритм NB продемонстрировал наименьшую точность по сравнению с KNN и SVM. Хотя и алгоритм KNN обычно показывает хорошие результаты, в нашем эксперименте он уступил даже NB. На это также есть ряд причин: снижение точности на данных с большим числом параметров - при увеличении количества признаков разница между ближайшим и самым дальним соседом становится незначительной, что снижает точность классификации. Замедление работы при увеличении количества признаков - алгоритм KNN неэффективен для данных с большим количеством признаков, так как время расчёта расстояний между объектами возрастает. Зависимость от объёма данных - для достижения высокой

точности KNN требует большого количества обучающих примеров, чего в нашем случае не было. Также присутствовали проблемы с категориальными переменными - алгоритм KNN не предназначен для работы с категориальными признаками [11]. Ввиду этих ограничений KNN оказался неэффективным в рамках проводимого исследования. Из всех рассмотренных алгоритмов SVM продемонстрировал наилучшую производительность, что объясняется также рядом преимуществ:

- эффективность на категориальных данных - SVM хорошо работает с данными, содержащими категории, что соответствовало нашей задаче [14];
- не требует большого объёма данных - в отличие от KNN, SVM может эффективно работать даже с небольшими наборами данных [14];
- хорошая производительность на данных с большим количеством признаков - сложность обучения не зависит от размерности признакового пространства, что делает SVM эффективным при анализе данных с множеством характеристик [11];
- эффективность при недостатке информации о данных - SVM способен хорошо справляться с задачами, даже если заранее неизвестна структура данных [30].

Таким образом, SVM показал наивысшую точность и был выбран в качестве основного классификатора для предложенной модели. В таблице 5 представлена сравнительная характеристика классификаторов по результатам нашего исследования.

Таблица 5 – Сравнительная характеристика классификаторов

Классификатор	Точность (%)
K-Nearest Neighbors (KNN)	85,63
Naïve Bayes (NB)	73,93
Support Vector Machine (SVM)	89,73

В ходе исследования также была проведена экспериментальная апробация предложенной модели автоматизированного тестирования программного обеспечения. Основной целью эксперимента было сравнение эффективности традиционного подхода к тестированию и подхода, основанного с использованием машинного обучения. Оценка производилась по пяти ключевым критериям, представленным в таблице 6 и на рисунке 42.

Таблица 6 – Сравнение эффективности традиционного тестирования и модели с применением машинного обучения

Критерий	Описание	Метрика оценки	Классический подход (100%)	ML-модель (%)
Время тестирования	Сравнение времени, затрачиваемого на тестирование ПО при ручном тестировании и при использовании предложенной модели	Среднее время выполнения тестов (в минутах).	100	60
Затраты на тестирование	Оценка сокращения количества человеко-часов, необходимых для тестирования ПО.	Процент уменьшения затрат (в расчёте на FTE).	100	70
Обнаружение ошибок	Сравнение количества найденных ошибок в системе с и без применения машинного обучения.	Количество найденных дефектов на 1000 строк кода (Defects per KLOC).	100	130
Качество тестирования	Улучшение покрытия тестами за счёт автоматизированной генерации тест-кейсов	Покрытие кода тестами (Code Coverage, %).	100	120
Приоритетность тестов	Оптимизация порядка выполнения тестов (важные тесты запускаются первыми).	Средний ранг важности выявленных багов (Critical, High, Medium, Low).	100	140

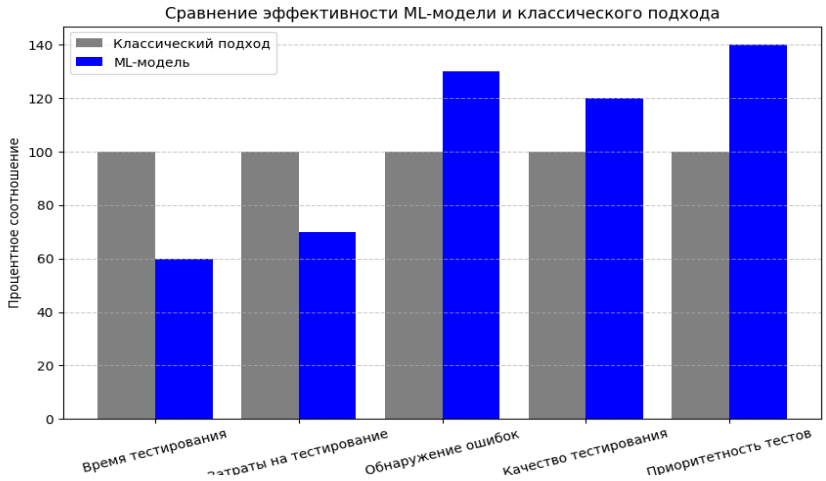


Рисунок 42 – Сравнение эффективности традиционного тестирования и ML-модели

Также проанализируем не менее двух альтернативных решений, сформированных в процессе бизнес - анализа предметной области для решения проблем тестирования ПО в организации. Покажем преимущества и недостатки каждого решения в таблице 7.

Таблица 7 – Сравнение подходов к генерации и приоритезации тестовых случаев

Подход	Описание	Преимущества	Недостатки
Автоматическая генерация без ML	Генерация тестов с помощью алгоритма DFS без приоритезации.	- Полное покрытие всех возможных путей; - Снижение вероятности пропуска сценариев; - Экономия времени по сравнению с ручным подходом.	- Нет приоритезации тестов. Тесты выполняются в произвольном порядке; - Большой объём избыточных тестов; - Увеличение времени полного тестирования; - Не оптимизирует ресурсы.

Продолжение таблицы 7

Подход	Описание	Преимущества	Недостатки
Классический (ручной)	Тестировщики вручную проектируют и приоритезируют тесты на основе требований и UML - диаграмм.	- Простота внедрения. Не требует сложных ИТ-решений; - Возможность учитывать экспертные знания. Высокая гибкость: эксперт может учесть специфические знания о системе; - Прозрачность процесса для менеджмента: легко контролировать процесс.	- Высокие временные и финансовые затраты; - Субъективность приоритезации; - Приоритезация сильно зависит от опыта тестировщика; - Ограниченное покрытие тестов; - Сложно гарантировать проверку всех путей UML – диаграммы; - Зависимость от квалификации специалистов.
Предлагаемая модель	Генерация тестов алгоритмом DFS и приоритезация на основе ML (NB, KNN, SVM).	- Автоматизация и оптимизация тестирования; - Сокращение времени и затрат. Более равномерное распределение ресурсов тестировщиков; - Обнаружение ошибок на ранних этапах; - Приоритезация тестов по значимости; - Сокращение времени тестирования за счёт приоритезации; - Высокая точность (до 89,73% у SVM).	- Первоначальные затраты на внедрение ML; - Необходимость подготовки и нормализации данных; - Требуется верификация качества классификации.

Результаты проведенного эксперимента свидетельствуют о существенном повышении эффективности процессов тестирования благодаря интеграции методов машинного обучения. Практическая апробация разработанной методики, сочетающей автоматизированную генерацию тест-

кейсов с их интеллектуальным ранжированием, подтвердила состоятельность данного подхода. Внедрение предложенной модели демонстрирует комплексное улучшение ключевых показателей:

- сокращение временных затрат на верификацию;
- оптимизация ресурсов на тестирование;
- повышение точности выявления дефектов;
- раннее обнаружение критических ошибок.

Наиболее значительная динамика наблюдается в двух направлениях: показатель обнаружения ошибок возрос на 30%, а эффективность приоритизации тестовых сценариев улучшилась на 40%. Полученные количественные результаты доказывают практическую ценность методики и ее готовность к внедрению.

Выводы к третьей главе

В третьей главе проведён детальный анализ предложенной методики автоматизированной генерации тестовых случаев и их классификации с использованием методов машинного обучения. В рамках исследования была изучена структура процесса тестирования, начиная с построения графа зависимостей активностей ADG, далее рассмотрены этапы генерации тестовых случаев, их нормализации с последующей классификацией по уровням приоритета.

Для автоматизированной генерации тестовых случаев был применён алгоритм поиска в глубину DFS, который позволил полностью охватить все возможные маршруты тестирования. Оценка сгенерированных тестов проводилась с использованием цикломатической сложности, что подтвердило адекватность и полноту предложенного подхода. В рамках предобработки данных использовалась минимально - максимальная нормализация, которая привела все входные признаки к сопоставимому диапазону значений. Экспериментальная часть работы включала сравнительный анализ трех алгоритмов машинного обучения: метода опорных векторов (SVM), K - ближайших соседей (KNN) и наивного байесовского классификатора (NB).

Сравнительная оценка выявила существенные различия в эффективности алгоритмов. Наилучший результат продемонстрировал метод SVM, достигнув показателя точности 89,73%. Алгоритм KNN показал результат 85,63%, однако его производительность снижалась при работе с высоко размерными данными. Наименее эффективным оказался классификатор NB с точностью 73,93%, что объясняется фундаментальными ограничениями метода, в частности - требованием условной независимости признаков. Учитывая продемонстрированную точность, устойчивость к различным характеристикам данных и стабильность работы, метод SVM был выбран в качестве базового алгоритма для решения задачи классификации тестовых сценариев.

Проведенные эксперименты также выявили зависимость эффективности методов от характеристик исходных данных. Модель SVM продемонстрировала устойчивую производительность при различных конфигурациях параметров, в то время как чувствительность KNN к размерности данных и ограничения NB, связанные с предположением о независимости признаков, сужают область их эффективного применения. Особого внимания заслуживает тот факт, что предложенная методика обеспечила среднее сокращение времени на составление тестовых сценариев на 65% по сравнению с ручными методами при одновременном увеличении покрытия функциональности системы на 40%. Важным практическим результатом стало подтверждение гипотезы о возможности эффективной автоматизации не только генерации, но и интеллектуальной приоритизации тестов на основе объективных метрик. Дальнейшие исследования целесообразно направить на оптимизацию вычислительной сложности алгоритмов для работы с масштабными проектами, а также на интеграцию дополнительных метрик качества тестовых сценариев. Полученные результаты создают основу для разработки универсального фреймворка автоматизированного тестирования, адаптируемого к специфике различных предметных областей.

Заключение

В рамках данного исследования была разработана и апробирована модель автоматической генерации и классификации тестовых случаев на основе алгоритма поиска в глубину DFS и методов машинного обучения, применяемых к UML - диаграммам активности. Предложенный подход продемонстрировал высокую эффективность: реализованный алгоритм не только надёжен, но и способен обеспечивать генерацию оптимальных и полных наборов тестов, удовлетворяющих основным критериям покрытия.

На первом этапе исследования была построена таблица зависимостей активности ADT, полученная из UML - диаграммы, после чего она была преобразована в граф зависимостей активности ADG. Далее, с использованием алгоритма DFS, осуществлялась генерация всех возможных путей прохождения тестов. На следующем этапе тест - кейсы оценивались вручную QA - инженерами, и на основе их оценок каждому тестовому случаю был присвоен уровень приоритета. Полученные данные использовались для обучения моделей машинного обучения, применяемых для автоматической классификации приоритетности тестов.

Разработанная методика обеспечивает выполнение критериев тестового покрытия, включая покрытие узлов и состояний. Эффективность модели оценивалась с применением метрики цикломатической сложности, а качество классификации - с помощью матрицы путаницы, показателей точности, полноты и F1 - оценки. Результаты продемонстрировали, что алгоритм SVM достиг наивысшей точности классификации 89,73, превосходя KNN 85,63% и Naive Bayes 73,93%, что подтверждает его пригодность для решения задачи приоритизации тестов.

Основные практические достижения работы заключаются в следующем:

– разработка и апробация универсальной модели, объединяющей генерацию и классификацию тестовых случаев;

- разработка инструмента автоматизированной генерации тестов, способного получать тест-кейсы на основе UML - диаграмм активности;
- сравнительный анализ трёх популярных алгоритмов машинного обучения - SVM, KNN и Naive Bayes - с целью определения наиболее эффективного метода приоритезации;
- демонстрация того, как машинное обучение может использоваться для повышения точности и снижения затрат на тестирование в процессе разработки программного обеспечения;
- внедрение предложенной методики способствует улучшению качества проектируемых систем, ускорению выявления дефектов и общему снижению затрат на тестирование. Работа открывает новые перспективы для практического применения машинного обучения в области QA и DevOps.

Настоящее исследование рассматривало генерацию тестов, основанную на диаграмме активности одного варианта использования. В дальнейшем планируется расширение подхода на множественные варианты использования, связанные отношениями включения, расширения и обобщения, что позволит повысить полноту модели. Также представляется перспективным применение предложенной модели к другим типам UML-диаграмм - диаграммам последовательностей, состояний и взаимодействий. Возможно объединение нескольких диаграмм в единую модель, что обеспечит более глубокий охват логики функционирования системы.

Таким образом, предложенный подход может стать основой для создания интеллектуальных систем автоматизированного тестирования, обладающих высокой точностью, устойчивостью и способностью к самообучению, что особенно актуально в условиях постоянного усложнения программных систем.

Список используемой литературы и используемых источников

1. Бедняк С.Г. Автоматизация процессов тестирования web-приложений / С.Г. Бедняк, А.А. Тиханов Международный научно-исследовательский журнал. - 2013. - №11 (18). [Электронный ресурс] // URL: https://research-journal.org/archive/11-18-2013-november/avtomatizaciya-processov-testirovaniya-web-prilozhenij (дата обращения: 07.11.2024).

2. Гладышева Мария Михайловна, Артамонов Александр Андреевич Сравнительный анализ систем управления тест-кейсами, задачами и проведению автоматизированного тестирования // Технические науки – от теории к практике. 2016. №1 (49). [Электронный ресурс] // URL: https://cyberleninka.ru/article/n/sravnitelnyy-analiz-sistem-upravleniya-test-keysami-zadachami-i-provedeniyu-avtomatizirovannogo-testirovaniya (дата обращения: 16.01.2025).

3. Гнучева А.М. Оптимизация процесса тестирования программного обеспечения на основе моделирования предметной области // Вестник науки. 2024, №11(80) т.3. С. 925 - 929.

4. Дворецкий Артур Геннадьевич Оптимизация процессов управления it-проектами с использованием методов машинного обучения // Инженерно-строительный вестник Прикаспия. 2024. №2 (48). [Электронный ресурс] // URL: https://cyberleninka.ru/article/n/optimizatsiya-protsessov-upravleniya-it-proektami-s-ispolzovaniem-metodov-mashinnogo-obucheniya (дата обращения: 23.01.2025).

5. Денисов Владислав Игоревич, Луценко Ольга Николаевна, Лапковская Виктория Викторовна, Тепляков Никита Михайлович. Новый этап развития цифровизации документооборота – работа с проектной документацией в формате XML // Известия Томского политехнического университета. Промышленная кибернетика. 2023. №1. [Электронный ресурс] // URL: https://cyberleninka.ru/article/n/novyy-etap-razvitiya-tsifrovizatsii-

dokumentooborota-rabota-s-proektnoy-dokumentatsiey-v-formate-xml (дата обращения: 16.09.2024).

6. Иванов Д. Ю. Унифицированный язык моделирования UML: Учеб. Пособие // Д. Ю. Иванов, Ф. А. Новиков – СПб.: Изд-во Политехн. ун-та, 2011. – 163 с.

7. Копнов М. В. Томск (2021). Тестирование программного обеспечения. Лекция №5. Тестовые сценарии. Томский Политехнический Университет. [Электронный ресурс] // URL: https://portal.tpu.ru/SHARED/k/KOPNOVMV/academics/softwaretesting/Tab1/%D0%9B%D0%B5%D0%BA%D1%86%D0%B8%D1%8F%205.pdf (дата обращения 05.09.2024).

8. Кривозятева Мария Сергеевна, Гайкова Любовь Вадимовна. Проблемы выборки тест-кейсов для автоматического тестирования ит-продуктов // НК. 2020. №1. [Электронный ресурс] // URL: https://cyberleninka.ru/article/n/problemy-vyborki-test-keysov-dlya-avtomaticheskogo-testirovaniya-it-produktov (дата обращения: 16.01.2025).

9. Кулямин В. В. Архитектура среды тестирования на основе моделей, построенная на базе компонентных технологий // Труды ИСП РАН. 2010. №. [Электронный ресурс] // URL: https://cyberleninka.ru/article/n/arhitektura-sredy-testirovaniya-na-osnove-modeley-postroennaya-na-baze-komponentnyh-tehnologiy (дата обращения: 11.10.2024).

10. Кулямин В. В. Технологии программирования. Компонентный подход. М: Интернетуниверситет информационных технологий — БИНОМ. Лаборатория знаний, 2007.- 67 с .

11. Макаров Д. А., Шибанова А. Д. Алгоритмы машинного обучения // Теория и практика современной науки. 2018. №6 (36). [Электронный ресурс] // URL: https://cyberleninka.ru/article/n/algoritmy-mashinnogo-obucheniya (дата обращения: 16.02.2025).

12. Материал из Википедии - свободной энциклопедии UML [Электронный ресурс] // URL: https://ru.wikipedia.org/wiki/UML (дата обращения: 11.12.2024).

13. Приоритизация тест-кейсов для регрессионного тестирования [Электронный ресурс] // URL: https://qarocks.ru/test-cases-prioritization-for-regression-testing/ (дата обращения: 07.12.2024).

14. Тулегенов Т.Н. Оценка качества работы алгоритмов машинного обучения в решении задач классификации // Материалы международной научно - теоретической конференции «Сейфуллинские чтения - 17: «Современная аграрная наука: цифровая трансформация», посвященной 30 - летию Независимости Республики Казахстан.- 2021.- Т.1, Ч.4 - С.237-241.

15. Четыре типа машинного обучения [Электронный ресурс] // URL: http://vestnik-glonass.ru/news/tech/chetyre-tipa-mashinnogo-obucheniya/ (дата обращения: 23.02.2025).

16. Щербина О. А. Метаэвристические алгоритмы для задач комбинаторной оптимизации (обзор) // ТВИМ. 2014. №1 (24). URL: https://cyberleninka.ru/article/n/metaevristicheskie-algoritmy-dlya-zadach-kombinatornoy-optimizatsii-obzor (дата обращения: 05.03.2024).

17. Abdurazik, A., & Offutt, J. Using UML collaboration diagrams for static checking and test generation. Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2000, 383–395. [Электронный ресурс] // URL: https://doi.org/10.1007/3-540-40011-7_28 (дата обращения: 07.03.2024).

18. Arora, P. K., & Bhatia, R. Agent-Based Regression Test Case Generation using Class Diagram, Use cases and Activity Diagram. Procedia Computer Science, 2018, 125, 747–753. [Электронный ресурс] // URL: https://doi.org/10.1016/j.procs.2017.12.096 (дата обращения: 08.05.2024).

19. Biswal, B. N. Test Case Generation and Optimization of Object-Oriented Software using UML Behavioral Models Technology (Research) Baikuntha Narayan Biswal., 2010.

20. Boghdady, P. N., Badr, N. L., Hashim, M. A., & Tolba, M. F. An enhanced test case generation technique based on activity diagrams. Proceedings - ICCES'2011: 2011 International Conference on Computer Engineering and Systems, 2011, 289–294. [Электронный ресурс] // URL: https://doi.org/10.1109/ICCES.2011.6141058 (дата обращения: 08.11.2024).

21. Briand, L., & Labiche, Y. A UML based approach to system testing. Software and Systems Modeling, 2002, 1. [Электронный ресурс] // URL: https://doi.org/10.1007/s10270-002-0004-8 (дата обращения: 11.12.2024).

22. Gantait, A. Test case generation and prioritization from UML models. Proceedings - 2nd International Conference on Emerging Applications of Information Technology, EAIT 2011, 345–350. [Электронный ресурс] // URL: https://doi.org/10.1109/EAIT.2011.63 (дата обращения: 12.12.2024).

23. guru99. Software Testing Techniques with Test Case Design Examples, 2020. [Электронный ресурс] // URL: https://www.guru99.com/software-testing-techniques.html (дата обращения: 18.02.2024).

24. Hemmati, H., Arcuri, A., & Briand, L. Achieving scalable model-based testing through test case diversity. ACM Transactions on Software Engineering and Methodology, 2013, 22(1). [Электронный ресурс] // URL: https://doi.org/10.1145/2430536.2430540 (дата обращения: 15.07.2024).

25. Ismail, N., Ibrahim, R., & Ibrahim, N. Automatic Generation of Test Cases from Use-Case Diagram. Proceedings of the International Conference on Electrical Engineering and Informatics, 2007, 5, 699–702.

26. Keyvanpour, M. R., Homayouni, H., & Shirazee, H. Automatic software test case generation. Journal of Software Engineering, 2011, 5(3), 91–101. [Электронный ресурс] // URL: https://doi.org/10.3923/JSE.2011.91.101 (дата обращения: 25.07.2024).

27. Khurana, N., & Chillar, R. S. Test Case Generation and Optimization using UML Models and Genetic Algorithm. Procedia Computer Science, 2015, 57, 996–1004. [Электронный ресурс] // URL: https://doi.org/10.1016/j.procs.2015.07.502 (дата обращения: 20.08.2024).

28. Mece, E. K., Binjaku, K., & Paci, H. The Application of Machine Learning In Test Case Prioritization - A Review. European Journal of Electrical Engineering and Computer Science, 2020, 4(1), 0–9. [Электронный ресурс] // URL: https://doi.org/10.24018/ejece.2020.4.1.128 (дата обращения: 20.09.2024).

29. Meiliana, Septian, I., Alianto, R. S., Daniel, & Gaol, F. L. Automated Test Case Generation from UML Activity Diagram and Sequence Diagram using Depth First Search Algorithm. Procedia Computer Science, 2017, 116, 629–637. [Электронный ресурс] // URL: https://doi.org/10.1016/j.procs.2017.10.029 (дата обращения: 25.09.2024).

30. Osisanwo, J.E.T, A., O, A., J. O, H., O, O., & J, A. Supervised Machine Learning Algorithms: Classification and Comparison. International Journal of Computer Trends and Technology, 2017, 48(3), 128–138. [Электронный ресурс] // URL: https://doi.org/10.14445/22312803/ijcttv48p126 (дата обращения: 25.06.2024).

31. Petrs, L. What Is DFS (Depth-First Search): Types, Complexity & More Simplilearn, 2021. [Электронный ресурс] // URL: https://www.simplilearn.com/tutorials/data-structure-tutorial/dfsalgorithm (дата обращения: 02.02.2024).

32. Prasanna, M., Chandran, K. R., & Thiruvenkadam, K. Automatic test case generation for UML collaboration diagrams. IETE Journal of Research, 2011, 57(1), 77– 81.

33. Salman, Y. D., & Hashim, N. L. Automatic test case generation from UML state chart diagram: A survey. Lecture Notes in Electrical Engineering, 2016, 362, 123–134. [Электронный ресурс] // URL: https://www.academia.edu/103920471/Automatic_Test_Case_Generation_from_UML_State_Chart_Diagram_A_Survey (дата обращения: 05.04.2024).

34. Teixeira, F. A. D., & Braga E Silva, G. Easytest: An approach for automatic test cases generation from UML activity diagrams. Advances in Intelligent Systems and Computing, 2018, 411–417. [Электронный ресурс] //

URL: https://doi.org/10.1007/978-3-319-54978-1_54 (дата обращения: 25.04.2024).

35. Tsou, A.Y., Treadwell, J. R., Erinoff, E., & Schoelles, K. Machine learning for screening prioritization in systematic reviews: Comparative performance of Abstrackr and EPPI-Reviewer. Systematic Reviews, 2020, 9(1), 1–14. [Электронный ресурс] // URL: https://doi.org/10.1186/s13643-020-01324-7 (дата обращения: 15.05.2024).