РЕФЕРАТ

Расчетно-пояснительная записка содержит 56 с., 16 рис., 2 табл., 29 ист., 1 прил.

Объектом работы является обнаружение дефектов программного обеспечения (ПО).

Ключевые слова: дефект, машинное обучение, градиентный бустинг, метрики кода, обнаружение, методы.

Целью работы является разработка и программная реализация метода обнаружения дефектов ПО с использованием алгоритмов машинного обучения.

В аналитическом разделе представлен обзор и сравнительный анализ существующих методов поиска дефектов ПО, описана формализованная постановка задачи обнаружения дефектов с использованием алгоритмов машинного обучения в виде IDEF0-диаграммы. В конструкторском разделе разработан метод обнаружения дефектов ПО с применением алгоритма градиентного бустинга. В технологическом разделе создан программный продукт, реализующий разработанный метод. В исследовательском разделе проведено исследование эффективности метода обнаружения дефектов ПО.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Одной из актуальных проблем разработки и внедрения программного обеспечения (ПО) является наличие дефектов, то есть ошибок в коде программы, приводящих к снижению качества продукции. Причинами появления дефектов могут стать некачественная организация процесса разработки ПО, недостаточная квалификация и опыт разработчиков, недостаточность ресурсов на разработку. Затраты на выявление и устранение дефектов могут составлять до 80% от общей стоимости ПО [1]. При этом чем раньше будет обнаружен дефект, тем меньше ущерба будет нанесено разработчику и эксплуатанту ПО.

Существует множество техник и технологий для определения дефектов: начиная от ручных средств, заканчивая автоматизированным тестированием ПО. Однако, чем больше объем исходного кода, тем больше трудозатрат необходимо для поддержания качества программной системы, при этом ресурсов может не хватать. В данном случае решением могут стать методы машинного обучения, которые на основе ранее написанного кода позволят дать вероятностную оценку нахождения дефекта в том или ином месте программы.

Цель данной работы: разработка и программная реализация метода обнаружения дефектов ПО с использованием алгоритмов машинного обучения.

Для достижения цели необходимо решить следующие **задачи**:

— проанализировать и сравнить существующие методы машинного обучения для обнаружения дефектов ПО;
— разработать метод обнаружения дефектов ПО с применением ансамбля деревьев решений (градиентного бустинга);
— разработать программное обеспечение, реализующее метод обнаружения дефектов ПО;
— провести исследование эффективности разработанного метода и сравнение его с существующими реализациями.

1 Аналитический раздел

1.1 Дефекты разрабатываемого ПО

1.1.1 Понятие дефекта ПО

Согласно стандартному глоссарию терминов и определений, используемых в тестировании ПО, **дефект** – это изъян в компоненте или системе, который может привести компонент или систему к невозможности выполнить требуемую функцию [2]. Другими словами, дефект – это отклонение от первоначальных бизнес-требований, логическая ошибка в исходном коде программы. Дефект не оказывает влияния на функционирование ПО до тех пор, пока он не будет обнаружен при эксплуатации программы. Это может привести к тому, что продукт не будет удовлетворять потребностям пользователя, а также к отказам компонента или системы. Последствия программной ошибки для пользователя могут быть серьезны. Например, дефект может поставить под угрозу бизнес, репутацию, государственную безопасность, безопасность пользователей, окружающую среду [3].

Выделяют **три вида** дефектов: программные, технические, архитектурные.

1. Программные ошибки – возникают из-за несовершенства исходного кода конечного продукта.
2. Технические дефекты – сводятся к доступу тех или иных функций готового решения или его дизайна.
3. Архитектурные ошибки – это ошибки, вызванные внешними факторами, которые заранее не были учтены при проектировании решения, вследствие чего приложение демонстрирует результат работы, отличный от ожидаемого [4].

Также с точки зрения степени **влияния на работоспособность** ПО можно выделить пять видов.

1. Блокирующий дефект (англ. blocker) – ошибка, которая приводит программу в нерабочее состояние.

2. Критический дефект (англ. critical), приводящий некоторый ключевой функционал в нерабочее состояние, существенное отклонение от бизнес-логики, неправильная реализация требуемых функций и т.д.
3. Серьезная ошибка (англ. major), свидетельствующая об отклонении от бизнес-логики или нарушающая работу программы (не имеет критического воздействия на приложение).
4. Незначительный дефект (англ. minor), не нарушающий функционал тестируемого приложения, но который является несоответствием ожидаемому результату.
5. Тривиальный дефект (англ. trivial), не имеющий влияние на функционал или работу программы, но который может быть обнаружен визуально.

Помимо этого, **по приоритетности исправления** дефекты могут быть с высоким, средним и низким приоритетом.

1. С высоким приоритетом (англ. high) – должен быть исправлен как можно быстрее, т.к. критически влияет на работоспособность программы.
2. Со средним приоритетом (англ. medium) – дефект должен быть обязательно исправлен, но он не оказывает критическое воздействие на работу.
3. С низким приоритетом (англ. low) – ошибка должна быть исправлена, но не имеет критического влияния на программу и устранение может быть отложено [5].

На рисунке 1.1 представлена классификация дефектов на основе приведенного описания.

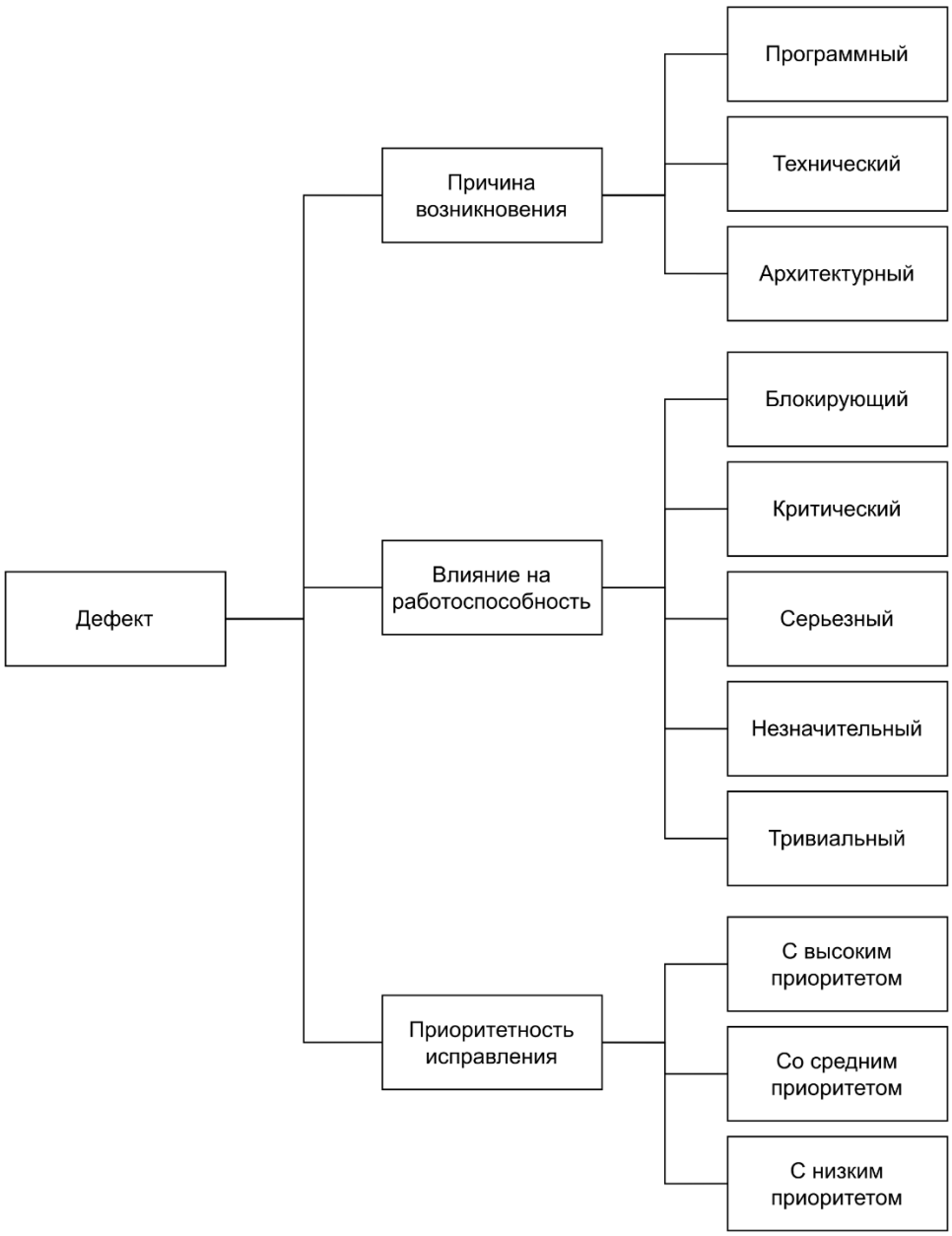

Рисунок 1.1 – Классификация дефектов ПО

Основными **причинами** возникновения дефектов являются:

— сложность реализации задачи;

— сжатые сроки разработки;

— несовершенство документации;

— изменение требований;

— недостаточная квалификация и опыт разработчиков;

— неправильная организация процесса разработки [4].

1.1.2 Классификация методов для обнаружения дефектов ПО

Дефекты в программе могут быть обнаружены не сразу и при этом иметь отрицательное влияние на процесс ее использования. При позднем обнаружении дефектов снижаются качество и надежность ПО, увеличиваются затраты на его переработку, проявляются негативные последствия. Своевременное выявление и исправление дефектов играют большую роль в жизненном цикле ПО, помогает улучшить качество разрабатываемых систем [6].

Для выявления дефектов ПО используются различные методы, которые делятся на **три категории**.

1. Статические методы – методы, при которых ПО тестируется без какого-либо выполнения программы (системы). При этом программные продукты проверяются вручную или с помощью различных средств автоматизации.
2. Динамические методы – в данных методах ПО тестируется путем выполнения программы. С помощью этого метода можно обнаружить различные типы дефектов:
   – функциональные – возникают, когда функции системы работают не в соответствии со спецификацией. Данные дефекты находятся благодаря функциональному тестированию, которое подразделяется на:
      1) модульное – используется для тестирования отдельно взятого модуля программы (помогает разработчикам узнать, правильно ли работает каждый блок кода в изоляции от остальных);
      2) интеграционное – позволяет убедиться, что при взаимодействии интегрированные блоки работают без ошибок;
      3) системное – программное обеспечение тестируется как единое целое, проверяется функциональность, безопасность и переносимость;
      4) регрессионное – проверка ранее протестированной программы, позволяющая убедиться, что внесенные изменения не повлекли за

собой появления дефектов в той части программы, которая не менялась;

5) приемочное – комплексное тестирование, необходимое для определения уровня готовности системы к последующей эксплуатации;
6) дымовое (англ. smoke) – проверка программного обеспечения на стабильность и наличие явных ошибок [7];

– нефункциональные – соответственно затрагивают нефункциональные аспекты приложения, могут повлиять на производительность, удобство использования и т.д.

3. Эксплуатационные методы – методы, при которых дефект обнаруживается пользователями, клиентами или контролирующим персоналом, то есть в результате сбоя.

На рисунке 1.2 представлена классификация рассматриваемых методов. Все методы важны и необходимы в процессе управления дефектам. При их объединении достигается наиболее качественный результат, за счет чего повышается производительность ПО. На ранней стадии наиболее эффективными методами являются статические. Они позволяют снизить затраты, необходимые для исправления дефектов, и сводят к минимуму их влияние [6].

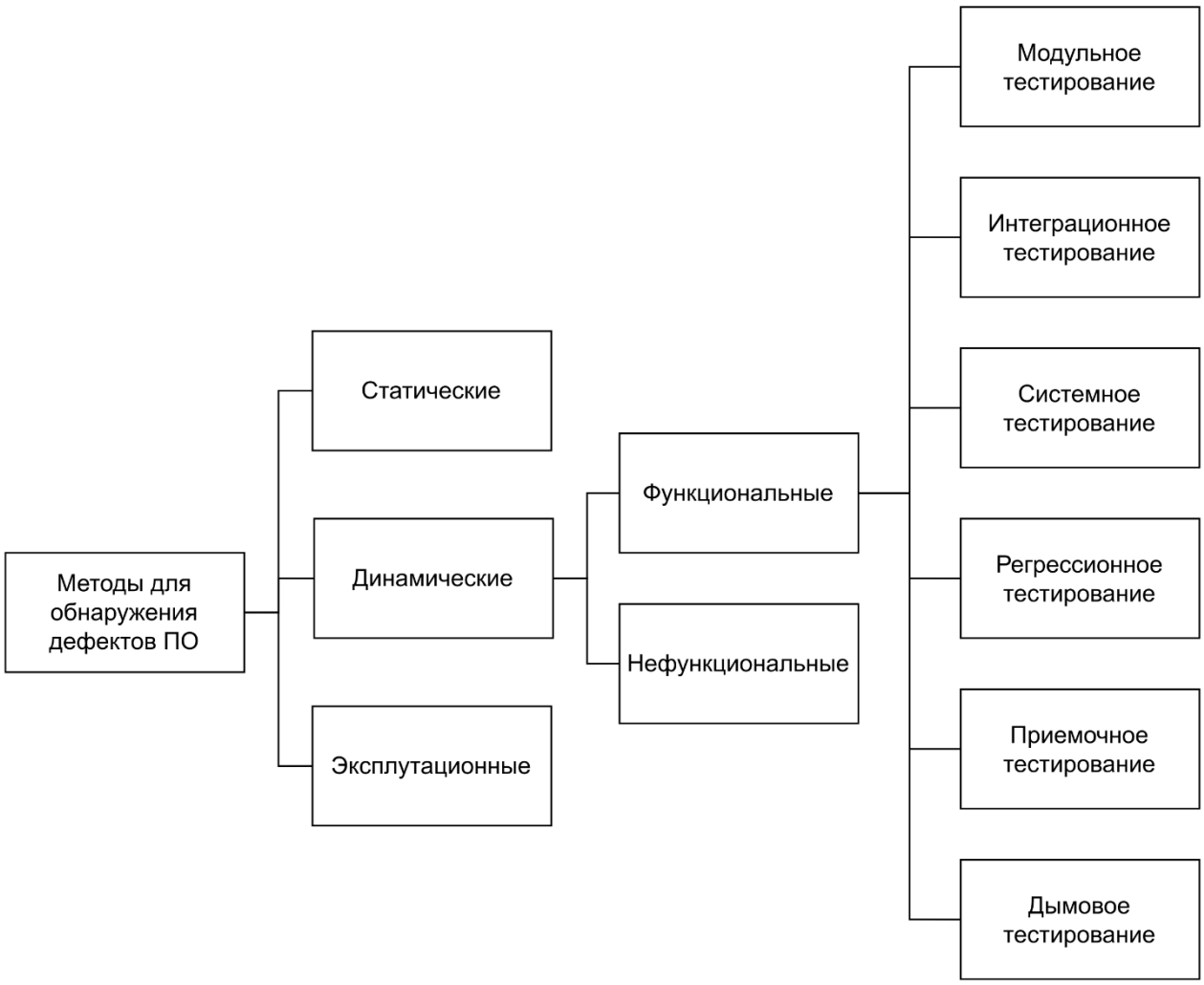

Рисунок 1.2 – Классификация методов для обнаружения дефектов ПО

1.2 Машинное обучение

1.2.1 Понятие машинного обучения

Искусственный интеллект – это комплекс технологических решений, позволяющий имитировать когнитивные функции человека (включая самообучение и поиск решений без заранее заданного алгоритма) и получать при выполнении конкретных задач результаты, сопоставимые, как минимум, с результатами интеллектуальной деятельности человека [8].

Машинное обучение – это математическая дисциплина, в основе которой лежат теория вероятностей, математическая статистика, численные методы

оптимизации, дискретный анализ, основной целью которого является выделение знаний из имеющихся данных. Под знанием подразумевается обученная модель, способная совершать предсказания на новых поступающих данных [9].

С машинным обучением тесно связаны понятие глубокого обучения. **Глубокое обучение** – это разновидность машинного обучения на основе искусственных нейронных сетей. Процесс обучения называется глубоким, так как структура искусственных нейронных сетей состоит из нескольких входных, выходных и скрытых слоев. Каждый слой содержит единицы, преобразующие входные данные в сведения, которые следующий слой может использовать для определенной задачи прогнозирования [10]. Соотношение рассматриваемых терминов представлено на рисунке 1.3.

Рисунок 1.3 – Отношение машинного обучения к искусственному интеллекту и глубокому обучению

Важность машинного обучения возрастает не только в исследованиях, имеющих отношение к компьютерным наукам, но и в нашей *повседневной жизни*. Оно позволяет пользоваться фильтрами почтового спама, удобным ПО распознавания текста и речи, распознаванием лиц, поисковыми механизмами. Кроме того, произошел прогресс в медицинской области. Например,

исследователи продемонстрировали, что модели глубокого обучения способны обнаруживать рак кожи с почти человеческой точностью [11].

1.2.2 Типы машинного обучения

Выделяют три основных типа машинного обучения: с учителем, с подкреплением и без учителя. Рассмотрим каждый из них.

1. **Обучение с учителем**

Главная цель – обучить модель на помеченных обучающих данных, что позволит вырабатывать прогнозы на не встречавшихся ранее или будущих данных. Понятие «с учителем» относится к набору обучающих образцов (входных данных), где желаемые выходные сигналы (метки) уже известны. На рисунке 1.4 представлена последовательность действий при обучении с учителем.

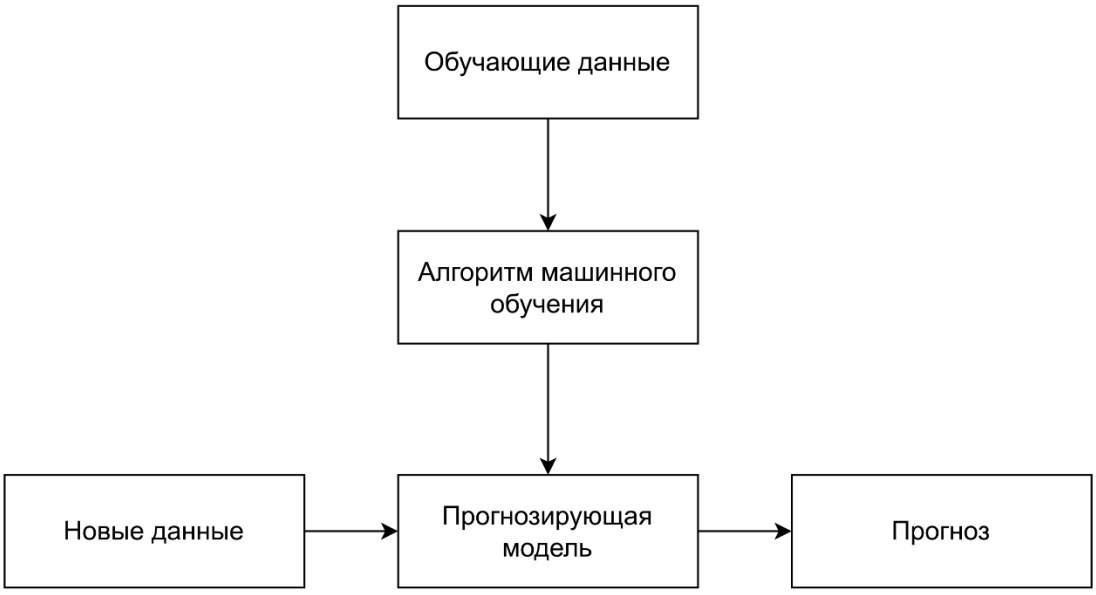

Рисунок 1.4 – Последовательность действий при обучении с учителем

К основным задачам, решаемым при помощи обучения с учителем, относят задачи классификации и регрессии [11].

2. **Обучение с подкреплением**

Целью такого обучения является разработка системы (агента), которая улучшает свои характеристики на основе взаимодействий со средой. Концепция обучения с подкреплением представлена на рисунке 1.5.

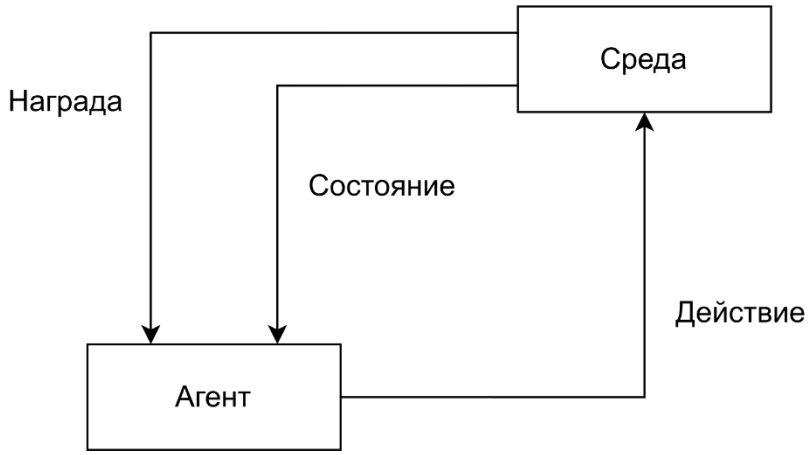

Рисунок 1.5 – Концепция обучения с подкреплением

Информация о текущем состоянии среды обычно включает так называемый сигнал награды. Такая обратная связь не будет истинной меткой или значением, а будет являться мерой того, насколько хорошо действие было оценено функцией наград. Агент использует обучение с подкреплением для выявления последовательности действий, которые доводят до максимума награду, применяя исследовательский метод проб и ошибок [11].

3. **Обучения без учителя**

При обучении с учителем правильный ответ известен заранее, с подкреплением – определяется мера награды для отдельных действий, предпринимаемых агентом. При обучении же без учителя данные приходят непомеченные. Использование приемов обучения без учителя дает возможность исследовать структуру данных для извлечения значимой информации без управления со стороны известной целевой переменной или функции награды. Обучение без учителя применяется, например, при решении задач кластеризации и понижения размерности [11].

1.2.3 Машинное обучение для обнаружения дефектов ПО

Прогнозирование дефектов является важной частью цикла разработки ПО. Ведь наличие дефектов сильно влияет на надежность, качество и стоимость обслуживания. Прогнозирование модулей с ошибками до развертывания повышает общую производительность [12]. Знание о компонентах, содержащих наибольшее число дефектов, позволяет распределить ресурсы тестирования так, чтобы в первую очередь проверялись компоненты с высокой вероятностью наличия дефектов [4]. Сложно составить правила при поиске дефектов, так как могут встретиться совершенно разные ошибки, поэтому довольно известными являются методы машинного обучения, которые решают данную проблему, обучаясь на примерах. Исследователи применяли различные алгоритмы для решения данной задачи. На рисунке 1.6 представлена общая схема процесса обучения модели обнаружения дефектов ПО.

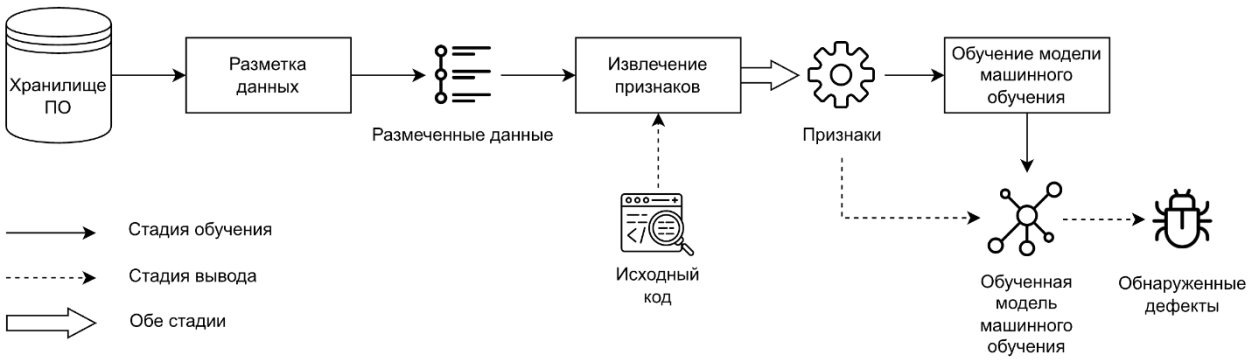

Рисунок 1.6 – Процесс обучения модели обнаружения дефектов ПО

Первым шагом построения модели является создание и идентификация положительных и отрицательных образцов из набора данных. Каждый образец может представлять собой программный компонент, файл исходного кода, класс или функцию в зависимости от выбранной степени детализации.

Экземпляр имеет метрики и метки, которые указывают, склонен он к дефектам или нет. Затем они передаются в модель машинного обучения для обучения. Наконец, обученная модель может классифицировать различные фрагменты кода как ошибочные или безопасные на основе закодированных знаний [12].

1.3 Методы машинного обучения для обнаружения дефектов ПО

1.3.1 Существующие методы

Для обучения моделей во многих статьях [13-18] используются классические алгоритмы машинного обучения такие, как дерево решений, алгоритм случайного леса, градиентный бустинг, метод опорных векторов, наивный байесовский классификатор [12], решающие задачу классификации. Приведем описание каждого из них.

1. **Наивный байесовский классификатор**

Наивный байесовский классификатор – простой классификатор, основанный на применении теоремы Байеса с наивным предположением о независимости. Для каждого класса рассчитывается вероятность по теореме Байеса – формула (1), и объект d считается принадлежащим классу c_j ($c_j \in C$), если при этом классе достигается наибольшая апостериорная вероятность: $\max_c P(c_j|d)$. Для рассматриваемой задачи имеются два класса в зависимости от того, есть дефект в программе или нет.

$$P(c_j|d) = \frac{P(d|c_j)P(c_j)}{P(d)} \approx P(d|c_j)P(c_j), \quad (1)$$

где $P(d|c_j)$ – вероятность того, что объект d принадлежит классу c_j;

$P(c_j)$ и $P(d)$ – априорные вероятности класса c_j и объекта d (последняя не влияет на выбор класса и может быть опущена).

Если же сделать «наивное» предположение, что все признаки, описывающие классифицируемые объекты, не связаны друг с другом, то $P(d|c_j)$ можно вычислить как произведение вероятностей встретить признак x_i ($x_i \in X$) среди объектов класса c_j:

$$P(d|c_j) = \prod_{i=1}^{|X|} P(x_i|c_j). \quad (2)$$

На практике при умножении малых вероятностей может наблюдаться потеря значащих разрядов. В связи с этим применяют логарифмы вероятностей. Так как логарифм – монотонно возрастающая функция, то класс c_j с наибольшим значением логарифма вероятности останется наиболее вероятным. Тогда решающее правило принимает следующий вид:

$$c^* = \underset{c}{\operatorname{argmax}} \left[\log P(c_j) + \sum_{i=1}^{X} P(x_i|c_j) \right]. \quad (3)$$

Несмотря на простоту, наивные байесовские классификаторы могут иметь достаточно высокую точность и производительность по сравнению с другими алгоритмами. К достоинствам также можно отнести малое количество данных, необходимых для обучения [19].

2. Метод опорных векторов

В случае решения задачи методом опорных векторов рассматривается задача классификации на два непересекающихся класса, в котором объекты описываются n-мерными вещественными векторами. Необходимо найти и построить гиперплоскость вида $w^T x + b = 0$, разделяющую объекты на два подмножества с максимальной граничной областью. В случае линейной разделимости – построение сводится к решению задачи условной оптимизации, которая с помощью методов Лагранжа может быть сведена к задаче квадратичного программирования. Однако в реальных задачах такое

встречаются крайне редко. Для решения линейно неразделимых задач обычно применяют два подхода.

1. Ослабить жесткие ограничения, что приводит к так называемой «мягкой» граничной области – позволяет некоторым точкам нарушать ограничения стандартного метода. В частности, вводятся дополнительные переменные для учета количества ошибок классификации.
2. Применить ядерные функции для линеаризации нелинейных задач – идея заключается в определении ядерной функции, основанной на скалярном произведении данных как нелинейного перехода от пространства входных данных к пространству с большим количеством измерений с целью сделать задачу линейно разделимой. В результате использование ядерных функций делает алгоритм нечувствительным к размерности пространства [20].

3. Дерево решений

Деревья решений зачастую применяются в задачах классификации – принимается решение о принадлежности объекта к одному из непересекающихся классов. Деревья состоят из вершин, в которых записываются проверяемые условия (признаки), и листьев, в которых записаны ответы дерева (один из классов). Обучение состоит в настройке условий в узлах дерева и ответов в его листьях с целью достижения максимального качества классификации.

Пусть заданы конечное множество объектов $X = \{x_1, \dots, x_L\}$ и алгоритмов $A = \{a_1, \dots, a_D\}$ и бинарная функция потерь $I: A \times X \to \{0,1\}$. $I(a,x) = 1$ тогда и только тогда, когда алгоритм допускает ошибку на объекте x. Число ошибок алгоритма a на выборке X определяется как $n(a,x) = \sum_{x \in X} I(a,x)$. Частота ошибок алгоритма на выборке определяется как $\nu = \frac{n(a,x)}{|x|}$. Под *качеством классификации* понимается частота ошибок алгоритма на контрольной выборке.

Преимущества:

— интерпретируемость – позволяют строить правила в форме, понятной эксперту;

— автоматический отбор признаков – признаки в вершины дерева выбираются автоматически из набора признаков;
— управляемость – если некоторые примеры классифицируются неправильно, можно заново обучить только те вершины дерева, из-за которых это происходит. Кроме того, при тренировке разных поддеревьев могут оказаться более эффективными разные алгоритмы обучения.

Недостатки:

— зависимость от сбалансированности обучающих примеров – при неправильных пропорциях классов в обучающей выборке дерево обучится некорректно;
— требуются методы предотвращения переобучения – явление переобучения возникает из-за излишней сложности модели, когда обучающих данных недостаточно. При их нехватке высока вероятность выбрать закономерность, которая выполняется только на этих данных, но не будет верна для других объектов;
— экспоненциальное уменьшение обучающий выборки – так как после обучения каждой вершины дерева происходит разделение на два подмножества, то на каждом следующем уровне дерева обучающее множество вершины содержит все меньше и меньше примеров [21].

4. Алгоритм случайного леса

Проблема переобучения, возникающая при использовании дерева решений, может быть решена лесом решений – несколько деревьев, при этом результат определяется путем голосования.

Пусть имеется множество деревьев решений, каждое из которых относит объект $x \in X$ к одному из классов $c \in Y$. Считаем, что если $f_c^t(x) = 1$, то дерево t относит объект $x \in X$ к одному из классов c. При использовании алгоритма простого голосования для каждого класса подсчитывается число деревьев, относящих объект к данному классу – формула (4).

$$G_c(x) = \frac{1}{T_c}\sum_{t=1}^{T_c} f_c^t(x)\,, c\epsilon Y. \quad (4)$$

Ответом леса является тот класс, за который подано наибольшее число голосов:

$$\alpha(x) = \underset{c\in Y}{\operatorname{argmax}}\, G_c(x). \quad (5)$$

Если обучать деревья на одном и том же множестве тренировочных примеров одним и тем же методом, получатся одинаковые или очень похожие деревья. Поэтому для достижения независимости ошибок деревьев, составляющих лес решений, применяются специальные методы, например, случайный лес. Для каждого дерева случайным образом выбираются объекты из обучающей выборки. При этом некоторые могут быть выбраны несколько раз, а некоторые вовсе пропущены. В результате создается новое подмножество, на котором и обучается модель [21].

5. Бустинг

Бустинг – модификация алгоритма случайного леса, обучение происходит путем последовательного обучения нескольких моделей для повышения точности всей системы. Выходным данным отдельных деревьев присваиваются веса. Затем неправильным классификациям из первого дерева решений присваивается больший вес, после чего данные передаются в следующее дерево. После многочисленных циклов бустинг объединяет «слабые» классификаторы в один алгоритм [22].

Существуют две основные разновидности бустинга: адаптивный и градиентный.

1. **Адаптивный бустинг** (англ. AdaBoost) – одна из самых ранних реализаций бустинга, которая адаптируется и самостоятельно корректирует классификаторы в каждой итерации бустинга.

Тренировочным примерам назначаются веса $w_1^1, \dots, w_m^1$. Так как они имеют вероятностную природу, для них выполняется условие: $\sum_{j=1}^{m} w_j^1 = 1, w_j^1 \in [0,1]$. Начальное распределение весов является равномерным. Происходит обучение первого дерева, с помощью которого производится классификация тренировочных примеров. Веса правильно классифицированных примеров снижаются, неправильно – повышаются. Следующее дерево строится с учетом обновленных весов, и так далее до достижения заданного количества деревьев или требуемой ошибки классификации [21].

2. **Градиентный бустинг** (англ. Gradient Boosting) – похож на адаптивный бустинг, разница состоит в том, что он не присваивает неправильно классифицированным элементам больший вес. Вместо этого модель градиентного бустинга оптимизирует функцию потерь, используя градиентный спуск, в результате чего текущая базовая модель всегда становится эффективнее предыдущей. Градиентный бустинг пытается сразу генерировать точные результаты, а не исправлять ошибки [22].

1.3.2 Параметры сравнения методов

В результате классификации объекты помечаются как положительные (имеются дефекты) и отрицательные (дефекты отсутствуют). Для оценки качества работы полученных моделей используются различные метрики. Так, в статьях [14-18] при сравнении результатов рассматриваются такие метрики, как accuracy, точность (precision), полнота (recall) и F-мера. Для их определения используется матрица матрица ошибок, которая содержит 4 ячейки:

— верно-положительные объекты (TP) – объекты, которые были верно классифицированы как положительные;

— верно-отрицательные объекты (TN) – объекты, которые были верно классифицированы как отрицательные;
— ложно-положительные объекты (FP) – объекты, которые были ложно классифицированы как положительные;
— ложно-отрицательные объекты (FN) – объекты, которые были ложно классифицированы как отрицательные.

1. **Accuracy** (точность) – широко используемая метрика, представляет собой отношение всех правильных прогнозов к общему числу предсказанных образцов (формула (6)). В ряде задач (с неравными классами) метрика может являться неинформативной.

$$\text{accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}. \quad (6)$$

2. **Precision** (точность) – это доля прогнозируемых положительных результатов, которые действительно относятся к этому классу, от всех положительно предсказанных объектов – формула (7).

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}. \quad (7)$$

3. **Recall** (полнота) – пропорция всех верно-положительных предсказанных объектов к общему количеству действительно положительных (формула (8)). Чем выше значение полноты, тем меньше положительных примеров пропущено в классификации.

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}. \quad (8)$$

4. **F-measure** (F-мера) – взвешенное гармоническое среднее полноты и точности (формула (9)). Этот показатель демонстрирует, как много объектов классифицируется моделью правильно, и сколько истинных экземпляров она не пропустит [23].

$$\mathrm{F-measure} = \frac{\mathrm{precision} \cdot \mathrm{recall}}{\mathrm{precision} + \mathrm{recall}}. \quad (9)$$

1.3.3 Сравнение методов

Для сравнения рассмотренных методов использованы результаты, описанные в статьях [14-18]. Модели обучались на метриках, представленных в репозитории PROMISE [24]. Данные представляют собой набор метрик Маккейба и Холстеда для кода, написанного на языках программирования C и C++. В таблице 1 отображены средние арифметические значения для каждой из метрик по всем наборам данных и рассматриваемым статьям.

Таблица 1 – Сравнительная таблица результатов работы алгоритмов

Метрики / **Алгоритмы**	**Accuracy (точность)**	**Precision (точность)**	**Recall (полнота)**	**F-measure (F-мера)**
Наивный байесовский классификатор	0.795	0.845	0.803	0.849
Метод опорных векторов	0.841	0.901	0.879	0.902
Дерево решений	0.823	0.845	0.878	0.889
Случайный лес	0.845	0.859	0.863	0.890
Градиентный бустинг	**0.847**	**0.903**	**0.883**	**0.903**
Адаптивный бустинг	0.835	0.858	0.861	0.889

Таким образом, по каждой из метрик алгоритм градиентного бустинга показал наивысший результат. Можно сделать вывод, что его применение является наиболее выгодным для рассматриваемой задачи. Благодаря использованию ансамблю моделей и большому варьированию параметров при обучении, данный алгоритм имеет высокую перспективу использования.

1.4 Формализованная постановка задачи

Для дальнейшей работы необходимо формализовать задачу обнаружения дефектов ПО. Наиболее удобным способом для этого является диаграмма в нотации IDEF0. Поставленная задача представлена на рисунке 1.7. На вход системе подаются набор данных для обучения и исходный код на языке программирования C++, содержащий блоки с функциями. Система путем статического анализа кода с использованием алгоритма градиентного бустинга определяет и размечает в коде блоки, подверженные дефектам, то есть места в коде, в которых вероятней всего будет возникать отклонения от первоначальных бизнес-требований.

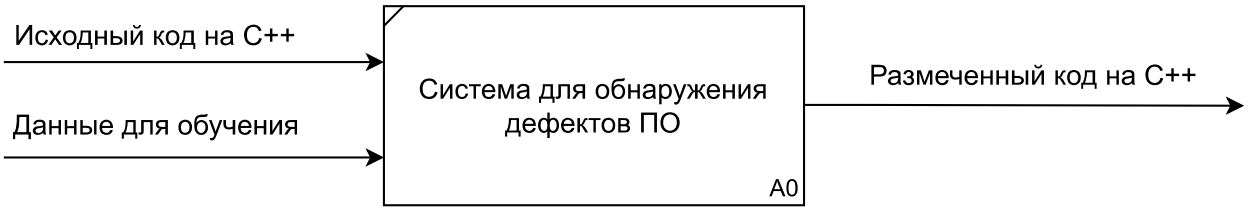

Рисунок 1.7 – IDEF0-диаграмма метода обнаружения дефектов ПО

Вывод

Таким образом, был представлен обзор дефектов разрабатываемого ПО, классифицированы методы для их обнаружения, описаны метрики и проведено сравнение методов машинного обучения для обнаружения дефектов ПО. На основе этого выбран метод, который будет разрабатываться в дальнейшем:

статический анализ кода для поиска программных дефектов с использованием алгоритма градиентного бустинга. Проведена формализация задачи обнаружения дефектов ПО в виде диаграммы в нотации IDEF0.

2 Конструкторский раздел

2.1 Метод обнаружения дефектов ПО

Предлагаемый метод обнаружения дефектов разрабатываемого ПО представляет собой анализ текста программ, написанных на языке программирования C++. Предполагается, что на вход подается код, состоящий из функций, каждая из которых занимает не более 2000 строк, и написанный с соблюдением общих правил оформления кода.

Данный метод включает в себя несколько этапов. Перед выполнением процесса обнаружения дефектов происходит обучение модели градиентного бустинга на обучающей выборке. Код, для которого нужно определить наличие дефектов, делится на блоки, содержащие входящие функции. Для каждого блока вычисляются метрики. Вычисленные значения поступают обученной модели, где происходит классификация каждого из блоков по наличию или отсутствию дефекта. Последним этапом метода является разметка: необходимо графически отметить места в коде, в которых вероятней всего имеются дефекты, а также вероятность этого для каждого блока.

На рисунках 2.1 представлена детализированная IDEF0-диаграмма разработанного метода.

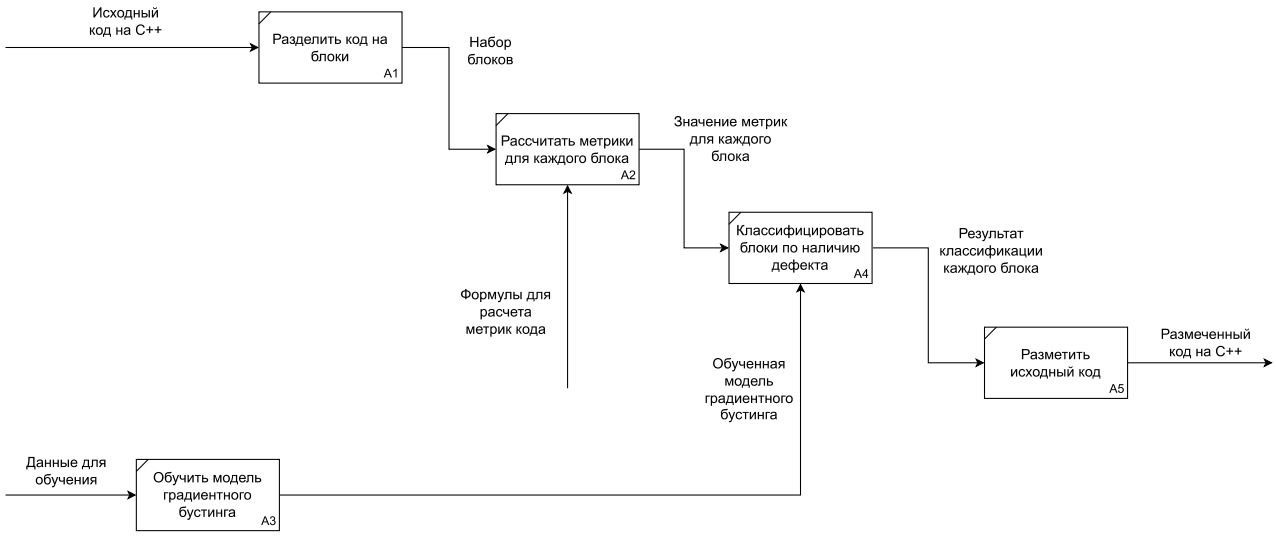

Рисунок 2.1 – Детализированная IDEF0-диаграмма разработанного метода обнаружения дефектов ПО

2.2 Обучение модели градиентного бустинга

Классификация модулей происходит с помощью алгоритма градиентного бустинга. В основе данного алгоритма лежит использование деревьев решений. Поэтому разработку следует начинать именно с них.

Целью алгоритма является создание модели, которая представляет собой бинарное дерево, состоящее из узлов, ребер и листьев. В каждом узле содержится некоторое подмножество примеров и признак, на основе которого создаются дочерние элементы. Ребра соответствуют значениям этого признака. В листьях, в отличие от узлов, отсутствует признак. В них указывается доля примеров, класс которых наиболее встречается в подмножестве.

Процесс построения дерева начинается с корневого узла, содержащего все примеры. По выбранному критерию определяется признак, который лучше всего разделяет данные на два подмножества с сохранением в них как можно большего количества одинаковых значений целевой переменной. Для поставленной задачи выбран критерий Джини – формула (10). Значение данного выражения необходимо минимизировать. Для этого выборка из узла поочередно разбивается

на две части по каждому признаку (в одной части значения признака в примерах меньше медианы, в другой – больше) и выбирается максимальное значение.

$$H(R) = 2p_+(1 - p_+), \tag{10}$$

где p_+ – доля объектов, принадлежащих положительному классу (имеющих дефект) и попавших в узел R.

Каждое из подмножеств становится узлом следующего уровня дерева. Процесс разбиения продолжается до тех пор, пока не будет достигнут критерий остановки – глубина дерева не больше максимального заданного. Далее в каждом из листовых узлов указывается доля примеров, значение целевой переменной которых соответствует наличию дефектов (объекты положительного класса).

После обучения можно классифицировать приходящие объекты. Для этого необходимо «спуститься» по дереву до соответствующего листа на основе заданных правил. По полученному значению можно сделать вывод, имеется в заданном коде дефект или нет.

На рисунке 2.2 представлен алгоритм дерева решений.

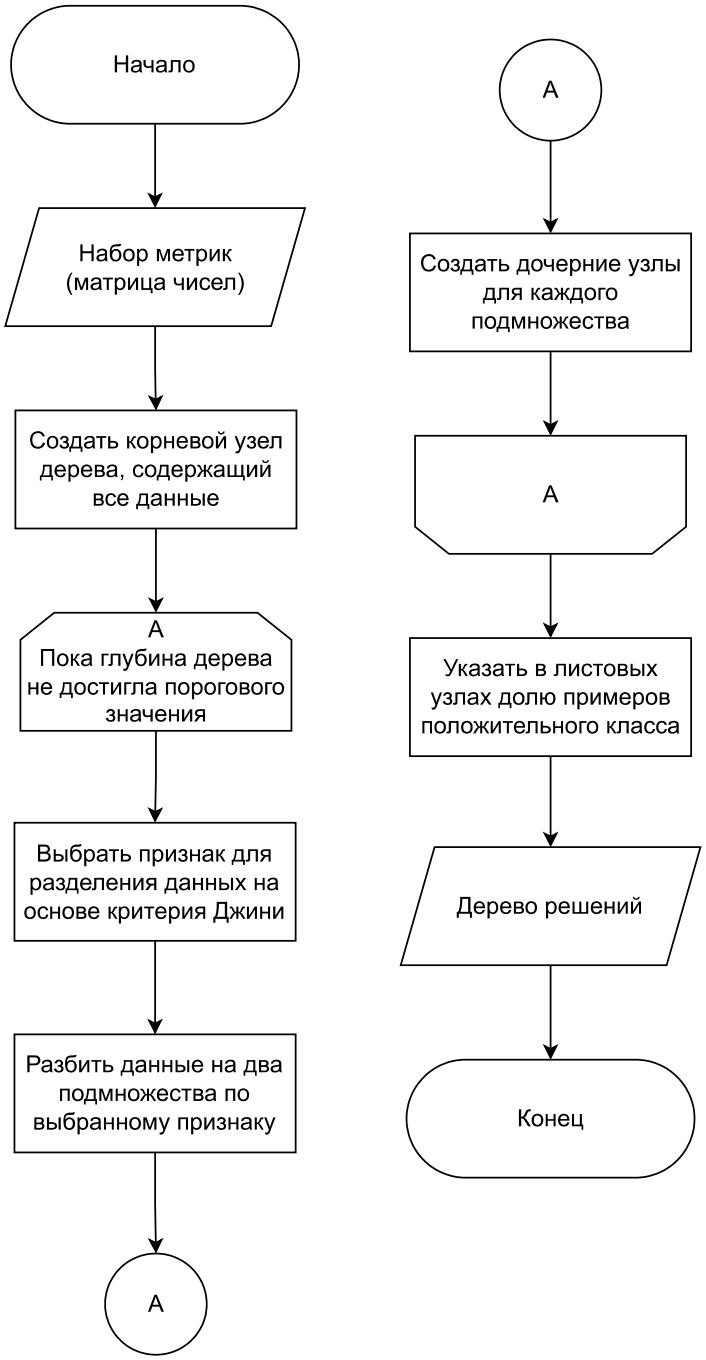

Рисунок 2.2 – Схема алгоритма дерева решений

Градиентный бустинг является разновидностью ансамблей деревьев. Деревья в нем обучаются последовательно, при этом каждое новое учитывает ошибки предыдущих, за счет чего достигается большая точность.

Итоговая модель градиентного бустинга представляет композицию деревьев – формула (11). Сумма предсказаний каждого дерева является вероятностью, с которой во входной программе имеется дефект. Однако такой

подход может привести к переобучению. Для решения этой проблемы вводят специальный параметр, называемый темпом обучения (англ. learning rate). Благодаря нему, каждый алгоритм вносит относительно небольшой вклад в общий результат. Значение параметра обычно определяется эмпирически.

$$a_k(x) = \eta b_1(x, y_1) + \eta b_2(x, y_2) + \cdots + \eta b_k(x, y_k), \quad (11)$$

где k – количество деревьев, участвующих при построении модели;

η – темп обучения, $\eta \in (0, 1]$;

$b_j(x, y_j)$ – результат классификации j-го дерева, $j \in \overline{1; k}$;

x – значения метрик для каждого объекта;

y_j – значения целевой переменной для каждого объекта.

Первое дерево строится обычным образом, обучаясь на входных данных. Каждое последующее должно улучшать результат путем оптимизации функции потерь. Для решения исследуемой задачи используется логарифмическая функция потерь (англ. Log Loss) – формула (12). На каждом шаге значение данного выражения необходимо минимизировать. Для этого в качестве целевой переменной при построении очередного дерева используется антиградиент – значение, противоположное градиенту (формула (13)). Стоит отметить, что действительные значения y_i соответствуют антиградиентам, вычисленным на предыдущем шаге (кроме первого).

$$\mathcal{L}(y, p) = -(y \cdot \log(p) + (1 - y) \cdot \log(1 - p)), \quad (12)$$

где y – действительное значение целевой переменной;

p – предсказанное значение целевой переменной.

$$g_i^k = \frac{\partial \mathcal{L}(y_\text{i}, p)}{\partial p}\bigg|_{p=a_k(x_i)} = -\frac{a_k(x_i) - y_i}{a_k(x_i)(a_k(x_i) - 1)}, \quad (13)$$

где y_i – действительное значение целевой переменной для i-го объекта;

$a_k(x_i)$ – предсказанное значение целевой переменной для i-го объекта;

x_i – значения метрик для i-го объекта.

Ансамбль строится до тех пор, пока не будет достигнут критерий остановки – достигнуто заданное количество итераций.

На рисунке 2.3 представлена схема алгоритма градиентного бустинга.

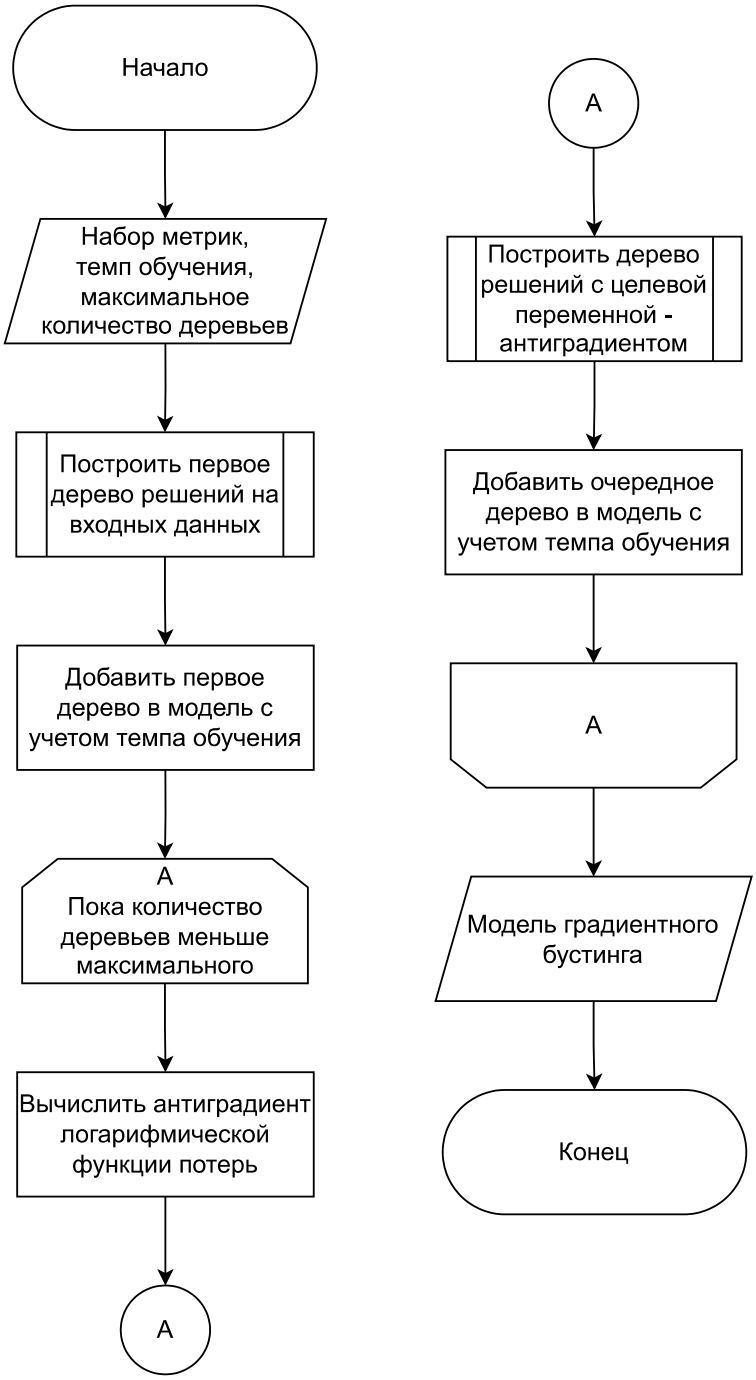

Рисунок 2.3 – Схема алгоритма градиентного бустинга

2.3 Расчет метрик кода

Модели, описанные в анализируемых статьях, обучались на метриках, представленных в репозитории PROMISE. Поэтому целесообразно использовать именно эти данные. Предполагается использовать для обучения набор из данного репозитория под названием JM1.

Обучающая выборка представляет собой вычисленные метрики Маккейба и Холстеда для кода, написанного на языке программирования C и C++. Каждый пример – это набор из 18 чисел.

1. LOC (англ. McCabe's line count of code) – количество всех непустых строк кода без комментариев. Для подсчета необходимо удалить из текста программы все комментарии. Затем разбить на строки по пробельным символам и посчитать их количество.
2. $v(g)$ (англ. McCabe's cyclomatic complexity) – цикломатическая сложность программы. Данная метрика отображает количество линейно независимых маршрутов через программный код. В случае, если число подпрограмм равно одному, цикломатическая сложность может быть подсчитана на основе числа точек ветвлений и точек выхода в программе – формула (14). Число ветвлений соответствует количеству условных операторов, операторов цикла и т.д., то есть конструкций, меняющих ход выполнения программы.

$$v(g) = \pi - s + 2, \tag{14}$$

где π – число точек ветвления в программе;

s – число точек выхода в программе.

3. N (англ. Halstead's total operators and operands) – общее количество операторов и операндов в программе. Для подсчета операторов можно выделить все существующие операторы в C++ и посчитать их количество в коде, операндов – разбить код по выделенным операторам и посчитать,

сколько получилось строк, при этом без учета ключевых слов, типов данных и т.д.

4. V (англ. Halstead's volume) – объем, описывает размер реализации программы. Рассчитывается по формуле (15).

$$V = N \cdot \log_2(n), \tag{15}$$

где n – общее количество уникальных операторов и операндов.

5. D (англ. Halstead's difficulty) – сложность программы, пропорциональна количеству уникальных операторов. Она может быть вычислена по формуле (16).

$$D = \frac{n_1}{2} \cdot \frac{N_2}{n_2}, \tag{16}$$

где n_1 – общее количество уникальных операторов в программе;

N_2 – общее количество операндов в программе;

n_2 – общее количество уникальных операндов в программе.

6. I (англ. Halstead's intelligence) – информационное содержание программы, позволяет определить умственные затраты на создание. Рассчитывается по формуле (17).

$$I = \frac{V}{D}. \tag{17}$$

7. E (англ. Halstead's effort) – оценка необходимых усилий по реализации программы. Вычисляется по формуле (18).

$$E = V \cdot D. \tag{18}$$

8. B (англ. Halstead's number of delivered bugs) – количество предполагаемых ошибок. Коррелирует с общей сложностью кода и может быть рассчитана по формуле (19).

$$B = \frac{V}{3000}. \tag{19}$$

9. T (англ. Halstead's time estimator) – время реализации программы, пропорционально величине E. Рассчитывается по формуле (20). Холстед обнаружил, что деление на 18 дает наиболее близкое время в секундах.

$$T = \frac{E}{S}, \tag{20}$$

где S – специальный коэффициент для калибровки T, равен 18.

10. $loCode$ (англ. Halstead's line count) – количество строк кода вместе с комментариями и пустыми строками. Считается количество строк, которые можно визуально выделить в тексте программы.
11. $loComment$ (англ. Halstead's count of lines of comments) – количество строк комментариев в программе.
12. $loBlank$ (англ. Halstead's count of blank lines) – количество пустых строк.
13. $loCodeAndComment$ (англ. Halstead's mixed count) – количество строк в коде, содержащих как код, так и комментарии.
14. $uniq\_Op$ – количество уникальных операторов.
15. $uniq\_Opnd$ – количество уникальных операндов.
16. $total\_Op$ – общее количество операторов.
17. $total\_Opnd$ – общее количество операндов.

18. $defects$ – целевая переменная, принимает два значения: true (в программе имеется один или несколько зарегистрированных дефектов) или false (в программе дефекты отсутствуют).

2.4 Классификация кода по наличию дефекта

Полученные значения метрик поступают на вход обученной модели градиентного бустинга. Данный набор необходимо классифицировать на каждом из деревьев. Результатом являются вероятности нахождения дефекта в рассматриваемом блоке кода. Следующий этап – вычисление общего прогноза с учетом темпа обучения: прогноз каждого дерева умножается на данный коэффициент, и суммируются все значения. Итогом работы является определение наличия дефекта во входной программе и соответствующая вероятность его присутствия.

На рисунке 2.4 представлена детализированная диаграмма в нотации IDEF0 метода определения наличия дефекта в коде.

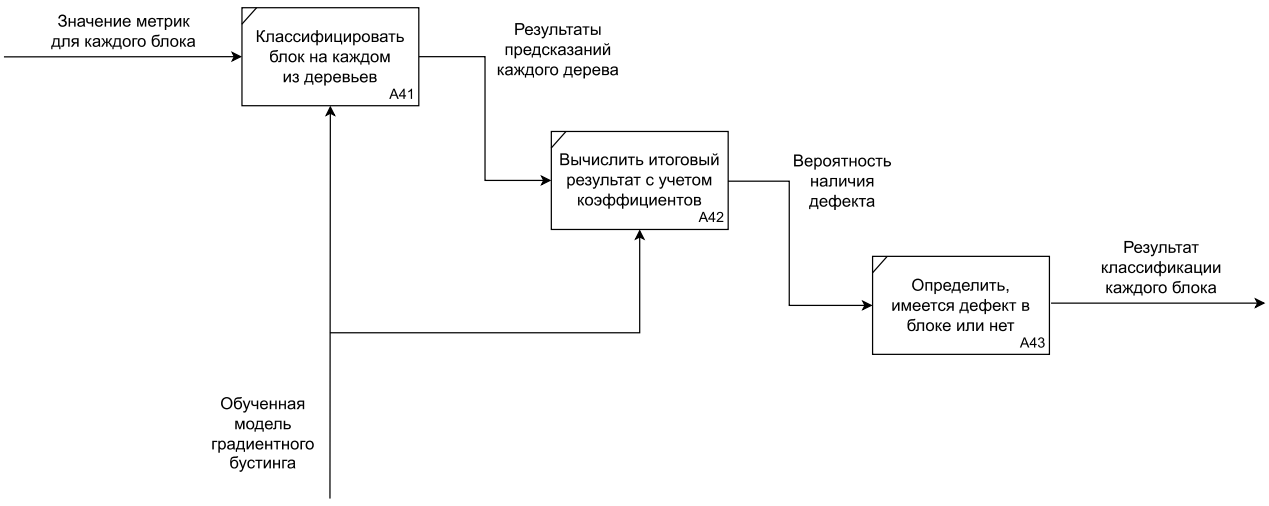

Рисунок 2.4 – Детализированная IDEF0-диаграмма метода определения наличия дефектов в блоке

Вывод

Таким образом, был разработан метод обнаружения дефектов ПО, включающий в себя 5 этапов. Метод представлен в виде детализированной IDEF0-диаграммы. Обнаружение дефектов происходит с помощью модели градиентного бустинга, разработан соответствующий алгоритм и схемы к нему. Входной код обрабатывается путем расчета метрик Маккейба и Холстеда. Приведены описания каждой из них, а также соответствующие формулы. Разработана IDEF0-диаграмма метода определения наличия дефектов в блоке кода.

3 Технологический раздел

3.1 Выбор средств программной реализации

В качестве языка программирования выбран Python [25]. Данный язык позволяет достаточно быстро разрабатывать, при этом поддерживает объектную-ориентированную парадигму программирования. Также Python имеет обширное количество библиотек и фреймворков, в том числе для машинного обучения.

Для предобработки данных используется библиотека scikit-learn [26], которая содержит множество функций для различных преобразований. Обучение производится с применением библиотеки XGBoost [27], которая позволяет построить оптимизированную модель градиентного бустинга с варьированием множества параметров.

Для разработки desktop-приложения выбран фреймворк PyQt5 [28]. Ввиду своей простоты, высокой производительности и кроссплатформенности данное решение является наиболее оптимальным. При этом для создания графического интерфейса можно использовать QtDesigner [29].

В качестве среды разработки используется PyCharm – удобная среда для работы с Python, которая предоставляет широкий набор инструментов и функций.

3.2 Программная реализация разработанного метода

Реализованное программное обеспечение имеет многомодульную структуру. Каждый модуль соответствует определенному этапу разработанного метода. На рисунке 3.1 представлена диаграмма компонентов программного продукта.

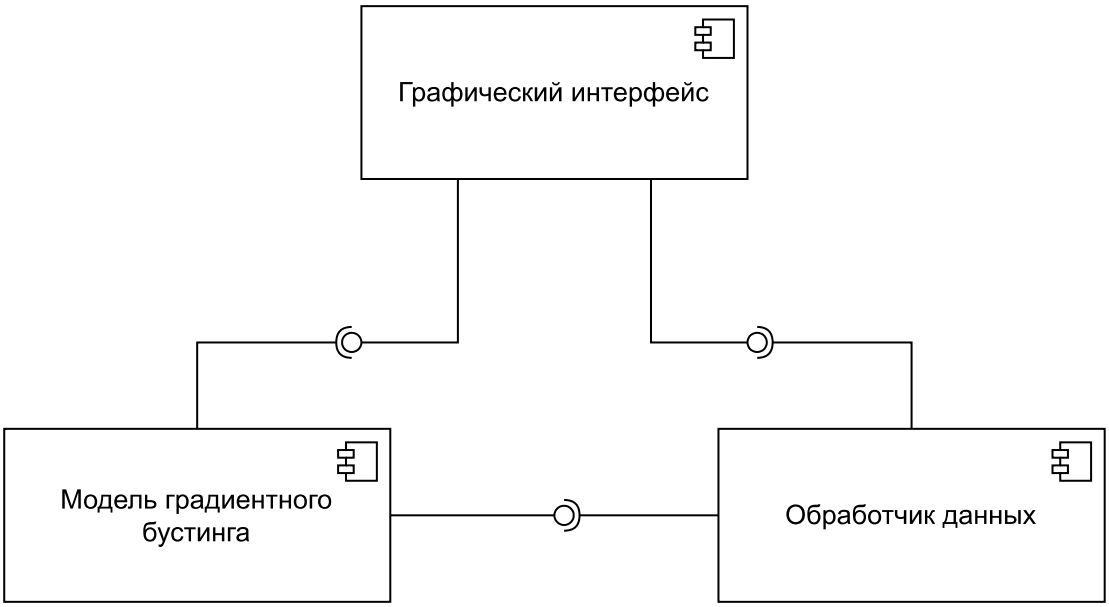

Рисунок 3.1 – Диаграмма компонентов разработанной программы

1. Пользовательский интерфейс представляет собой компонент реализации графического интерфейса системы обнаружения дефектов ПО.
2. Обработчик данных включает в себя предобработку и подготовку данных, полученных из репозитория, для обучения модели. Также производит расчет метрик для текста программы, введенного пользователем.
3. Компонент модели градиентного бустинга состоит из обучения модели и предсказания результатов для рассчитанных метрик исходного кода.

Разработанные модули реализованы в виде классов и вспомогательных функций.

1. PromiseDataset – класс, отвечающий за загрузку и обработку набора данных для обучения. Единственный метод prepare производит анализ данных, удаление выбросов, дубликатов, масштабирование. Происходит разделение выборки на тренировочную и тестовую – выделенные признаки и целевая переменная используются в модели.

2. GBDDModel – класс, в котором реализуется обучение модели методом градиентного бустинга. Одним из атрибутов является классификатор типа XGBClassifier из библиотеки XGBoost, применяющий соответствующий алгоритм. Стоит отметить, что обученная модель сохраняется в файл и загружается из него при классификации.

   Для подбора параметров используется метод grid_search, который в автоматическом режиме сопоставляет разные параметры и выбирает те, что в совокупности показывают наибольшую точность. Исследования показали – для данной задачи лучшим образом подходят следующие: learning_rate = 0.01 (темп обучения), n_estimators = 1000 (количество деревьев), max_depth = 7 (максимальная глубина обучаемых деревьев).

   Для контроля обучения применяется метод debug_fit, позволяющий по окончании отобразить на графике значения функции потерь на каждой итерации. Это дает возможность более точно изменять параметры модели и методы для обработки данных.

   Метод fit позволяет обучить модель и сохранить ее в соответствующий файл, методы predict и predict_proba рассчитывают результат классификации и вероятности принадлежности каждому классу соответственно. Для итоговой оценки точности модели используется функция get_statistics, выводящая на экран метрики, описанные в аналитической части работы (accuracy, precision, recall, f-measure).
3. MetricsCppCode – класс, производящий расчет метрик кода на основе описанных ранее формул. Метод count возвращает словарь, содержащий названия метрик и их значения. Также написаны вспомогательные функции для подсчета каждой из них в отдельности. Для поиска операторов и операндов, удаления комментариев, грамотного разбиения кода на строки используются регулярные выражения.
4. Window – класс, отвечающий за пользовательский интерфейс. Реализованы методы для обработки соответствующих кнопок: open_file – открытие файла и загрузка в текстовый редактор, clean_editor – очистка

поля с текстом, run_searching – запуск работы алгоритма и разметка кода на основе посчитанных вероятностей. Для разделения исходного кода на функции применяется split_code_by_lines, которая на основе регулярного выражения делит текст на блоки.

5. SyntaxHighlighter – класс, позволяющий отображать подсветку строк кода. В метод highlight_line передается номер строки и формат, который применяется для данной строки.

На рисунке 3.2 представлена диаграмма классов разработанной программы.

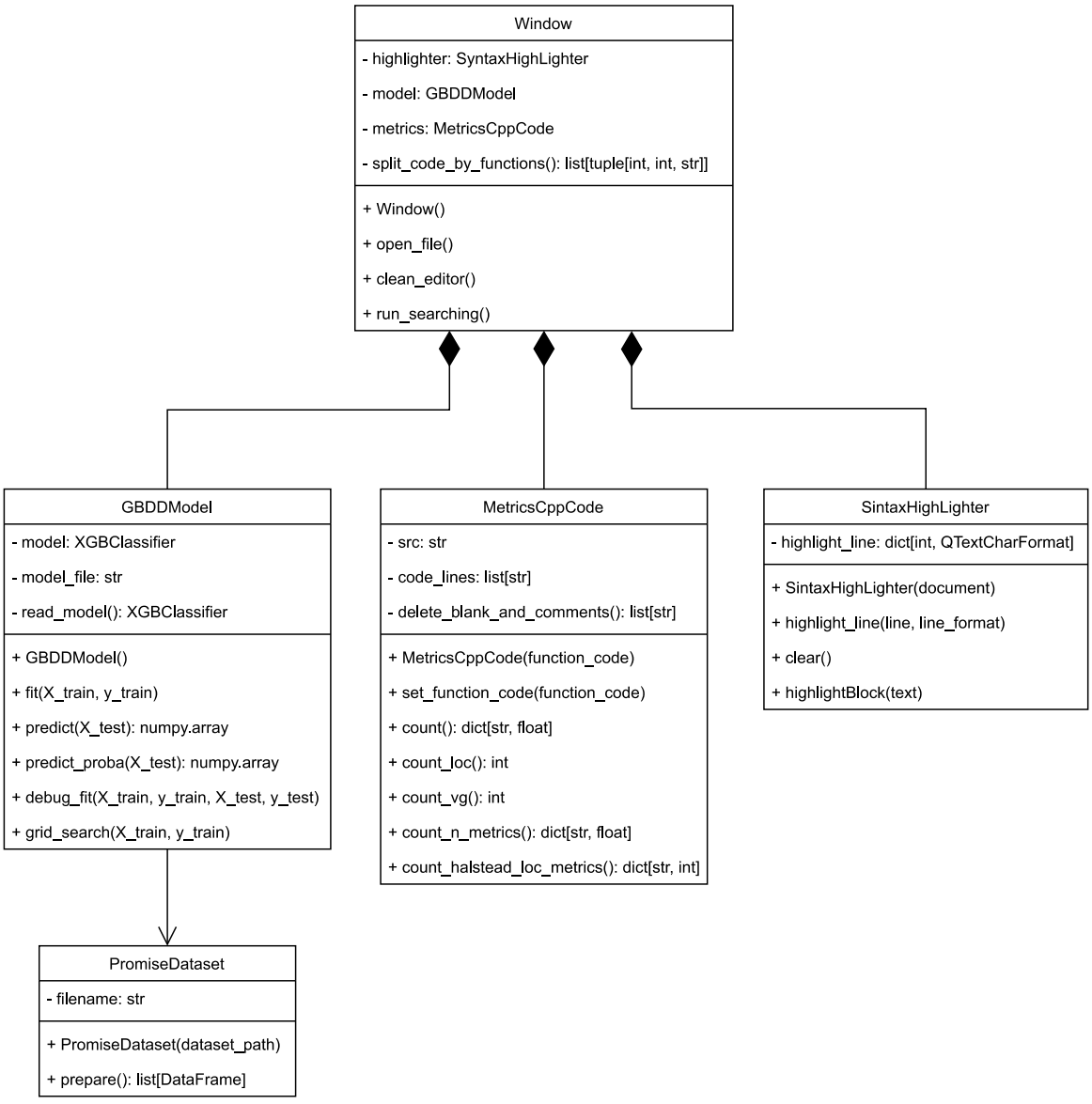

Рисунок 3.2 – Диаграмма классов разработанной программы

3.3 Системное тестирование

Для разработанной программы выполнено системное тестирование методом черного ящика, то есть вся система тестировалась целиком, важны только входные и выходные данные.

Для этого написаны примеры кода на C++, содержащие функции с разным количеством ошибок. Дефекты, операторы и операнды случайным образом распределены по программе. Использовались следующие дефекты:

— дефекты вычислений;
— дефекты адресной арифметики;
— дефекты явного и неявного преобразования типов;
— дефекты инициализации;
— дефекты размерности данных;
— дефекты форматов входных данных и другие.

Сгенерировано 1000 тестов. При этом в дополнительный файл вынесены доли строк с дефектами в каждом экземпляре.

Тестирование проводилось в автоматическом режиме. Для каждого теста считаются метрики кода, затем – вероятность нахождения в нем дефекта. Результаты, полученные из системы, сравниваются с посчитанными заранее значениями на основе заданной разницы. Более 75% случаев выявлены верно, что свидетельствует о достижении необходимого качества программы.

3.4 Взаимодействие пользователя с ПО

Система обнаружения дефектов ПО представляет собой приложение, состоящее из текстового поля для ввода и загрузки программы, а также кнопок загрузки файла, очистки окна и запуска алгоритма поиска дефектов.

Пользователь имеет возможность загрузить файл с кодом на C++ либо ввести его вручную. Выводится предупреждение об ограничения на размер исходных данных (не более 2000 строк). После нажатия на кнопку «Показать дефекты» код размечается разными цветами в зависимости от вероятности нахождения в функции дефектов:

— голубой – очень низкая, 0-20%;

— зеленый – низкая, 20-40%;

— желтый – средняя, 40-60%;

— оранжевый – высокая, 60-80%;

— красный – 80-100%.

На основе результатов можно сделать вывод, какие функции наиболее подвержены рискам и какие необходимо протестировать в первую очередь.

На рисунке 3.3 представлен пример работы программы.

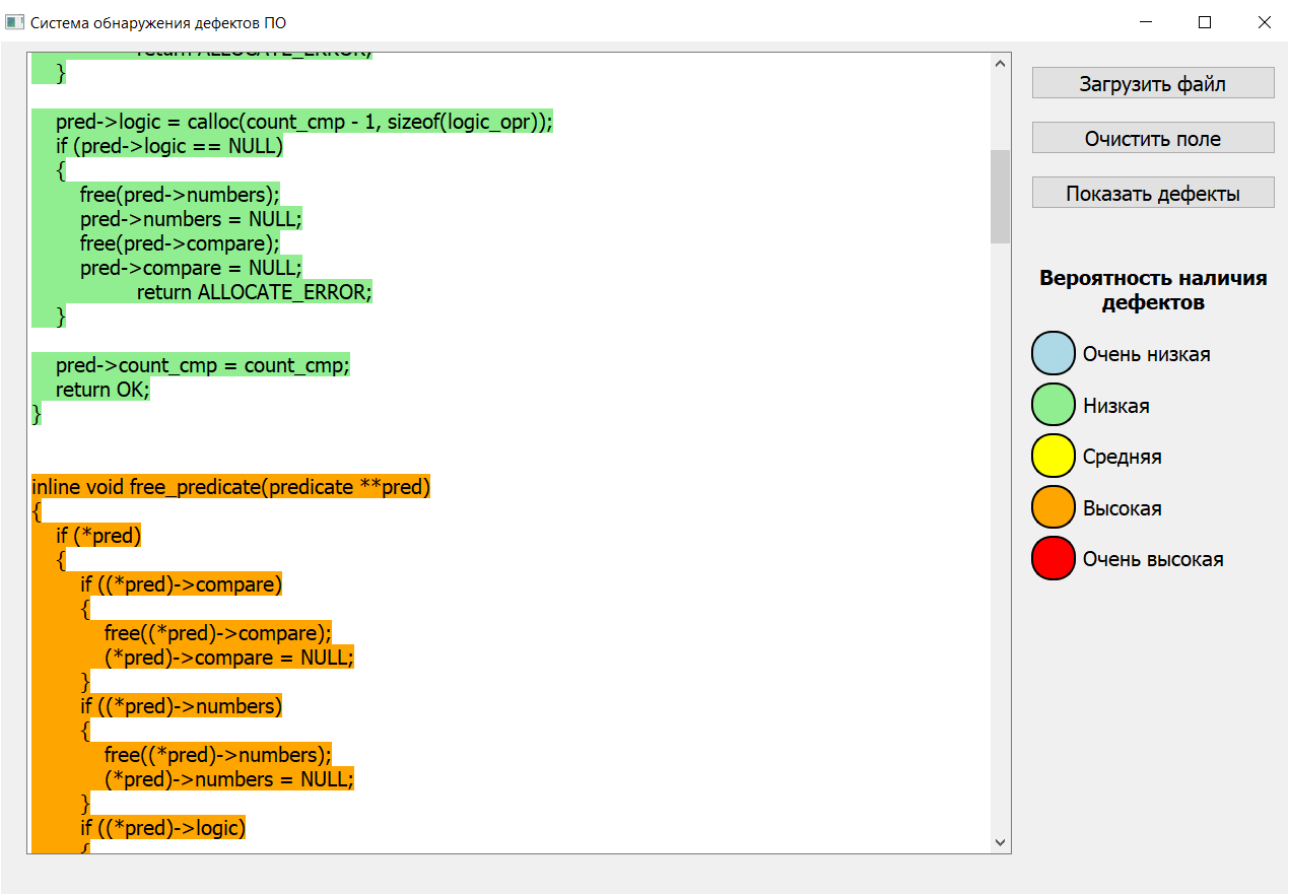

Рисунок 3.3 – Пример работы системы обнаружения дефектов ПО

Вывод

Таким образом, было приведено обоснование выбора средств разработки – язык программирования Python, библиотеки для разработки модели (scikit-learn, XGBoost), фреймворк для реализации приложения – PyQt5, среда разработки – PyCharm. Описана структуры кода программного продукта, разработаны диаграммы компонентов и классов. Реализовано программное обеспечение системы обнаружения дефектов, проведено системное тестирование, представлен пример работы программы.

4 Исследовательский раздел

4.1 Анализ точности метода

Исследование эффективности разработанного метода заключается в анализе точности реализованной модели градиентного бустинга и времени вычисления результата для кода с разным количеством строк.

Для оценки точности обучающая выборка делится на тренировочную и тестовую. Тестовая часть составляет 20% от общего количества. На каждой итерации считается результат на частично обученной модели и значение функции потерь, что позволяет понять, правильно ли идет обучение. На рисунке 4.1 представлена данная зависимость.

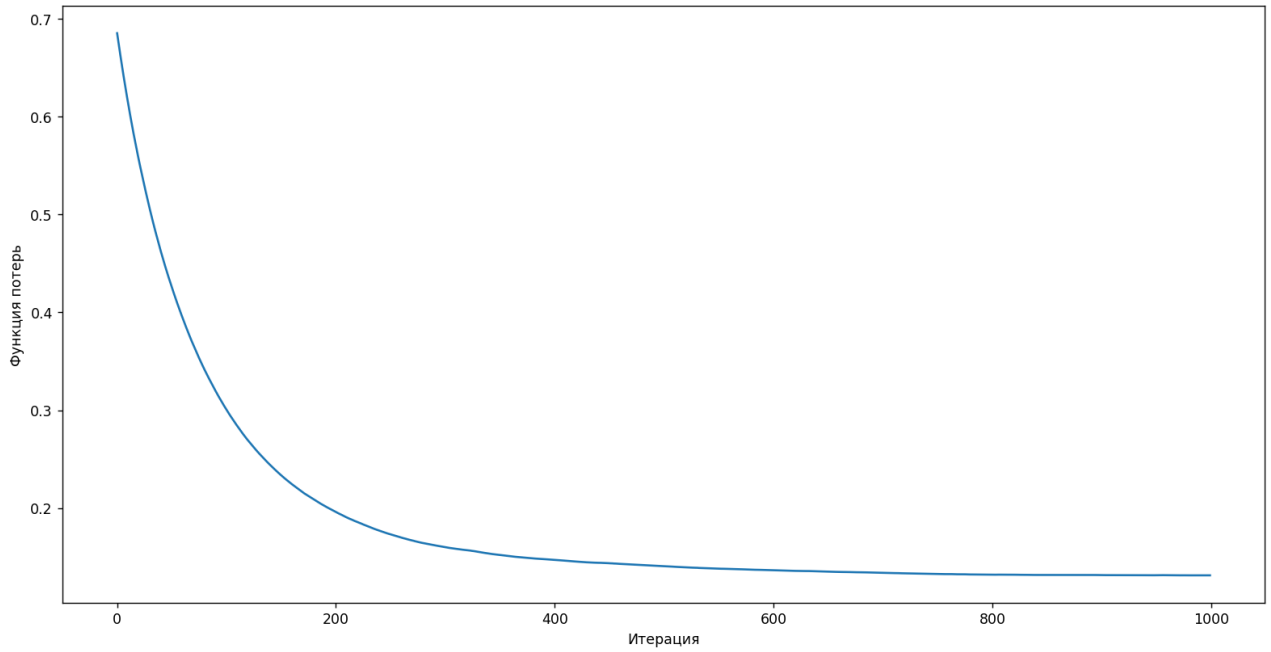

Рисунок 4.1 – Зависимость функции потерь на тестовой выборке от итерации

Видно, что график имеет гиперболическую форму и на более обученной модели становится линейным. Это говорит о том, что обучение проходит верно, функция потерь постепенно становится ниже и ближе к концу выходит на плато. Однако основным средством анализа точности является подсчет следующих

метрик: accuracy – формула (21), precision – формула (22), recall – формула (23), f-measure – формула (24).

$$\text{accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}, \quad (20)$$

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}, \quad (21)$$

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}, \quad (22)$$

$$\text{F} - \text{measure} = \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}, \quad (21)$$

где TP – объекты, которые были верно классифицированы как положительные;
TN – объекты, которые были верно классифицированы как отрицательные;
FP – объекты, которые были ложно классифицированы как положительные;
FN – объекты, которые были ложно классифицированы как отрицательные.

В таблице 2 представлено сравнение метрик, посчитанных для обученной модели, и результатов исследований, приведенных в таблице 1.

Таблица 2 – Сравнительная таблица точности работы алгоритмов

Метрики / **Алгоритмы**	**Accuracy (точность)**	**Precision (точность)**	**Recall (полнота)**	**F-measure (F-мера)**
Наивный байесовский классификатор	0.795	0.845	0.803	0.849
Метод опорных векторов	0.841	0.901	0.879	0.902
Дерево решений	0.823	0.845	0.878	0.889
Случайный лес	0.845	0.859	0.863	0.890

Метрики / Алгоритмы	Accuracy (точность)	Precision (точность)	Recall (полнота)	F-measure (F-мера)
Градиентный бустинг	0.847	**0.903**	0.883	**0.903**
Адаптивный бустинг	0.835	0.858	0.861	0.889
Разработанный метод	**0.941**	0.818	**0.936**	0.873

Видно, что разработанный метод является довольно точным, все значения метрик близки к единице и результатам существующих реализаций. Стоит отметить, что accuracy и recall превышают другие показатели. В рамках рассматриваемой задачи это является значительным преимуществом, так как лучше отнести сомнительный код к дефективным, чем пропустить и не обратить внимания пользователя на него.

4.2 Оценка времени выполнения программы

Замеры времени работы производились на ноутбуке Huawei MateBook D14. Важно отметить, что ноутбук был включен в сеть питания и нагружен только встроенными приложениями окружения и непосредственно самой системой. Технические характеристики:

— операционная система: Windows 10 (64-разрядная);
— оперативная память: 8 Гб;
— процессор: AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx 2.10 GHz;
— количество ядер: 4;
— количество логических процессоров: 8.

Для оценки времени происходит генерация функций с различным количеством строк, каждая из которых состоит из операторов и операндов. Для данной функции считаются метрики и происходит предсказание результата моделью. Для большей объективности на каждой итерации вычисляется среднее

арифметическое из 50 значений времени, посчитанных для одних и тех же данных. На рисунке 4.2 представлена зависимость времени работы от количества строк кода.

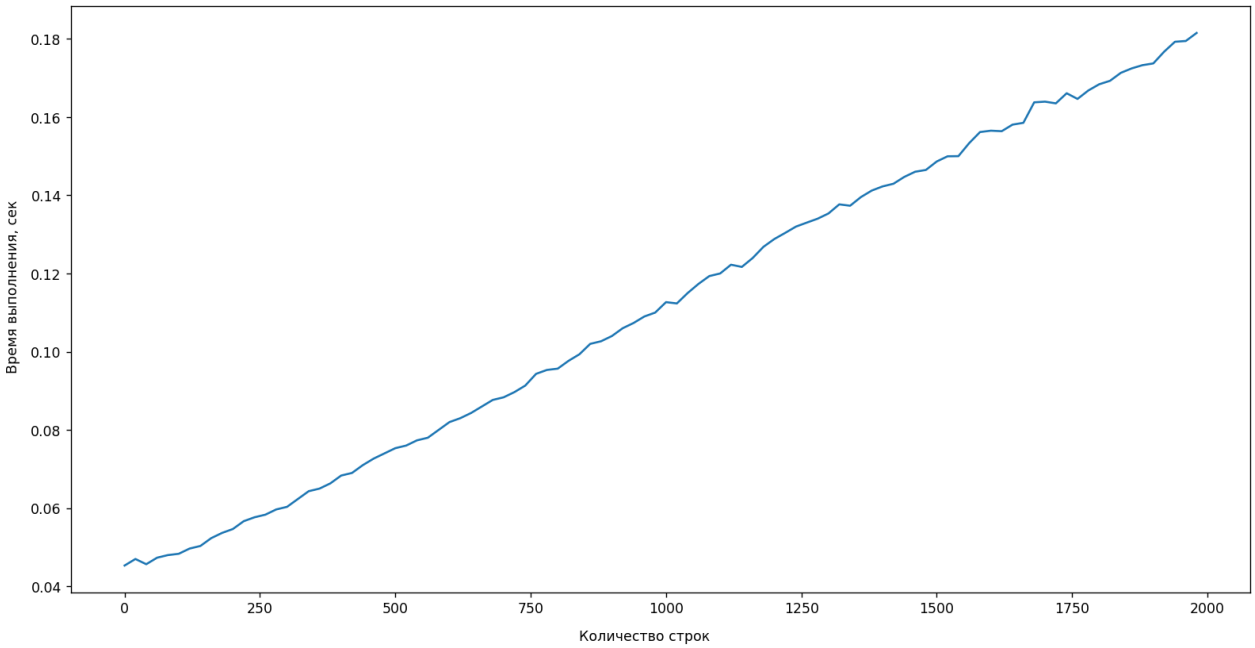

Рисунок 4.2 – Зависимость времени выполнения от количества строк

Даже для максимальных значений количества строк время имеет довольно низкие значения. График имеет линейный характер, что говорит о прямой зависимости между временем выполнения и количество строк в функции. То есть влияние объема текста программы не является критичным для разработанного метода.

4.3 Оценка разработанного ПО

Реализованная система обнаружения дефектов ПО дает возможность пользователю перед тестированием программного продукта оценить вероятность наличия дефектов в каждой функции, то есть в каких места происходит отклонение от первоначальных бизнес-требований, и начать тестирование с блоков, наиболее подверженных дефектам.

Благодаря тому, что в качестве входных параметров модели используются метрики кода, разработанный метод позволяет анализировать любые функции, размер которых до 2000 строк, независимо от сложности конструкций. Также обеспечена высокая скорость работы – время вычисления результата для максимальных объемов данных занимает менее полсекунды.

В качестве дальнейшего развития следует рассмотреть возможность детального исследования функций программы, так как на данном этапе, исходя из поставленной задачи, вероятность наличия дефектов определяется целиком для каждого блока.

Вывод

Таким образом, было проведено исследование эффективности разработанного метода: точности и времени. Сравнение с существующими реализациями показало, что разработанный метод является довольно точными. При сомнительных ситуациях предпочтение отдается в пользу наличия дефекта. Время выполнения анализа кода является довольно низким, при максимально возможном количестве строк – это меньше 200 миллисекунд. При этом зависимость времени от количества является линейной. Также проведена оценка разработанного ПО.

ЗАКЛЮЧЕНИЕ

В результате выполнения выпускной квалификационной работы была достигнута цель – выполнена разработка и программная реализация метода обнаружения дефектов ПО с использованием алгоритмов машинного обучения. При этом решены все поставленные задачи:

— проанализированы и сравнены существующие методы машинного обучения для обнаружения дефектов ПО;
— разработан метод обнаружения дефектов ПО с применением ансамбля деревьев решений (градиентного бустинга);
— разработано программное обеспечение, реализующее метод обнаружения дефектов ПО;
— проведено исследование эффективности разработанного метода и сравнение его с существующими реализациями.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Викторов, Д.С. Методика статического анализа для поиска дефектов естественной семантики программных объектов и ее программная реализация на базе инфраструктуры компилятора LLVM и фронтенда Clang [Текст] / Д.С. Викторов, Е.Н. Жидков, Р.Е. Жидков // Журнал Сибирского федерального университета. – 2018. – С. 801-810.
2. International Software Testing Qualifications Board. Стандартный глоссарий терминов, используемых в тестировании программного обеспечения. Версия 2.3 (от 9 июля 2014 года) [Текст] / ред. пер. Александр Александров. – С. 17.
3. ГОСТ Р 56920 – 2016. Системная и программная инженерия. Тестирование программного обеспечения. Часть 1. Понятия и определения. // М.: Стандартинформ, 2016. – С. 9.
4. Юхименко, Н.В. Обзор методов прогнозирования дефектов программного обеспечения [Текст] / Н.В. Юхименко, Ю.С. Белов // Программные продукты, системы и алгоритмы. – 2019. – №1 – С. 2.
5. QR Solutions. List of Defects in Software Testing | Severity & Priority in Testing [Электронный ресурс]: Режим доступа URL: https://qrsolutions.com.au/defects-in-software-testing/ (Дата обращения: 10.11.2022).
6. GeeksforGeeks. Techniques to Identify Defects [Электронный ресурс]: Режим доступа URL: https://www.geeksforgeeks.org/techniques-to-identify-defects/ (Дата обращения: 10.11.2022).
7. Logrocon Software Engineering. Что такое тестирование программного обеспечения? Определение, основы и типы [Электронный ресурс]: Режим доступа URL: https://logrocon.ru/news/testing_is (Дата обращения: 11.11.2022).
8. О развитии искусственного интеллекта в Российской Федерации [Текст]: указ Президента РФ от 10 октября 2019 г. №490 // Собр. законодательства РФ. – 2019. – 14 окт. – Ст. 5700.
9. Платонов, А.В. Машинное обучение: учебное пособие для вузов [Текст] / А.В. Платонов. // Москва: Издательство Юрайт, 2022. – 85 с.

10. Microsoft Learn. Глубокое обучение и машинное обучение в Машинном обучении Azure [Электронный ресурс]: Режим доступа URL: https://learn.microsoft.com/ru-ru/azure/machine-learning/concept-deep-learning-vs-machine-learning (Дата обращения 14.11.2022).
11. Рашка, Р. Python и машинное обучение: машинное и глубокое обучение с использованием Python, scikit-learn и TensorFlow 2. 3-е издание [Текст] / Рашка Р., Мирджалили В. // Москва: ООО «Диалектика», 2020 – 848 с.
12. Sharma, T. A Survey on Machine Learning Techniques for Source Code Analysis [Текст] / T. Sharma, M. Kechagia, S. Georgiou, R. Tiwari, I. Vats, H. Moazen, F. Sarro. – 2022. – С. 11-13.
13. Assim, A. Software Defects Prediction using Machine Learning Algorithms [Текст] / A. Assim, Q. Obeidat, M. Hammad // 2020 International Conference on Data Analytics for Business and Industry: Way Towards a Sustainable Economy (ICDABI) – 2020. – С. 1-6.
14. Shah, M. Software Defects Prediction Using Machine Learning [Текст] / M. Shah, N. Pujara – 2020. – С. 1-5.
15. Iqbal, A. Performance Analysis of Machine Learning Techniques on Software Defect Prediction using NASA Datasets [Текст] / A. Iqbal, S. Aftab, U. Ali, A. Husen // International Journal of Advanced Computer Science and Applications – 2019. – С. 1-6.
16. Aleem, S. Comparative performance analysis of machine learning techniques for software bug detection / S.Aleem, L.F. Capretz, F. Ahmed – 2015. – С. 1-9.
17. Cetiner, M. A Comparative Analysis for Machine Learning based Software Defect Prediction Systems [Текст] / M. Cetiner, O.K. Sahingoz // 11th International Conference on Computing, Communication and Networking Technologies – 2020. – С. 1-7.
18. Bhandari, G.P. Machine learning based software fault prediction utilizing source code metrics [Текст] / G.P. Bhandari, R. Gupta // IEEE 3rd International Conference on Computing, Communication and Security (ICCCS), Kathmandu (Nepal) – 2018 – С. 1-6.

19. Шитиков, В.К. Классификация, регрессия, алгоритмы Data Mining с использованием R [Текст] / В.К. Шитиков, С.Э. Мастицкий // Тольятти, 2017.
20. Федотов, Д.В. О решении задачи классификации методом опорных векторов [Текст] / Д.В. Федотов // Математические методы моделирования, управления и анализа данных – 2013 – С. 1-3.
21. Кафтанников, И.Л. Особенности применения деревьев решений в задачах классификации [Текст] / И.Л. Кафтанников, А.В. Парасич // Вестник ЮУрГУ. Серия «Компьютерные технологии, управление, радиоэлектроника» – 2015. – С. 26-32.
22. Amazon Web Services. Что такое бустинг? [Электронный ресурс]: Режим доступа URL: https://aws.amazon.com/ru/what-is/boosting/ (Дата обращения 23.11.2022).
23. Дудченко, П.В. Метрики оценки классификаторов в задачах медицинской диагностики [Текст] / П.В. Дудченко // Молодежь и современные информационные технологии: сборник трудов XVI Международной научно-практической конференции студентов, аспирантов и молодых ученых – 2019. – С. 164-165.
24. NASA. Promise software engineering repository [Электронный ресурс]: Режим доступа URL: http://promise.site.uottawa.ca/SERepository/datasets-page.html (Дата обращения: 29.11.2022).
25. Python 3.11.3 documentation [Электронный ресурс]. Режим доступа URL: https://docs.python.org/3/ (Дата обращения 20.05.2023).
26. Scikit-learn. Machine Learning in Python [Электронный ресурс]. Режим доступа URL: https://scikit-learn.org/stable/ (Дата обращения 20.05.2023).
27. XGBoost [Электронный ресурс]. Режим доступа URL: https://xgboost.readthedocs.io/en/stable/ (Дата обращения 20.05.2023).
28. Qt for Python [Электронный ресурс]. Режим доступа URL: https://doc.qt.io/qtforpython/ (Дата обращения 20.05.2023).
29. Qt Designer Manual [Электронный ресурс]. Режим доступа URL: https://doc.qt.io/qt-5/qtdesigner-manual.html (Дата обращения: 20.05.2023).

ПРИЛОЖЕНИЕ А

Презентация