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 ране написанного кода позволят дать вероятностную оценку нахождения дефекта в том или ином месте программы.

Цель данной работы: провести сравнение методов машинного обучения для обнаружения дефектов ПО.

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

- представить обзор дефектов разрабатываемого ПО;
- классифицировать методы для обнаружения дефектов ПО;
- рассмотреть возможность использования методов машинного обучения в данной сфере;
- сформулировать параметры сравнения методов машинного обучения для обнаружения дефектов ПО;
- провести обзор и сравнение существующих методов машинного обучения для обнаружения дефектов ПО;
- описать формализованную постановку задачи в виде диаграммы в нотации IDEF0.

1 Дефекты разрабатываемого ПО

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, бизнес или безопасность пользователей или 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 (blocker)* – ошибка, которая приводит программу в нерабочее состояние.
2. *Критический дефект (critical)*, приводящий некоторый ключевой функционал в нерабочее состояние? существенное отклонение от бизнес-логики, неправильная реализация требуемых функций и т.д.

3. *Серьезная ошибка (major)*, свидетельствующая об отклонении от бизнес-логики или нарушающая работу программы (не имеет критического воздействия на приложение).
4. *Незначительный дефект (minor)*, не нарушающий функционал тестируемого приложения, но который является несоответствием ожидаемому результату.
5. *Тривиальный дефект (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 (high)* – должен быть исправлен как можно быстрее, т.к. критически влияет на работоспособность программы.
2. *Со средним приоритетом (medium)* – дефект должен быть обязательно исправлен, но он не оказывает критическое воздействие на работу.
3. *С низким приоритетом (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.

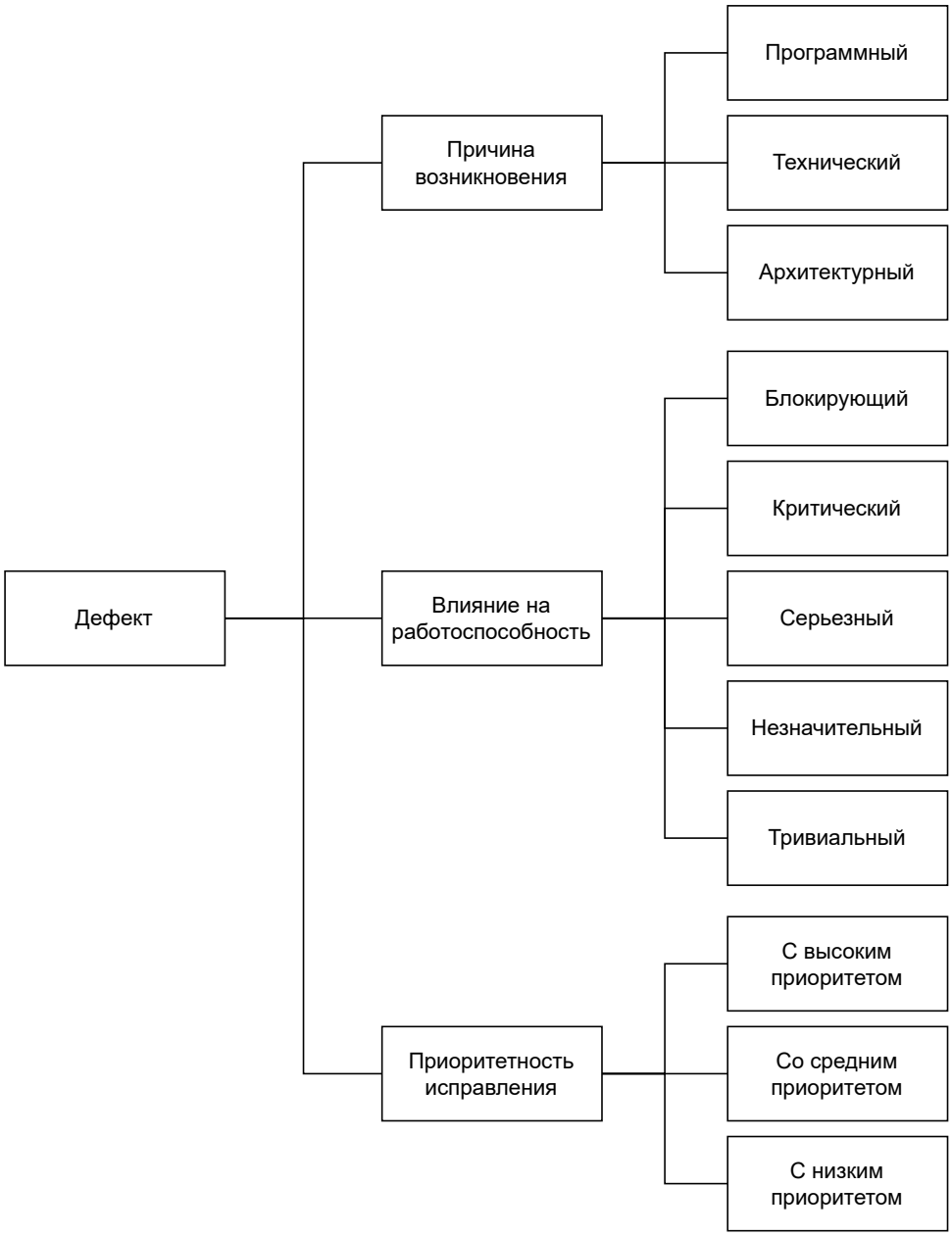

Рисунок 1.1 – Классификация дефектов ПО

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].

1.2 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. Иными словами – функциональное тестирование, которое подразделяется на:
        - модульное – используется для тестирования отдельно взятого модуля программы (помогает разработчикам узнать, правильно ли работает каждый блок кода в изоляции от остальных);
        - интеграционное – позволяет убедиться, что при взаимодействии интегрированные блоки работают без ошибок;
        - системное – программное обеспечение тестируется как единое целое, проверяется функциональность, безопасность и переносимость;

        - регрессионное – проверка ранее протестированной программы, позволяющая убедиться, что внесенные изменения не повлекли за собой появления дефектов в той части программы, которая не менялась;
        - приемочное – комплексное тестирование, необходимое для определения уровня готовности системы к последующей эксплуатации;
        - дымовое (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 объединения 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].

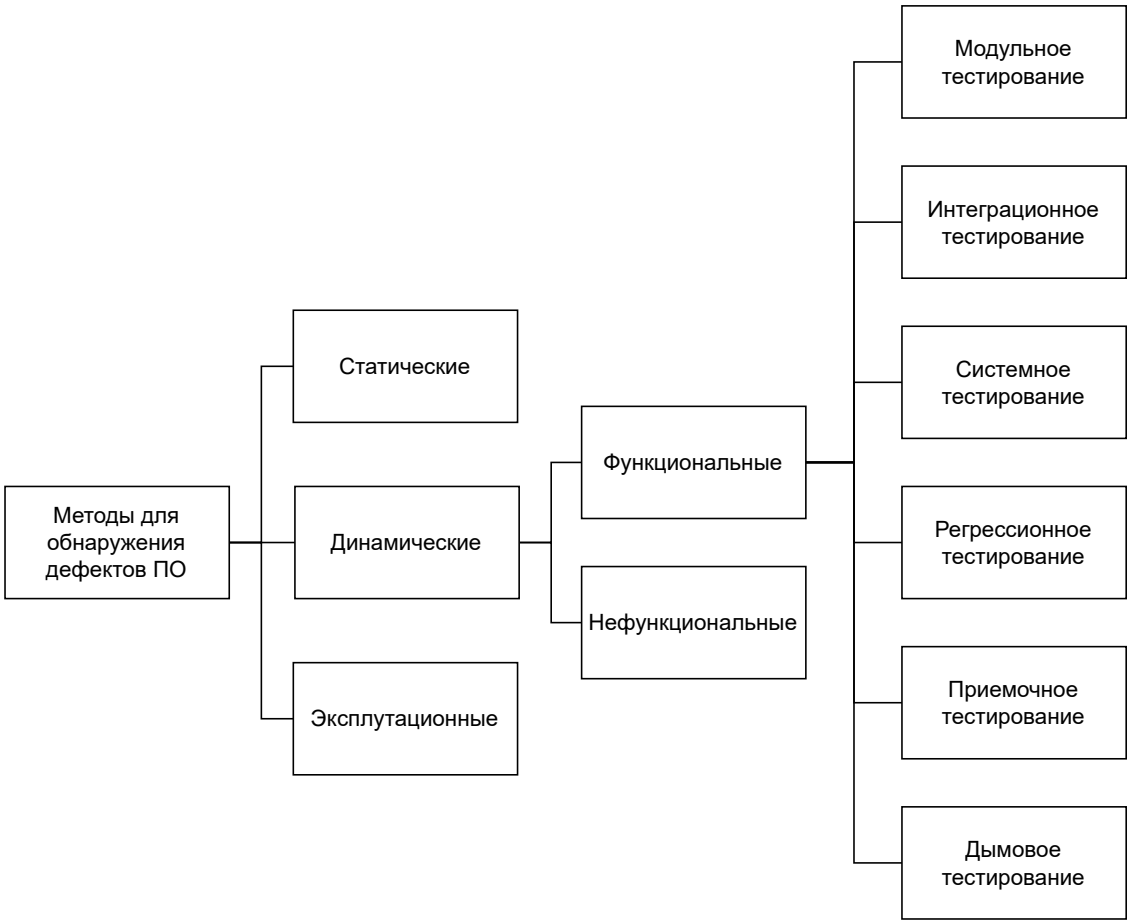

Рисунок 1.2 – Классификация методов для обнаружения дефектов ПО

2 Машинное обучение

2.1 Понятие машинного обучения

Искусственный интеллект – это методика, которая позволяет компьютерам имитировать человеческий интеллект. **Машинное обучение** – это наука о разработке алгоритмов и статистических моделей, которые компьютерные системы используют для выполнения задач без явных инструкций, полагаясь вместо этого на шаблоны и логические выводы [8]. Со второй половины XX века машинное обучение развивается как подобласть искусственного
интеллекта, применяющая алгоритмы, которые выводят знания из данных с целью вырабатывания прогнозов. То есть вместо ручного вывода правил и построения моделей путем анализа крупных данных, машинное обучение предлагает более эффективную альтернативу [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 2.1.

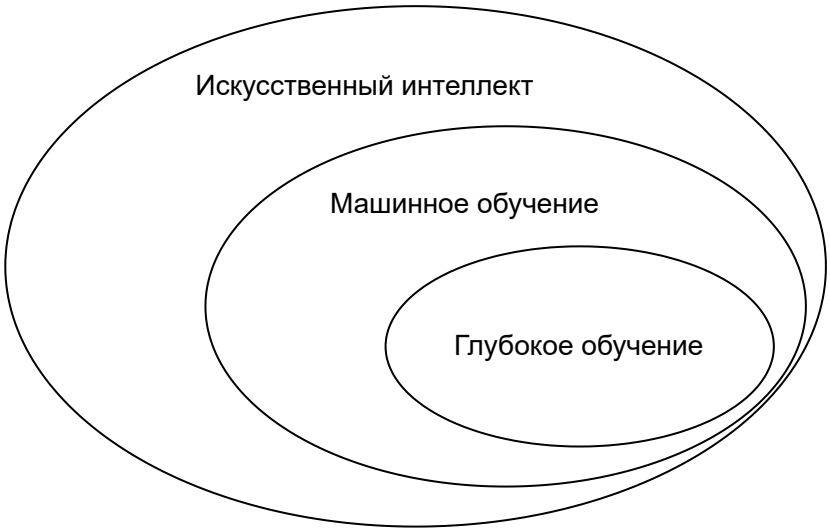

Рисунок 2.1 – Отношение машинного обучения к искусственному интеллекту и глубокому обучению

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 [9].

2.2 Типы машинного обучения

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

2.2.1 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 2.2 представлена последовательность
действий при обучении с учителем.

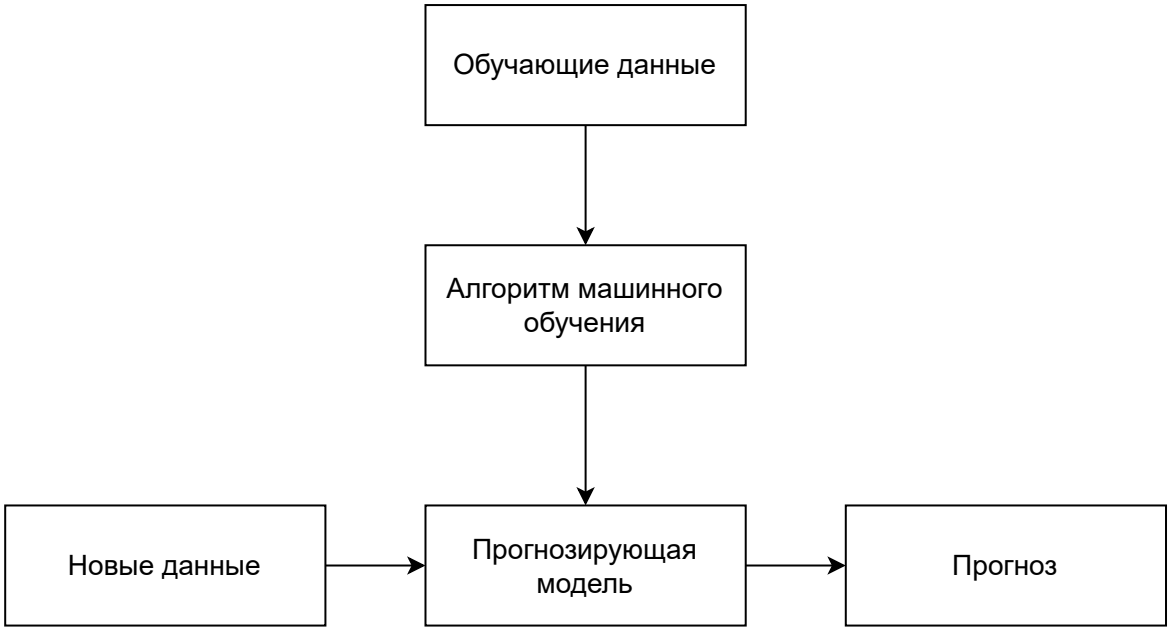

Рисунок 2.2 – Последовательность действий при обучении с учителем

К основным задачам, решаемым при помощи обучения с учителем, относят задачи классификации и регрессии [9].

2.2.2 Обучение с подкреплением

Целью такого обучения является разработка системы (агента), которая улучшает свои характеристики на основе взаимодействий со средой. На рисунке 2.3 представлен концепция обучения с подкреплением.

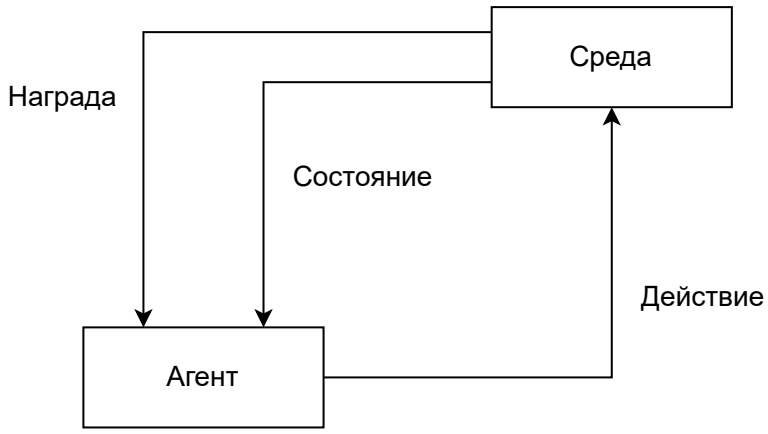

Рисунок 2.3 – Концепция обучения с подкреплением

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 [9].

2.2.3 Обучение 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 [9].

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 [11]. 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 2.4 представлена общая схема процесса обучения модели обнаружения дефектов ПО.

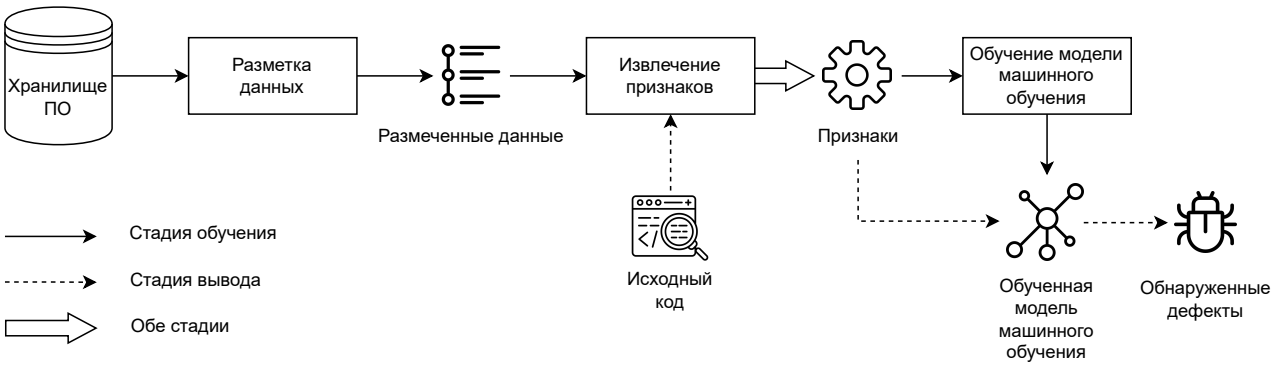

Рисунок 2.4 – Процесс обучения модели обнаружения дефектов ПО

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, которые затем передаются в модель машинного обучения для обучения. Наконец, обученная модель может классифицировать различные фрагменты кода как ошибочные или безопасные на основе закодированных знаний [11].

3 Методы машинного обучения для обнаружения дефектов ПО

3.1 Существующие методы

Для обучения моделей во многих статьях [12-17] используются классические алгоритмы машинного обучения такие, как дерево решений, алгоритм случайного леса, градиентный бустинг, метод опорных векторов, наивный байесовский классификатор [11], решающие задачу классификации. Приведем описание каждого из них.

3.1.1 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: $\max\limits_{c} P(c_j|d)$. Для рассматриваемой задачи имеются два класса в зависимости, есть дефект в программе или или нет.

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

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 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:

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

где $P(x_i|c_j)$ – вероятностная оценка вклада признака x_i в то, что $d \in c_j$.

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:

$$c* = \underset{c_j \in C}{\operatorname{argmax}} [\log P(c_j) + \sum_{i=1}^{X} P(x_i|c_j)] \tag{3}$$

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 [18].

3.1.2 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 [19].

3.1.3 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 $X = \{x_1, \ldots, x_L\}$ и алгоритмов $A = \{a_1, \ldots, a_D\}$ 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 $\nu(a, X) = \frac{n(a,x)}{|X|}$ 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 [20].

3.1.4 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.

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

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

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

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 [20].

3.1.5 Бустинг

Бустинг – модификация алгоритма случайного леса, обучение происходит путем последовательного обучении 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 [21].

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 $w_1^1, \ldots, w_m^1$. 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 [20].
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 [21].

3.2Параметры сравнения методов

Как правило, результаты классификации помечаются как положительные (имеются дефекты) и отрицательные (дефекты отсутствуют). Для оценки качества работы полученных моделей используются различные метрики. Так, в статьях [13-17] 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** – 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.

$$\text{accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{6}$$

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

$$\text{точность} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{7}$$

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

$$\text{полнота} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{8}$$

4. **F-мера** (F-measure) – взвешенное гармоническое среднее полноты и точности (формула 9). Этот показатель демонстрирует, как много объектов классифицируется моделью правильно, и сколько истинных экземпляров она не пропустит [22].

$$\text{F-мера} = 2 \cdot \frac{\text{точность} \cdot \text{полнота}}{\text{точность} + \text{полнота}} \tag{9}$$

3.3 Сравнение методов

Для сравнения рассмотренных методов использованы результаты, описанные в статьях [13-17]. Модели обучались на данных, представленных в репозитории NASA [23]. В таблице 3.1 отображены средние арифметические значения для каждой из метрик по всем наборам данных.

Алгоритмы классификации дефектов	Accuracy	Точность	Полнота	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
Алгоритм случайного леса	**0.847**	**0.903**	**0.883**	**0.903**
Градиентный бустинг	0.845	0.859	0.863	0.890
Адаптивный бустинг	0.835	0.858	0.861	0.889

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

4 Формализованная постановка задачи

Для дальнейшего продолжения работы необходимо формализовать задачу. Наиболее удобным способом для этого является диаграмма в нотации IDEF0. Поставленная задача представлена на рисунке 4.1. На вход программе подается исходный код, который под воздействием механизмов и блока управления, анализируется, и программа возвращает модули, в которых имеются дефекты ПО.

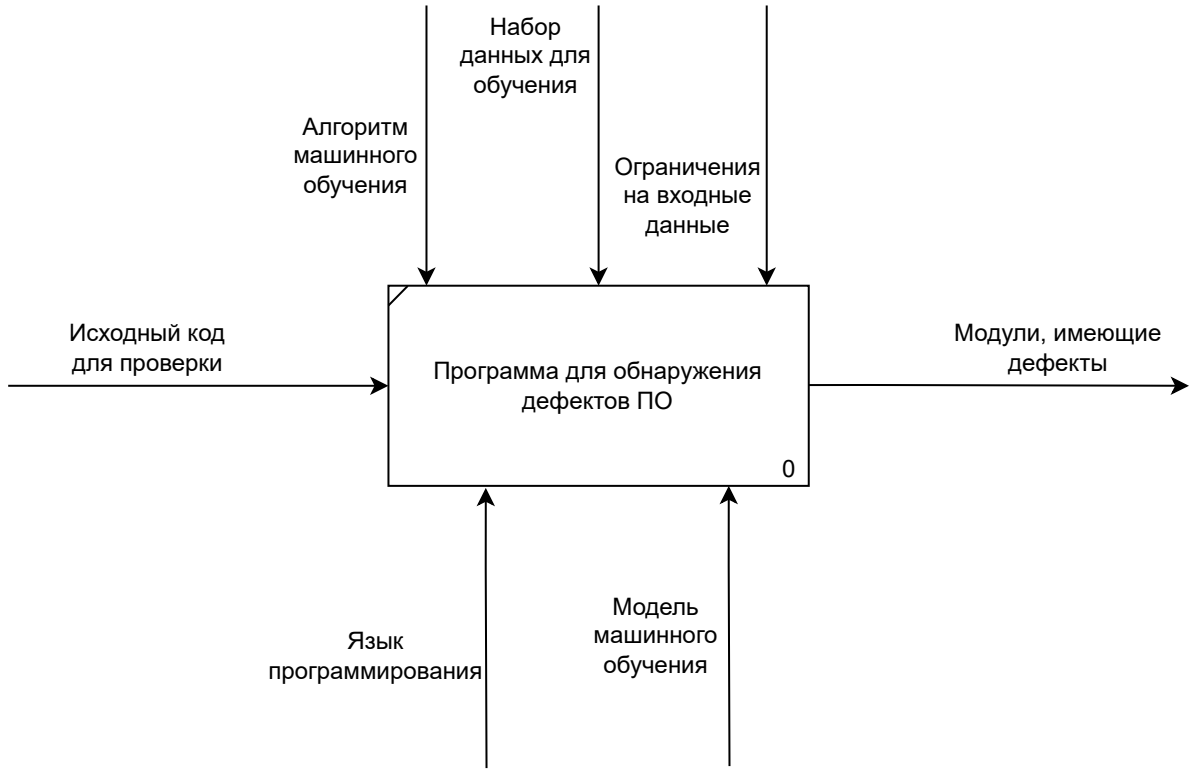

Рисунок 4.1 – Формализованная постановка задачи

Заключение

В результате выполнения работы достигнута цель – проведено сравнение методов машинного обучения для обнаружения дефектов ПО. Также решены все поставленные задачи:

- представлен обзор дефектов разрабатываемого ПО;
- классифицированы методы для обнаружения дефектов ПО;
- рассмотрена возможность использования методов машинного обучения в данной сфере;
- сформулированы параметры сравнения методов машинного обучения для обнаружения дефектов ПО;
- проведен обзор и сравнение существующих методов машинного обучения для обнаружения дефектов ПО;
- описана формализованная постановка задачи в виде диаграммы в нотации IDEF0.