DOI: 10.15514/ISPRAS-2025-37(6)-6

Предсказание истинности предупреждений промышленного статического анализатора с использованием методов машинного обучения

[1,3] *У.В. Тяжкороб, ORCID: 0000-0002-2375-6842 <tsiazhkorob@ispras.ru>*
[1,2] *М. В. Беляев, ORCID: 0000-0003-3489-3508 <mbelyaev@ispras.ru>*
[1,2] *А.А. Белеванцев, ORCID:0000-0003-2817-0397 <abel@ispras.ru>*
[1,2] *В. Н. Игнатьев, ORCID: 0000-0003-3192-1390 <valery.ignatyev@ispras.ru>*

[1] *Институт системного программирования им. В. П. Иванникова РАН, Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.*

[2] *Московский государственный университет имени М. В. Ломоносова, Россия, 119991, Москва, Ленинские горы, д. 1.*

[3] *Московский физико-технический институт, 141701, Россия, Московская область, г. Долгопрудный, Институтский переулок, д. 9.*

Аннотация. В данной работе описан механизм автоматической классификации предупреждений статического анализа с использованием методов машинного обучения. Статический анализ является инструментом поиска потенциальных уязвимостей и ошибок в исходном коде. Однако зачастую статические анализаторы генерируют большое количество предупреждений, причем как истинных, так и ложных. Вручную проанализировать все найденные анализатором дефекты является трудоемкой и времязатратной задачей. Разработанный механизм автоматической классификации показал высокую точность более 93% при полноте около 96% на наборе предупреждений, сгенерированных промышленным инструментом статического анализа Svace при анализе реальных проектов. Генерация набора данных для модели машинного обучения основана на предупреждениях и метриках исходного кода, полученных в процессе анализа проекта статическим анализатором. В работе рассматриваются различные подходы к отбору и обработке признаков классификатора с учетом различных особенностей рассматриваемых алгоритмов машинного обучения. Эффективность работы механизма и его независимость от языка программирования позволили добавить его в промышленный инструмент статического анализа Svace. Были рассмотрены различные подходы к интеграции инструмента, учитывающие специфику статического анализатора, и выбран наилучший из них.

Ключевые слова: машинное обучение; статический анализ; классификация; метрики исходного кода; предупреждения.

Для цитирования: Тяжкороб У.В., Беляев М. В., Белеванцев А.А., Игнатьев В.Н. Предсказание истинности предупреждений промышленного статического анализатора с использованием методов машинного обучения. Труды ИСП РАН, том 37, выпи. 6, часть 1, 2025 г., стр. 101–120. DOI: 10.15514/ISPRAS–2025–37(6)–6.

Machine learning-based validation of warnings in an industrial static code analyzer

[1,3] *U.V. Tsiazhkorob, ORCID:0000-0002-2375-6842 <tsiazhkorob@ispras.ru>*
[1,2] *M.V. Belyaev, ORCID: 0000-0003-3489-3508 <mbelyaev@ispras.ru>*
[1,2] *A.A. Belevantsev, ORCID: 0000-0003-2817-0397 <abel@ispras.ru>*
[1,2] *V.N. Ignatiev, ORCID: 0000-0003-3192-1390 <valery.ignatyev@ispras.ru>*

[1] *Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.*

[2] *Lomonosov Moscow State University, 1, Leninskie Gory, Moscow, 119991, Russia.*

[3] *Moscow Institute of Physics and Technology, 9, Institutskiy per., Dolgoprudny, Moscow Region, 141701, Russia.*

Abstract. This paper describes a mechanism for the automatic classification of static analysis warnings using machine learning methods. Static analysis is a tool for detecting potential vulnerabilities and bugs in source code. However, static analyzers often generate a large number of warnings, including both true and false positives. Manually analyzing all the defects found by the analyzer is a labor-intensive and time-consuming task. The developed automatic classification mechanism demonstrated high precision of more than 93% with a recall of about 96% on a set of warnings generated by the industrial static analysis tool Svace during the analysis of real-world projects. The dataset for the machine learning model is generated based on the warnings and source code metrics obtained during the static analysis of the project. The paper explores various approaches to feature selection and processing for the classifier, taking into account the characteristics of different machine learning algorithms. The mechanism's efficiency and its independence from the programming language allowed it to be integrated into the industrial static analysis tool Svace. Various approaches to integrating the tool were considered, accounting for the specifics of the static analyzer, and the most convenient one was selected.

Keywords: machine learning; static analysis; classification; source code metrics; warnings.

For citation: Tsiazhkorob U.V., Belyaev M.V., Belevantsev A.A., Ignatiev V.N. Machine learning-based validation of warnings in an industrial static code analyzer. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 6, part 1 , 2025, pp. 101-120 (in Russian). DOI: 10.15514/ISPRAS-2025-37(6)-6.

1. Введение

Использование статического анализа исходного кода в промышленной разработке становится стандартной практикой. Это позволяет значительно повысить качество и безопасность программного обеспечения, снижая затраты на поддержку создаваемых программных продуктов. Однако статические анализаторы кода могут генерировать как истинные, так и ложные предупреждения. По этой причине для оценки качества работы статического анализатора принято использовать точность – долю истинных предупреждений среди всех, выданных анализатором. Точность популярных статических анализаторов варьируется от 50% до 100%, в зависимости от типа искомой уязвимости. Современные статические анализаторы способны выявлять сотни различных типов уязвимостей и ошибок в коде, что приводит к генерации большого количества предупреждений для крупных проектов (в среднем порядка 10 предупреждений на 1000 строк исходного кода). Поскольку сгенерированные предупреждения могут быть как истинными, так и ложными, необходимо производить анализ результатов, что чаще всего разработчикам приходится делать вручную. Это значительно усложняет и замедляет процесс разработки программного обеспечения. Таким образом, актуальность разработки механизма автоматической классификации предупреждений на истинные и ложные обусловлена потребностью в повышении эффективности разработки программного обеспечения и отсутствием существующих

масштабируемых, проверенных и интегрированных в промышленные статические анализаторы решений. Возможность автоматической классификации предупреждений статического анализа уже была показана в работе [1]. Однако во время и после интеграции данного механизма в статический анализатор **Svace** [2] были выявлены особенности статического анализатора и анализируемых языков программирования, учитывать которые необходимо для корректной классификации предупреждений.

Целью данной работы является доработка и повышение точности механизма автоматической классификации выданных статическим анализатором предупреждений с использованием методов машинного обучения и его интеграция в промышленный инструмент статического анализа **Svace**.

Для достижения поставленной цели необходимо выполнить следующие задачи:

- найти и проанализировать существующие проблемы в работе механизма;
- доработать схему интеграции со статическим анализатором **Svace**, учитывая особенности работы анализатора и различия в анализе разных языков программирования;
- повысить качество и эффективность классификации с помощью отбора признаков и выбора наилучшего алгоритма машинного обучения;
- рассмотреть различные подходы к интеграции механизма в **Svace** с поддержкой различных сценариев использования и реализовать наиболее оптимальный из них.

Ключевым требованием к работе механизма классификации является повышение точности результатов работы статического анализатора без потерь в полноте.

- поддержка форматов представления данных, полученных в процессе работы статического анализатора;
- эффективная обработка больших объемов данных, включая десятки тысяч предупреждений и сотни различных метрик, собранных из миллионов строк кода;
- обработка результатов работы анализатора с учетом особенностей их структуры и различий для разных языков программирования;
- возможность дообучения модели на пользовательских данных;
- компактное хранение модели, позволяющее включить её в состав дистрибутива статического анализатора.

2. Существующие решения

Для автоматической классификации предупреждений, полученных от статических анализаторов, существует ряд решений и исследований.

Одно из них основано на использовании сверточных нейронных сетей (CNN) [3]. В этом решении классификатор использует фрагменты исходного кода в качестве признаков модели машинного обучения. Средняя точность решения достигает 79.72%. Однако представлены результаты только для шести детекторов, для которых удалось выявить лексические шаблоны исходного кода, ведь для большинства дефектов недостаточно локального контекста для определения истинности. Более того, характерной особенностью моделей на основе сверточных нейронных сетей является длительное время обучения (порядка нескольких часов) даже при использовании графического ускорителя.

Также существуют исследования по применимости машинного обучения для классификации предупреждений статических анализаторов [4]. В данной работе рассматривается применение алгоритмов машинного обучения, таких как метод опорных векторов (SVM) [5], K-ближайших соседей (KNN) [6], случайный лес (Random Forest) [7, 8] и алгоритм RIPPER

[9], для классификации предупреждений статических анализаторов. Наилучшую точность показал алгоритм случайного леса (83%–98%).

В качестве признаков для моделей машинного обучения в данной работе используются 111 метрик исходного кода, рассчитываемые с помощью механизма Understand [10]. Тестирование производилось только для 7 типов ошибок из перечня общих дефектов и уязвимостей CWE [11]. В качестве набора предупреждений использовались результаты работы статического анализатора на синтетически сгенерированной выборке Juliet для C++ [12].

Также существует решение с инкрементальным механизмом машинного обучения. Тестирование производилось на 9 проектах с открытым исходным кодом на языке программирования Java. В качестве признаков использовались характеристики исходного кода и предупреждений, полученные с помощью статического анализа кода, метаданных проекта и истории версий. Были исследованы три модели машинного обучения, при инкрементальном активном обучении наилучшую точность показал метод опорных векторов (около 90%). Однако данное решение плохо масштабируемо и ресурсозатратно, поскольку для каждого нового проекта необходимо снова запускать активное обучение, и требуется контроль качества на каждом этапе обучения модели.

Существующие исследования отражают лишь теоретическую возможность использования методов машинного обучения для верификации результатов работы статических анализаторов, а существующие решения имеют ряд недостатков и ограничений. Однако с помощью анализа существующих решений можно сделать выводы о важности определенных признаков и качестве определенных моделей машинного обучения, что позволяет проводить исследования, опираясь на наилучшие практики.

В данной работе представлен подход к автоматической классификации предупреждений, учитывающий все недостатки существующих решений. Механизм не зависит от синтаксиса языка программирования и типа дефекта, классификация производится за малое по сравнению со статическим анализом время. При тестировании на реальных проектах с открытым исходным кодом была достигнута высокая точность, составляющая 92%.

3. Разработанный механизм

Разработанный алгоритм автоматической классификации предупреждений состоит из 4 этапов:

- генерация признаков: создание и обработка признаков для модели машинного обучения на основе предупреждений и метрик исходного кода;
- обучение модели: подбор гиперпараметров модели и обучение на предоставленном наборе данных;
- классификация предупреждений статического анализатора: модель предсказывает вероятность истинности предупреждений;
- отображение или фильтрация: результаты классификации отображаются в пользовательском интерфейсе или используются для фильтрации ложных предупреждений.

В качестве признаков для модели машинного обучения используются метрики исходного кода. *Метриками* [13] исходного кода называются числовые значения, характеризующие качество исходного кода и широко применяемые для обеспечения качества программного обеспечения (QA) [14], например, цикломатическая сложность, то есть количество линейно независимых путей в коде. Метрики вычисляются отдельно для каждой директории, файла, класса и метода, которые в работе будем называть *сущностями*. Вычисление метрик исходного кода в **Svace** осуществляется компонентом **SCRA** (**Source Code Relation Analyzer**) [15]. В результате работы **Svace** генерируются файл с информацией о метриках и

файл с информацией о сущностях. Для каждой сущности анализируемого проекта статический анализатор сохраняет имя, числовой уникальный идентификатор и уникальный идентификатор родительской сущности. Для каждой метрики исходного кода сохраняется тип, значение и уникальный идентификатор сущности, к которой относится метрика. Всего **Svace** рассчитывает значения для 141 типа метрик.

Рассмотрим особенности некоторых стадий работы механизма автоматической классификации предупреждений.

3.1 Особенности реализации механизма генерации признаков

Реализация алгоритма сопоставления предупреждения с его уникальным набором значений метрик должна учитывать особенности хранения результатов работы **Svace**. Основная сложность состоит в том, чтобы определить, какие именно значения метрик соответствуют конкретному предупреждению. Поэтому, сперва необходимо определить, какие сущности относятся к предупреждению по полному пути к предупреждению и сигнатуре функции. После этого становится возможным по уникальному идентификатору сущности найти все соответствующие ей значения метрик, тем самым сопоставив предупреждения со специфичным для него набором метрик исходного кода.

3.1.1 Обобщенный алгоритм

Схема алгоритма генерации признаков представлена на рис. 1.

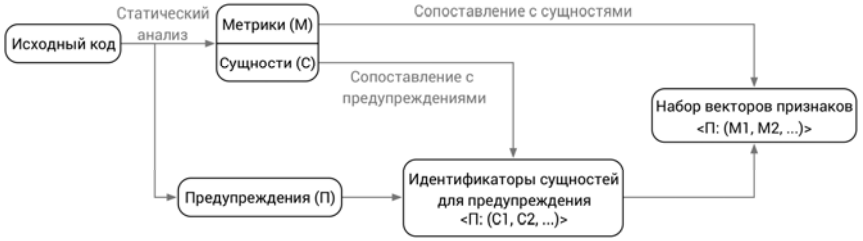

Рис. 1. Схема алгоритма генерации признаков.
Fig. 1. Feature generation scheme.

После составления списка сущностей, относящихся к предупреждению, происходит поиск значений метрик для каждой сущности из списка. Как известно, в полном пути к файлу может быть несколько директорий, аналогично, в сигнатуре функции может быть несколько классов. В вектор признаков записываются метрики, вычисленные для наиболее глубоких сущностей каждого типа. То есть из всех директорий метрики записываются для той, в которой непосредственно находится файл. Помимо метрик, в файл с признаками записывается также тип найденного дефекта, статус предупреждения, язык программирования анализируемого проекта и порядковый номер предупреждения (для обратного сопоставления признаков с исходными предупреждениями).

Для ускорения составления списков идентификаторов сущностей используется кэширование: при полном или частичном совпадении искомого пути или сигнатуры функции с предыдущим предупреждением нет необходимости искать все сущности заново. Для использования кэширования после прочтения файла с предупреждениями информация о них сортируется по имени файла. Если имена файлов одинаковые, то сортировка происходит по имени функции.

3.1.2 Особенности реализации

В процессе разработки и работы механизма генерации признаков были выявлены особенности хранения результатов работы статического анализатора.

Для того, чтобы получить набор имен сущностей из полного пути к файлу и сигнатуры функции, нужно разделить их строковые представления по соответствующим символам-разделителям. Кроме того, для C++ имена функций являются декорированными (*mangled*), и чтобы получить исходные имена функций, их необходимо восстановить при помощи декодера.

Еще одной проблемой является невозможность однозначно определить сущность только по ее имени, поскольку несколько сущностей могут иметь одинаковые значения имен. Информация о сущностях имеет иерархическую структуру: проекты с точки зрения папок и файлов, исходный код – с точки зрения классов и функций. В качестве решения данной проблемы предлагается искать имена сущностей последовательно с учетом предыдущей сущности как родителя текущей. Чтобы поиск был наиболее эффективным, строится дерево сущностей.

Однако и это решение не гарантирует корректную работу поиска сущностей, поскольку одинаковые имена могут иметь и сущности с одним и тем же родителем, например, в случае с разделяемыми (`partial`) классами на языке C# или для языков C++ и C# при многократной компиляции файла с различными значениями символов условной компиляции. Для решения данной проблемы предложено добавить проход по дереву сущностей после его построения, в процессе которого составляется общий список потомков для сущностей с одинаковыми идентификаторами и именами. Чтобы сохранить исходную иерархию, указатели на родительский узел у всех потомков остаются прежними.

Из-за различия в форматах записи сигнатур функций с вложенными классами для предупреждения (`Outer.Inner.Func()`) и соответствующих им сущностей в файле с информацией о всех сущностях (`Outer` → `Outer$Inner` → `Func`) после построения дерева сущностей необходимо совершать еще один проход для устранения различий.

Описанный инструмент был реализован на языке C++ для уменьшения затрат ресурсов и времени.

3.2 Обработка данных и выбор модели

Механизм классификации реализован в виде программы на языке Python, использующей популярные библиотеки машинного обучения и анализа данных **pandas** [16] и **scikit-learn** [17], а также библиотеку **Beautiful Soup** [18] для работы с данными в формате XML.

3.2.1 Добавление и отбор признаков

В сгенерированном наборе векторов признаков значения типа дефекта и статуса предупреждения являются строковыми. Статус предупреждения кодируется следующим образом: истинное предупреждение кодируется значением 1, ложное предупреждение – значением 0. Тип дефекта определяется уникальным строковым значением. Однако к этим значениям могут быть добавлены строковые значения тегов в любом количестве и в любом порядке. Например, тег `.TEST` в типе предупреждения `HANDLE_LEAK.EXCEPTION.TEST` показывает, что дефект типа `HANDLE_LEAK` найден в коде тестов; в то время как тег `.RET` в типе предупреждения `DEREF_OF_NULL.RET.LIB.PROC` означает, что найденный дефект типа `DEREF_OF_NULL` относится к возвращаемому из функции значению. Список тегов является общим для всех типов дефектов. Список тегов является общим для всех типов дефектов. Поскольку каждое предупреждение имеет ровно один тип дефекта, но может иметь произвольное количество тегов, тип предупреждения кодируется с помощью инструмента `LabelEncoder`, а для тегов применяется тип кодирования One-Hot с помощью инструмента

`OneHotEncoder` библиотеки **scikit-learn**. То есть тип дефекта кодируется уникальным числовым значением, в то время как для каждого тега добавляется свой признак со значением 1, если такой тег добавлен к типу дефекта, и 0, если нет.

Для того, чтобы алгоритм машинного обучения лучше понимал структуру данных и закономерности в них, в качестве признаков добавляются исходная точность анализатора по истинным предупреждениям и общее количество предупреждений в файле и в функции. Исходная точность рассчитывается на наборе данных для обучения как доля истинных предупреждений от всего количества предупреждений, а количество предупреждений в файлах и в функциях рассчитывается на полном наборе данных. Создание новых признаков, специфичных для данной задачи, значительно повышает информативность данных. Общее количество признаков, включающих в себя метрики исходного кода, тип дефекта, количество предупреждений в файле и в функции и исходную точность детектора, равно 163.

При анализе накопленной дисперсии осей проекции (рис. 2) было выявлено, что при количестве признаков больше 100 этот показатель перестает расти. Поскольку накопленная дисперсия осей проекции показывает, насколько хорошо описывается изначальное распределение величин в зависимости от количества признаков, то при уменьшении размерности исходных признаков до 100 исходные зависимости не будут потеряны.

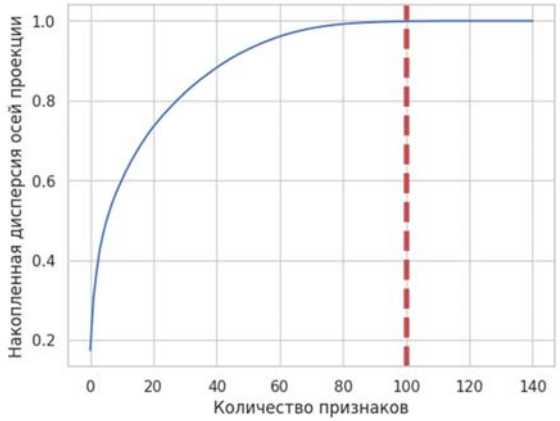

Рис. 2. Накопленная дисперсия осей проекции.
Fig. 2. Cumulative explained variance.

Отбор признаков производить важно, поскольку большое количество признаков может привести к переобучению: чем больше признаков, тем проще модели найти ложные зависимости в данных. Более того, при большей размерности пространства признаков увеличивается время обучения и время классификации, особенно для алгоритмов машинного обучения, использующих деревья, а также алгоритмов, основанных на градиентном бустинге.

3.2.2 Подготовка наборов данных

После создания набора векторов признаков для предупреждений, для обучения модели необходимо разделить набор данных на тренировочный, валидационный, и, в случае тестирования, тестовый. Обычно тренировочный набор данных составляет 80% от всех векторов, а валидационный, соответственно, 20%. В случае, когда набор данных делится на три части, тренировочный набор данных составляет 56%, валидационный – 14% и тестовый – 30%. Чтобы в тренировочный набор данных попали предупреждения для всех типов дефектов, разделение исходного набора данных производится отдельно для каждого типа

дефекта. Затем полученные данные объединяются в общий тренировочный, валидационный и, в случае тестирования, тестовый набор. Исходный набор данных является несбалансированным по количеству ложных и истинных предупреждений: количество объектов класса 0 в несколько раз меньше количества объектов класса 1. Чтобы сохранить отношения количества объектов класса 0 к количеству объектов класса 1 как во всех наборах данных, разделение исходного набора данных производится со стратификацией, то есть с балансировкой. Функция `train_test_split` библиотеки **scikit-learn**, которая является механизмом разбиения исходного набора данных, имеет параметр `stratify`, значением которого является признак для стратификации, в данном случае это целевое значение – статус предупреждения.

Важным этапом подготовки данных для модели машинного обучения является масштабирование – приведение распределения значений векторов признаков к среднему, равному 0, и стандартному отклонению, равному 1. Необходимость масштабирования обусловлена тем, что данные, разные по физическому смыслу, сильно отличаются по абсолютным значениям. Дисбаланс между значениями признаков негативно влияет на построение модели, приводя к смещенным результатам предсказаний, ошибкам классификации и, соответственно, снижению точности.

3.2.3 Анализ важности признаков для моделей машинного обучения

В качестве моделей машинного обучения рассматриваются классификаторы **CatBoost** [19], **Random Forest** [20] и **XGBoost** [21]. **CatBoost** и **XGBoost** используют алгоритмы градиентного бустинга. **CatBoost** является алгоритмом машинного обучения с высокой производительностью и использует разнообразные стратегии регуляризации для предотвращения переобучения.

Для уже обученных моделей с настроенными гиперпараметрами (настройка подробно описывается в статье [1]) по отдельности был произведен анализ важности признаков с помощью атрибута `feature_importances_`, рассчитываемого автоматически методами использованных библиотек после обучения модели. Первые 15 важных признаков для каждой модели представлены на графиках (рис. 3).

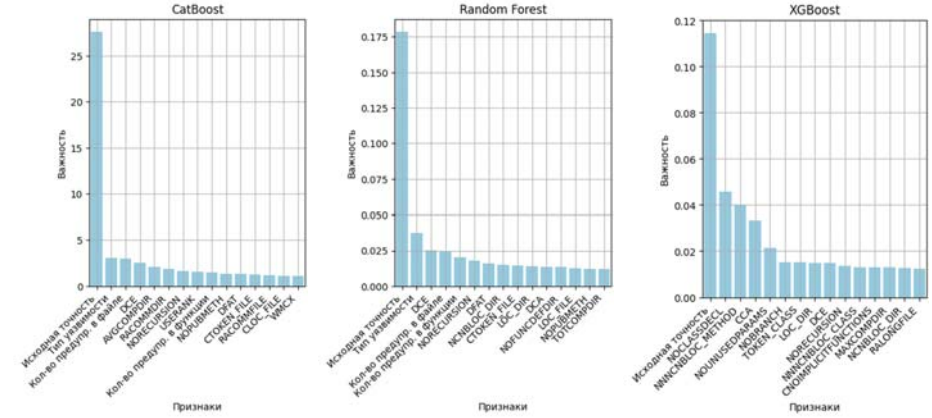

*Рис. 3. Важность признаков для **CatBoost**, **Random Forest** и **XGBoost** соответственно.*
*Fig. 3. Feature importances for **CatBoost**, **Random Forest** and **XGBoost** respectively.*

Из анализа графиков можно сделать вывод, что наиболее важным признаком для всех трех моделей является изначальная точность по истинным предупреждениям. Как уже было

упомянуто ранее, именно этот признак показывает, насколько хорошо работает детектор. Чем выше этот показатель, тем выше вероятность того, что предупреждение является истинным. Также важными признаками для моделей оказались количество предупреждений в файле и количество предупреждений в функции. Еще одним важным признаком для каждой модели является внешняя связность (DCE) [22]. Любое несогласованное изменение во внешних зависимостях может привести к ошибкам исполнения кода текущего модуля. Чем больше внешних зависимостей, тем больше вероятность таких изменений и, соответственно, истинности предупреждения. Также важными признаками являются NORECURSION (количество рекурсивных функций в файле), NOPUBMETH (количество публичных методов в классе), DFAT (количество зависимостей между классами каталога), CTOKEN_FILE (количество токенов в комментариях в файле), NCNBLOC_DIR (количество непустых строк кода в директории, не включая комментарии), LOC_DIR (количество строк кода в директории).

4. Интеграция механизма классификации предупреждений в статический анализатор Svace

Чтобы предоставить пользователям **Svace** доступ к функциональности классификации предупреждений, необходимо интегрировать механизм классификации в существующую инфраструктуру статического анализатора [23]. Статический анализатор **Svace** состоит из нескольких компонентов: подсистемы перехвата сборки, подсистемы анализа, включающей в себя большое количество независимых модулей (собственный многоязыковой движок символьного выполнения и анализа потоков данных, **Clang Static Analyzer** [24], **SpotBugs** [25], **SharpChecker** [26] и т.д.), а также **сервера истории** статического анализа. Сервер истории осуществляет хранение результатов статического анализа в базе данных и предоставляет возможности просмотра предупреждений и исходного кода в веб-интерфейсе, задания статусов предупреждений (не размеченное, истинное, ложное) и комментариев, а также автоматического сопоставления предупреждений между различными запусками статического анализатора, чтобы одни и те же предупреждения не приходилось просматривать повторно. При интеграции механизма классификации необходимо обеспечить выполнение всех этапов работы механизма (вычисление метрик, генерацию признаков, дообучение модели, выполнение классификации, загрузку результатов классификации на сервер для просмотра) наиболее удобным для использования способом, требующим наименьшего количества дополнительных действий со стороны пользователя.

4.1 Сценарии использования

Существует два основных сценария использования статического анализатора: проверка проекта «с нуля» и регулярное использование в процессе разработки программного продукта, при котором в базе данных сервера истории находится большое количество размеченных предупреждений проекта. Данные сценарии использования **Svace** задают два аналогичных сценария использования механизма классификации предупреждений, которые требуется поддержать – «холодный старт», при котором классификация выполняется при помощи предварительно обученной модели, что позволяет сразу начать использование механизма классификации, и классификацию с дообучением модели на пользовательской разметке, загружаемой с сервера истории, что позволяет существенно увеличить точность классификации.

4.2 Интеграция в процесс анализа

Вначале была рассмотрена задача «холодного старта». В этом режиме классификация выполняется при помощи предварительно обученной модели, распространяемой в составе дистрибутива **Svace**. Было решено встроить классификацию предупреждений в процесс

анализа после получения файла с предупреждениями от всех компонентов. Модификация процесса анализа показана в табл. 1.

4.3 Взаимодействие с сервером истории для обучения на пользовательских данных

При переходе от классификации при помощи предварительно обученной модели к дообучению на пользовательских данных возникла необходимость взаимодействия с сервером истории, чтобы автоматизировать процесс получения имеющихся на нём размеченных данных. **Svace** поддерживает работу с двумя серверами истории: встроенным сервером, реализованным непосредственно в составе **Svace**, и внешним сервером истории **Svacer** [27], разрабатываемым отдельной командой в ИСП РАН.

Табл. 1. Изменение последовательности действий в процессе анализа для интеграции механизма классификации предупреждений.
Table 1. Changes in the analysis actions order for integration of warnings classification mechanism.

Было	Стало
	вычисление полных метрик для кода на C/C++, Java и базовых метрик для кода на Kotlin, Go, Python
анализ кода на языках C/C++, Java, Kotlin, Go, Python	
анализ кода на C# и Visual Basic .NET и вычисление метрик для кода на C#[1] инструментом **SharpChecker**	
сохранение предупреждений, выданных всеми движками анализа, в единый файл	
вычисление базовых метрик для кода на C/C++, Java, Kotlin, Go, Python	генерация признаков для выданных предупреждений с использованием вычисленных метрик
	классификация полученных предупреждений при помощи предварительно обученной модели
	сохранение предупреждений и результатов классификации в итоговый файл

4.3.1 Встроенный сервер истории

Основной шаблон использования **Svace** со встроенным сервером истории состоит из следующих этапов:

- контролируемая сборка анализируемого проекта для генерации внутреннего представления;
- анализ полученного объекта сборки;
- импорт результатов анализа на встроенный сервер истории;
- просмотр и разметка результатов в веб-интерфейсе сервера истории.

Чтобы автоматизировать все шаги, необходимые для классификации предупреждений с использованием пользовательской разметки, без необходимости дополнительных действий

[1] За время разработки и интеграции механизма классификации в **SharpChecker** было реализовано вычисление метрик также для кода на языке Visual Basic .NET.

от пользователя, классификация была перенесена с этапа анализа на этап импорта на сервер истории, так как только на этом этапе **Svace** получает от пользователя информацию для подключения к серверу. Кроме того, анализ зачастую выполняется на другом компьютере или в изолированном окружении, не имеющем сетевого доступа к серверу истории, что обосновывает данное решение. Схема работы механизма классификации на этапе импорта результатов показана на рис. 4.

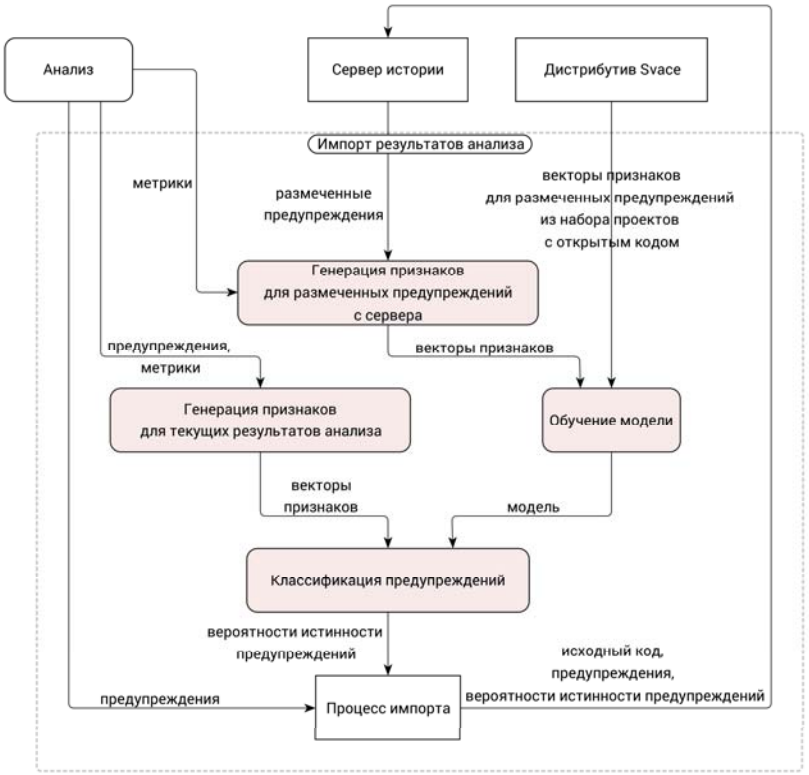

Рис. 4. Схема интеграции механизма классификации предупреждений в этап импорта.
Fig. 4. Scheme of integration of the warning's classification mechanism into the import stage.

Как показано на данной схеме, классификация предупреждений выполняется моделью, обученной на размеченных предупреждениях из анализируемого проекта, а также на предупреждениях из проектов с открытым кодом, размеченных разработчиками **Svace** и экспертами Центра верификации ОС Linux [28]. Предварительно сгенерированные векторы признаков для набора проектов с открытым кодом распространяются в составе дистрибутива **Svace**. Векторы признаков для предупреждений с сервера истории генерируются с использованием метрик, вычисленных в текущем запуске анализатора, так как метрики имеют большой объём, и сервер истории не обладает функциональностью хранения метрик.

Итоговая реализация интеграции механизма классификации предупреждений практически не требует от пользователя дополнительных действий, таких, как запуск дополнительных команд вручную или существенная модификация скриптов запуска статического анализатора

в системе непрерывной интеграции. Достаточно лишь включить вычисление полных метрик на этапах сборки и анализа, и тогда получение размеченных данных, обучение модели и классификация предупреждений будут выполнены автоматически, после чего результаты будут доступны для просмотра на встроенном сервере истории.

4.3.2 Сервер истории Svacer

Сервер истории **Svacer** является отдельным инструментом, разрабатываемым отдельно от **Svace** и не входящим в его состав, что затрудняет процесс внесения изменений и добавления новой функциональности. Это делает нежелательной интеграцию механизма классификации предупреждений непосредственно в процесс импорта результатов анализа в **Svacer**. К тому же, частичное дублирование реализации в двух независимых программных компонентах, написанных на разных языках программирования (Java и Go), существенно увеличило бы трудозатраты на разработку и поддержку и привело бы к неизбежному накоплению различий и неисправленных ошибок. Поэтому было решено не интегрировать запуск классификации предупреждений непосредственно в **Svacer**, а реализовать взаимодействие с сервером **Svacer** через публичный API [29] в виде вспомогательной команды **Svace**, повторно используя реализованный в составе команды импорта запуск процессов генерации признаков, обучения модели и классификации предупреждений.

4.4 Отображение результатов классификации предупреждений в пользовательском интерфейсе

Отображение результатов работы механизма автоматической классификации было внедрено в пользовательский интерфейс для просмотра и анализа предупреждений встроенного сервера истории. Результатами работы механизма являются предсказанные моделью вероятности истинности для каждого предупреждения.

На рис. 5 показано, как выглядит обновленный элемент пользовательского интерфейса.

Каждая строка представляет собой информацию о предупреждении. Проанализированные вручную предупреждения выделены цветом фона: истинные – зелёным, ложные –красным. Вероятность истинности отображается в процентах в правом верхнем углу каждого предупреждения.

Ручной анализ данных предупреждений производился уже после работы механизма автоматической классификации и подтвердил корректность работы механизма для данных предупреждений.

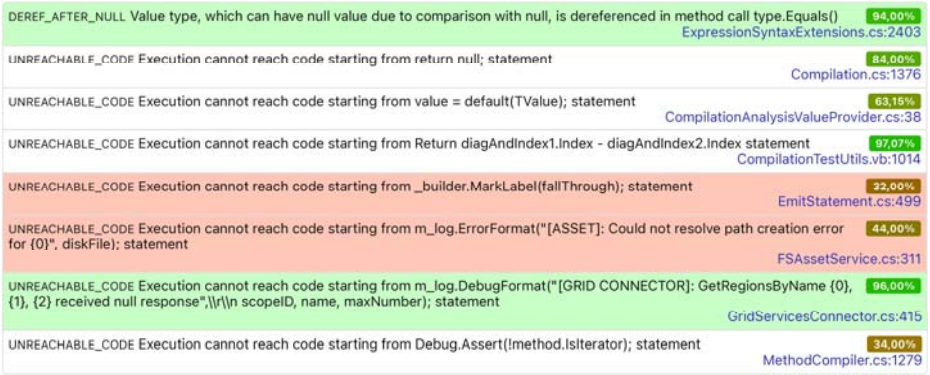

Рис. 5. Фрагмент пользовательского интерфейса встроенного сервера истории.
Fig. 5. Fragment of built-in history server UI.

Пользовательский интерфейс также предоставляет возможность сортировки предупреждений по вероятности истинности, а также фильтрации предупреждений по пороговому значению вероятности, задаваемому пользователем.

В пользовательском интерфейсе **Svacer** в настоящее время поддерживается только отображение вероятности истинности для текущего открытого предупреждения.

5. Результаты тестирования

Для тестирования моделей машинного обучения использовался набор данных, сгенерированных на основе предупреждений, полученных с помощью статического анализа 21 проекта с открытым исходным кодом на языке C#, имеющих суммарный размер более 6 млн. строк кода. Тестирование проводилось на компьютере с процессором AMD Ryzen 5 4600G и 64 ГБ оперативной памяти, работающем под управлением операционной системы Windows 10. Всего было сгенерировано 68182 предупреждения, 13182 из которых были проанализированы и размечены вручную более чем за 5 лет в процессе разработки инструмента статического анализатора. Генерация признаков для всех предупреждений заняла 2 минуты и 20 секунд, в то время как статический анализ этих проектов суммарно занял около 3 часов 30 минут. При процентном разделении набора данных на части, описанном ранее, тренировочный набор данных составляет 8167 векторов признаков, валидационный – 1605, тестовый – 3465.

Время обучения и настройка гиперпараметров для всех моделей классификаторов различается. Для данного набора данных классификатор **CatBoost** обучается около 43 секунд, настройка гиперпараметров и обучение для классификатора **Random Forest** длится 3 минуты и 13 секунд. Для классификатора **XGBoost** подготовка модели происходит дольше – около 20 минут. Так происходит из-за особенностей классификатора, для нормальной работы классификатора **XGBoost** требуется более тонкая настройка, чем для двух других моделей машинного обучения.

В табл. 2 представлены метрики качества бинарной классификации для результатов работы моделей на описанном наборе данных без отбора признаков.

Из таблицы видно, что объектов для истинных предупреждений все метрики выше, чем для ложных предупреждений. Это как раз связано с дисбалансом классов в существующем наборе данных. Чтобы равноценно классифицировать как истинные предупреждения, так и ложные, необходимо помимо точности рассматривать другие метрики качества для выбора подхода и модели машинного обучения. Однако стоит заметить, что показатель AUC-ROC (площадь под кривой ошибок) довольно высокий для всех моделей, что показывает, что каждая модель хорошо отличает класс истинных предупреждений от класса ложных.

Ранее было показано, что при ограничении количества признаков до 100 потери изначальных зависимостей происходить не будет, но при этом сами модели станут проще, время обучения станет меньше и, возможно, точность работы моделей может увеличиться. Метрики бинарной классификации для результатов работы моделей машинного обучения на наборах данных с первыми 100 по важности признаками для каждой модели машинного обучения по отдельности представлены в табл. 3.

Основное внимание при оценке результатов классификации необходимо уделить F_1-мерам для истинных и ложных предупреждений, поскольку F_1-мера является комбинацией точности и полноты классификации. Для классификатора **CatBoost**, как и для классификатора **Random Forest**, данный показатель вырос как для первого класса, так и для нулевого. Однако для модели **XGBoost** F_1-мера уменьшилась и для первого, и для нулевого класса. Это могло произойти из-за того, что данный классификатор обычно выигрывает за счет того, что каждый уровень дерева из ансамбля использует разные признаки для разбиения данных, то есть при удалении признаков деревья могли стать менее разнообразными, и точность могла упасть.

Табл. 2. Метрики для набора данных с полным списком признаков.
Table 2. Quality metrics for dataset with the full list of features.

Классификатор		CatBoost	Random Forest	XGBoost
Истинные предупреждения	Точность	93,21%	92,15%	93,21%
	Полнота	96,29%	97,36%	96,77%
	F_1	94,73%	94,68%	94,96%
Ложные предупреждения	Точность	84,46%	87,71%	86,16%
	Полнота	74,19%	69,46%	74,05%
	F_1	78,99%	77,53%	79,65%
Сбалансированная полнота		85,24%	83,41%	85,41%
AUC-ROC		96,61%	96,20%	96,77%

Табл. 3. Метрики для набора данных с отобранными признаками.
Table 3. Quality metrics for dataset with selected features.

Классификатор		CatBoost	Random Forest	XGBoost
Истинные предупреждения	Точность	93,73%	93,21%	93,13%
	Полнота	95,96%	96,77%	96,48%
	F_1	94,83%	94,96%	94,77%
Ложные предупреждения	Точность	83,70%	86,16%	85,05%
	Полнота	76,35%	74,05%	73,78%
	F_1	79,86%	79,65%	79,02%
Сбалансированная полнота		86,16%	85,41%	85,13%
AUC-ROC		96,49%	96,37%	96,80%

Все последующие графики будут для моделей **CatBoost** и **Random Forest**, обученных на наборе данных с сокращенным списком признаков, а для модели **XGBoost** – с полным списком признаков.

Визуализировать различимость классов каждой модели можно с помощью графика плотностей распределения классов (рис. 6). По оси абсцисс у графиков распределения плотности классов отложены значения предсказаний модели. То есть с их помощью можно увидеть, где, относительно предсказаний модели, сконцентрированы истинные и ложные предупреждения. Для всех трех моделей пики плотностей истинных и ложных предупреждений различимы и находятся на большом расстоянии. Это и говорит о том, что все три модели упорядочивают большинство пар (элемент класса ложных предупреждений, элемент класса истинных предупреждений) верно.

При оценке точности результатов работы моделей пороговое значение классификации по умолчанию равно 0.5. Однако на графиках плотностей распределения классов видно, что переход от значений, для которых плотность распределения ложных предупреждений выше, чем плотность распределения истинных, к значениям, где плотность распределения

истинных предупреждений выше, чем плотность распределения ложных, происходит при вероятности истинности, большей 0.5.

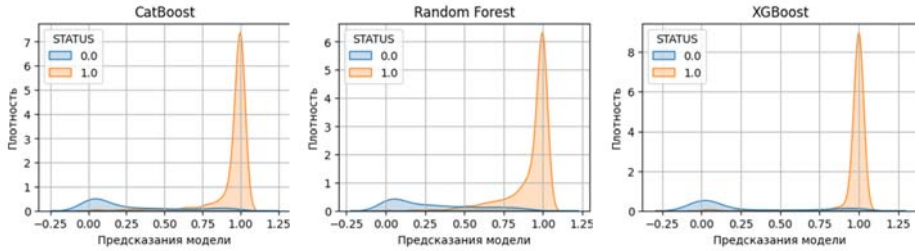

*Рис. 6. Плотность распределения классов для **CatBoost**, **Random Forest** и **XGBoost** соответственно.*
*Fig. 6. Class probability density for **CatBoost**, **Random Forest** and **XGBoost** respectively.*

Чтобы повысить точность работы моделей, можно попробовать изменить пороговое значение классификации. Из графиков также понятно, что при пороговом значении, равном 0.5, полнота по истинным предупреждениям будет выше, чем при пороговых значениях, больших 0.5. Таким образом при выставлении нового значения порогового значения нужно увеличить метрики качества для определения класса 0 и при этом сохранить метрики качества для определения класса 1. Чтобы более сбалансировано выбирать новое пороговое значение, построим графики F_1-мер (рис. 7) для истинных и ложных предупреждений с разными пороговыми значениями классификации.

В качестве пороговых значений берутся числа от 0 до 1 не включительно с шагом 0.01. Верхняя кривая на каждом графике показывает зависимость F_1-меры от порогового значения для истинных предупреждений, нижняя кривая – для ложных предупреждений. Красными горизонтальными прямыми отмечены максимумы данных кривых. Вертикальная прямая отображает пороговое значение. Пороговое значение выбирается так, чтобы F_1-мера была близка к максимуму как для истинных предупреждений, так и для ложных, поскольку с помощью F_1-меры можно найти баланс между точностью и полнотой.

Пороговыми значениями для моделей были выбраны следующие: **CatBoost** – 0.57, **Random Forest** – 0.54, **XGBoost** – 0.79. Метрики качества бинарной классификации с заданными пороговыми значениями представлены в табл. 4. Для всех моделей показатели качества классификации достаточно высокие, однако учитывая качество и время обучения модели, наилучшим классификатором является модель **CatBoost**. При высокой точности в 91.92% удалось добиться высоких показателей полноты, как для класса истинных предупреждений (94.89%), так и для класса ложных предупреждений (80.74%).

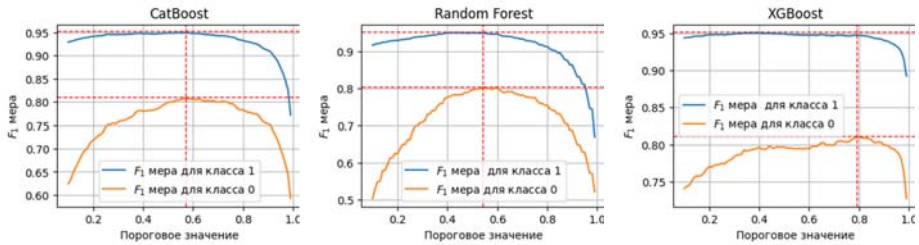

*Рис. 7. F_1-мера для **CatBoost**, **Random Forest** и **XGBoost** соответственно.*
*Fig. 7. F_1-measure for **CatBoost**, **Random Forest** and **XGBoost** respectively.*

Табл. 4. Метрики для моделей с выставленным пороговым значением.
Table 4. Quality metrics for models with threshold.

Классификатор		CatBoost	Random Forest	XGBoost
Истинные предупреждения	Точность	94,44%	93,89%	95,02%
	Полнота	95,34%	95,85%	94,57%
	F_1	94,89%	94,86%	94,79%
Ложные предупреждения	Точность	82,21%	83,46%	80,35%
	Полнота	79,32%	77,03%	81,76%
	F_1	80,74%	80,11%	81,04%
Сбалансированная полнота		87,33%	86,44%	88,16%
AUC-ROC		96,49%	96,37%	96,85%

Матрица ошибок для выбранной модели представлена в табл. 5. В матрице ошибок для классификатора **CatBoost** продемонстрированы абсолютные значения для количества предупреждений и их процентное соотношение для тестового набора данных, размером 3465 предупреждений.

Поскольку разработанный механизм автоматической классификации предупреждений не зависим от языка программирования, его можно применять ко всем языкам программирования, для которых статический анализатор рассчитывает метрики исходного кода, то есть можно получить результаты работы для языков программирования C, C++ и Java.

*Табл. 5. Матрица ошибок для модели **CatBoost**.*
Table 5. Confusion matrix for CatBoost model.

		Действительные	
		Ложные	Истинные
Предсказанные	Ложные	587 (82%)	127 (18%)
	Истинные	153 (6%)	2598 (94%)

Предсказание результатов производилось с помощью классификатора **CatBoost**. Для каждого языка обучалась отдельная модель. Результаты представлены в таблицах 6 и 7.

Для языков C, C++ тестирование производилось на наборе из 4483 предупреждений. В качестве признаков были отобраны первые 100 по важности для модели машинного обучения признаков. С пороговым значением 0.55 была достигнута точность классификации в 82.42%. Точность работы статического анализатора на валидационной выборке равна 68.34%. То есть при фильтрации ложных предупреждений с помощью механизма автоматической классификации точность вырастет на 16.43%. При этом полнота классификации истинных предупреждений – 89.87%.

Для языка Java тестирование производилось на наборе из 1155 предупреждений. В качестве признаков были отобраны первые 80 по важности для модели машинного обучения признаков. С пороговым значением 0.48 была достигнута точность классификации в 85.76%. Точность работы статического анализатора на валидационной выборке равна 61.63%. То есть при фильтрации ложных предупреждений с помощью механизма автоматической

классификации точность вырастет на 25.58%. При этом полнота классификации истинных предупреждений – 90.09%.

Из-за малого количества предупреждений в обучающем наборе данных, достигнуть высоких показателей для классификации ложных предупреждений для данных языков программирования не удалось: F_1-мера для ложных предупреждений составляет 69.67% для C, C++ и 80.93% для Java.

Табл. 6. Матрица ошибок для C/C++ и Java.
Table 6. Confusion matrix for C/C++ and Java.

		Действительные C/C++		Действительные Java	
		Ложные	Ложные	Истинные	Ложные
Предсказанные	Ложные	271 (75%)	91 (25%)	104 (83%)	21 (17%)
	Истинные	145 (15%)	807 (85%)	28 (13%)	191 (87%)

Табл. 7. Метрики для C/C++ и Java.
Table 7. Quality metrics for C/C++ and Java.

Язык программирования		C/C++	Java
Истинные предупреждения	Точность	84,77%	87,21%
	Полнота	89,87%	90,09%
	F1	87,24%	88,63%
Ложные предупреждения	Точность	73,86%	83,20%
	Полнота	65,14%	78,79%
	F1	69,67%	80,93%
Сбалансированная полнота		77,51%	84,44%
AUC-ROC		87,88%	92,38%

6. Заключение

В рамках данной работы были выполнены следующие задачи:

- доработан механизм генерации признаков с учетом выявленных особенностей статического анализатора **Svace** и языков программирования C#, C/C++, Java;
- выбрана модель машинного обучения **CatBoost**, имеющая оптимальный баланс между производительностью и показателями качества классификации для данной задачи;
- метод классификации предупреждений реализован и протестирован на существующем наборе данных и показал высокое качество предсказания истинности предупреждений;
- механизм автоматической классификации предупреждений интегрирован в инфраструктуру статического анализатора **Svace** [30];
- реализовано отображение результатов классификации предупреждений в пользовательском интерфейсе с возможностью сортировки и фильтрации.

Список литературы / References

[1]. Tsiazhkorob U. V., Ignatyev V. N. Classification of Static Analyzer Warnings using Machine Learning Methods //2024 Ivannikov Memorial Workshop (IVMEM). – IEEE, 2024. – С. 69-74.

[2]. Иванников В. П. и др. Статический анализатор Svace для поиска дефектов в исходном коде программ //Труды Института системного программирования РАН. – 2014. – Т. 26. – №. 1. – С. 231-250.
[3]. Lee S. et al. Classifying false positive static checker alarms in continuous integration using convolutional neural networks //2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST). – IEEE, 2019. – С. 391-401.
[4]. Alikhashashneh E. A., Raje R. R., Hill J. H. Using machine learning techniques to classify and predict static code analysis tool warnings //2018 IEEE/ACS 15th International Conference on Computer Systems and Applications (AICCSA). – IEEE, 2018. – С. 1-8.
[5]. Christmann A., Steinwart I. Support vector machines. – 2008.
[6]. Dasarathy B. V. Nearest neighbor (NN) norms: NN pattern classification techniques //IEEE Computer Society Tutorial. – 1991.
[7]. Breiman L. Random forests //Machine learning. – 2001. – Т. 45. – С. 5-32.
[8]. Witten I. H. et al. Data Mining: Practical machine learning tools and techniques. – Elsevier, 2025.
[9]. Rajput A. et al. J48 and JRIP rules for e-governance data //International Journal of Computer Science and Security (IJCSS). – 2011. – Т. 5. – №. 2. – С. 201.
[10]. Understand: The Software Developer's Multi-Tool [Электронный ресурс]. – URL: https://scitools.com.
[11]. CWE – Common Weakness Enumeration [Электронный ресурс]. – URL: https://cwe.mitre.org.
[12]. Center for Assured Software N.S.A. Juliet Test Suite v1.1 for C/C++ User Guide. [Электронный ресурс]. – URL: https://samate.nist.gov/SARD/downloads/documents/Juliet_Test_Suite_v1.1_for_C_Cpp_-_User_Guide.pdf.
[13]. Software metric [Электронный ресурс]. URL: https://en.wikipedia.org/wiki/Software_metric
[14]. Lee M. C. Software quality factors and software quality metrics to enhance software quality assurance //British Journal of Applied Science & Technology. – 2014. – Т. 4. – №. 21. – С. 3069-3095.
[15]. Белеванцев А. А., Велесевич Е. А. Анализ сущностей программ на языках Си/Си++ и связей между ними для понимания программ //Труды Института системного программирования РАН. – 2015. – Т. 27. – №. 2. – С. 53-64.
[16]. McKinney W. et al. Data structures for statistical computing in Python //SciPy. – 2010. – Т. 445. – №. 1. – С. 51-56.
[17]. Pedregosa F. et al. Scikit-learn: Machine learning in Python //the Journal of machine Learning research. – 2011. – Т. 12. – С. 2825-2830.
[18]. Beautiful Soup: We called him Tortoise because he taught us. [Электронный ресурс]. URL: https://www.crummy.com/software/BeautifulSoup/.
[19]. Dorogush A. V., Ershov V., Gulin A. CatBoost: Gradient boosting with categorical features support. arXiv 2018 //arXiv preprint arXiv:1810.11363. – 1810.
[20]. Parmar A., Katariya R., Patel V. A review on random forest: An ensemble classifier //International conference on intelligent data communication technologies and internet of things. – Cham : Springer International Publishing, 2018. – С. 758-763.
[21]. Chen T., Guestrin C. Xgboost: A scalable tree boosting system //Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining. – 2016. – С. 785-794.
[22]. Cohesion (computer science) [Электронный ресурс]. URL: https://en.wikipedia.org/wiki/Cohesion_(computer_science).
[23]. Belevantsev A. et al. Design and development of Svace static analyzers //2018 Ivannikov Memorial Workshop (IVMEM). – IEEE, 2018. – С. 3-9.
[24]. T. Kremenek. Finding software bugs with the Clang static analyzer [Электронный ресурс]. URL: https://llvm.org/devmtg/2008-08/Kremenek_StaticAnalyzer.pdf.
[25]. SpotBugs: Find bugs in Java Programs [Электронный ресурс]. URL: https://spotbugs.github.io.
[26]. Koshelev V. K. et al. SharpChecker: Static analysis tool for C# programs //Programming and Computer Software. – 2017. – Т. 43. – С. 268-276.
[27]. Svacer Wiki [Электронный ресурс]. URL: https://svacer.ispras.ru/.
[28]. О центре | Linux Verification Center. URL: http://linuxtesting.ru/about.
[29]. Swagger UI: Svacer Server 10.x.x. Svacer REST API documentation [Электронный ресурс]. URL: https://svacer-demo.ispras.ru/api/public/swagger/.
[30]. AI-ассистент для разметки предупреждений анализатора [Электронный ресурс]. URL: https://svace.pages.ispras.ru/svace-website/2025/02/21/ai-assistant.html.

Информация об авторах / Information about authors

Ульяна Владимировна ТЯЖКОРОБ – аспирант Физтех-школы Радиотехники и Компьютерных Технологий МФТИ, сотрудник ИСП РАН. Научные интересы: компиляторные технологии, статический анализ программ, машинное обучение.

Uljana Tsiazhkarob – postgraduate student of the Phystech School of Radio Engineering and Computer Technologies of MIPT, employee of the ISP RAS. Research interests: compiler technologies, static program analysis, machine learning.

Михаил Владимирович БЕЛЯЕВ – младший научный сотрудник ИСП РАН. Научные интересы: компиляторные технологии, статический анализ программ.

Mikchail Vladimirovich BELYAEV – researcher at ISP RAS. Research interests: compiler technologies, static program analysis.

Андрей Андреевич БЕЛЕВАНЦЕВ – доктор физико-математических наук, ведущий научный сотрудник ИСП РАН, профессор кафедры системного программирования ВМК МГУ. Сфера научных интересов: статический анализ программ, оптимизация программ, параллельное программирование.

Andrey Andreevich BELEVANTSEV – Dr. Sci. (Phys.-Math.), Prof., leading researcher at ISP RAS, Professor at Moscow State University. Research interests: static analysis, program optimization, parallel programming.

Валерий Николаевич ИГНАТЬЕВ – кандидат физико-математических наук, старший научный сотрудник ИСП РАН, доцент кафедры системного программирования факультета ВМК МГУ. Научные интересы включают методы поиска ошибок в исходных текстах программ на основе статического анализа.

Valery Nikolayevich IGNATYEV – Cand. Sci. (Phys.-Math.) in computer sciences, senior researcher at Ivannikov Institute for System Programming RAS and associate professor at system programming division of CMC faculty of Lomonosov Moscow State University. His research interests include program analysis techniques for error detection in program source code using classical static analysis and machine learning.