###### **ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

###### **"НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ**

###### **"ВЫСШАЯ ШКОЛА ЭКОНОМИКИ"**

###### **ФАКУЛЬТЕТ КОМПЬЮТЕРНЫХ НАУК**

Трофимов Николай Олегович, группа МНОД231

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

«Методы оценки качества классификационных деревьев»

по направлению подготовки 01.04.02 Прикладная математика и информатика

образовательная программа « Науки о данных »

Студент

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Н.О. Трофимов

д.т.н., профессор ФКН ВШЭ

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Б.Г. Миркин

Москва 2025

**АННОТАЦИЯ**

Деревья решений остаются одним из наиболее популярных и интерпретируемых методов машинного обучения. Однако выбор критерия расщепления существенно влияет на точность и стабильность модели: наряду с классическими показателями (Gini, энтропия, χ² и др.) в последние годы предложен ряд новых критериев, чья эффективность пока изучена недостаточно.

Цель дипломной работы — провести комплексный эксперимент по нескольким классическим и новейшим критериям разбиения дерева решений, чтобы количественно оценить качество каждой метрики и выявить их преимущества и недостатки. Для достижения цели поставлены задачи:

1. Реализовать выбранные критерии в едином программном модуле на C++ с акцентом на ускоренный подсчёт статистик и поиск оптимальных разбиений;
2. Провести масштабное сравнение по суперкритерям
3. Проанализировать результаты и определить области задач, где каждый критерий показывает себя наилучшим образом.

В ходе работы был детально изучен и реализован оптимальный по времени алгоритм классификации с использованием дерева решений. Исследованы и применены различные open source библиотеки для разработки ML-проектов. Также изучен альтернативный подход к разбиению дерева решений на основе научной статьи

Оглавление

[Введение 4](#_Toc198780252)

[1. Новые критерии разбиения дерева ‘ 7](#_Toc198780253)

[1.1 Принятые обозначения 7](#_Toc198780254)

[1.2 Определение новых критерий 7](#_Toc198780255)

[1.3 Упрощение критериев 8](#_Toc198780256)

[2. Написание программного модуля на C++ для python 9](#_Toc198780257)

[2.1 Обзор инструментов связывания C++ и Python для высокопроизводительного ML‑кода 9](#_Toc198780258)

[2.2 Эффективное использование библиотеки Pybind11 10](#_Toc198780259)

[2.3 Выбор структуры данных для хранения дерева 12](#_Toc198780260)

[2.4 Базовый способ подсчета критериев 13](#_Toc198780261)

[2.5 Динамический подсчет критериев 14](#_Toc198780262)

[2.6 Остановка разбиения дерева решений 15](#_Toc198780263)

[2.7 Удобность создаваемого интерфейса 16](#_Toc198780264)

[2.8 Бенчмарк полученных алгоритмов 17](#_Toc198780265)

[2.9 Разные способы компиляции 19](#_Toc198780266)

[2.10 Генерирование синтетических данных 21](#_Toc198780267)

[3. Проведение эксперимента и анализ качества модели 23](#_Toc198780268)

[3.1 Основные метрики эксперимента 23](#_Toc198780269)

[3.2 Результаты эксперимента 24](#_Toc198780270)

[3.3 Анализ полученных результатов 28](#_Toc198780271)

[3.4 Эксперимент на известных датасетах 34](#_Toc198780272)

[Заключение 36](#_Toc198780273)

[Приложение 38](#_Toc198780274)

[Список используемой литературы 39](#_Toc198780275)

# Введение

За последние два десятилетия машинное обучение совершило квантовый скачок: от простых линейных регрессий до глубоких нейронных сетей, способных распознавать речь, переводить тексты и создавать фотореалистичные изображения. Однако вместе с прорывом возникла новая проблема — непрозрачность. Большинство современных «тяжёлых» моделей — градиентный бустинг сотен тысяч деревьев, сверточные и трансформерные сети с миллиардами параметров — дают высокую точность, но не позволяют специалисту легко понять, *почему* было принято то или иное решение. Для критических областей — медицины, финансового скоринга, судебной экспертизы, промышленной диагностики — такая «чёрная‑коробка» неприемлема: регуляторы и конечные пользователи требуют объяснимости, а исследователи — контроля над выводами модели.

На этом фоне особую ценность приобретают интерпретируемые модели, архитектура и работа которых интуитивно ясны человеку. Среди них уже несколько десятилетий выделяются деревья решений.

Дерево решений — это ориентированная ациклическая структура, которая рекурсивно разбивает пространство признаков на подмножества с помощью простых логических тестов (сплитов), а в листовых вершинах хранит прогноз (метку класса или числовое значение). Каждому объекту сопоставляется путь от корня к листу, последовательно уточняющий принадлежность к той или иной подобласти данных.

Одним из первых и самых известных алгоритмов, который строил дерево решений, является CART [1] (Classification and Regression Trees). Его основная идея сводится к жадному рекурсивному разбиению выборки: начиная с корня, алгоритм перебирает для каждого признака все возможные пороги и выбирает тот бинарный сплит, который наибольшим образом уменьшает меру неоднородности узла. После разделения данные в дочерних узлах обрабатываются тем же принципом, формируя всё более однородные подмножества, пока не выполнится условие остановки (например, минимальный размер узла или нулевая импьюрити).

В качестве вычисления меры неоднородности узла в алгоритме карта использовалась gini-impurity, определяемая следующим образом:

Где — доля объектов l‑го класса внутри узла. Для регрессионных задач CART использует аналогичную идею мини­мизации разброса, измеряя дисперсию (сумму квадратов отклонений) целевой переменной. Эти функции стали первыми практическими критериями неоднородности, позволяющими быстро оценить, насколько «чистым» станет каждый из дочерних узлов после разбиения.

Со временем исследователи предложили и другие меры неопределённости, расширяющие или уточняющие классический индекс:

1. Information Gain (энтропия Шеннона) — введён Р. Куинланом в статье [2]. Измеряет уменьшение энтропии после разбиения.
2. Gain Ratio — предложен тем же автором в книге [2] нормализует Information Gain, избавляя от смещения в пользу признаков с большим числом уникальных значений.
3. Chi‑square (CHAID) — описан в работе [3] основан на статистическом тесте независимости между признаком и классом.

Выбор критерия неоднородности оказывает существенное влияние на итоговые показатели качества построенного дерева, поскольку каждый из них акцентирует различные аспекты структуры выборки. Индекс Джини характеризуется высокой вычислительной эффективностью и умеренной чувствительностью к редким классам, что часто обеспечивает наибольшую общую accuracy при умеренно сбалансированных данных. Энтропийный прирост информации, максимально штрафуя смешанность классов, способствует улучшению показателей F1‑score и balanced accuracy в задачах, где требуется равновесие между полнотой и точностью. Статистика χ², усиливая вклад малочисленных, но информативных категорий, позволяет повысить recall минорного класса в дисбалансных выборках, типичных для маркетинговых или медицинских приложений. Таким образом, грамотный подбор критерия разбиения способен привести не только к значимому приросту стандартных метрик (accuracy, recall, F1‑score), но и к формированию более компактной и интерпретируемой структуры дерева, что в конечном итоге повышает надёжность и адаптивность модели при дальнейшем её применении.

Но может существовать куда больше метрик, чем принято считать?  
Современные исследования показывают, что классические показатели — Джини, энтропия, χ² — являются лишь частными случаями более широкой формулы. В рамках данной работы будут введены альтернативные критерии разбиения и затем сравнены между собой на специально сгенерированных искусственных наборах данных.

# 1. Новые критерии разбиения дерева ‘

* 1. **Принятые обозначения**
* N – число объектов
* V = L – число признаков
* K – число кластеров
* – split (от английского «разбиение»)
* – конструируемое разбиение в ходе построения дерева
* – истинное разбиение наших данных
  1. **Определение новых критерий**

Решая задачу классификации у нас, есть какое – то заданное разбиение R = {…}, относительно которого мы хотим построить наш классификатор S = {…}.

В учебнике [4] «вводится функция для измерения качества классификатора S по отношению к «истинному» разбиению R. Выглядит данный скор следующим образом:

**(\*)**

Причем в этой формуле вероятности можно интерпретировать следующим образом,

– вероятность выбрать элемент k-ого split из всех элементов

– вероятность выбрать элемент k-ого split с меткой l из множества всех элементов

– шкалирующий фактор

Поскольку ​ входит только в знаменатель, он задаёт относительный «вес» этого класса при оценке сходства двух разбиений. Выбирая различные значения ​, можно гибко подстраивать критерий под статистические свойства выборки. В работе будут расммотрены следующие возможные значения шкалиируюшего параметра:

1. . Это случай, когда фактор не зависит от частоты
2. . Фактор равен стандартному отклонению при гипотезе Пуассонова распределения
3. . Фактор равен стандартному отклонению при гипотезе биномиального распределения Бернулли
4. . Фактор равен частоте
   1. **Упрощение критериев**

Чтобы сделать вычисления практичными, целесообразно привести «общую» формулу **(\*)** к упрощённым видам для четырех конкретных вариантов шкалирующего фактора ​.

При

**-**2g(R) -g(S)-1 **(1\*)**

В данных обозначениях g – это gini variance

При не получается интересных упрощений

При

**-**2=(3\*)

Причем , данная функция слегка напоминает gini variance

При . Фактор равен частоте

**-**2,

F(S,R)= **–** L =**–** L (4\*)

Среди полученных формул (1\*), (2\*), (3\*), (4\*) можно выбирать любую и стараться её максимизировать при разбиении. На последнем этапе алгоритмы мы также запомним эту величину и выведем пользователю

# 2. Написание программного модуля на C++ для python

## **` 2.1 Обзор инструментов связывания C++ и Python для высокопроизводительного ML‑кода**

В прикладных системах машинного обучения на Python к критическим участкам вычислений предъявляются требования, выходящие за пределы возможностей интерпретатора: динамическая типизация, управление ссылками и глобальная блокировка GIL привносят существенный накладной расход при многократных линейных операциях и итеративных оптимизационных циклах. Интеграция скомпилированного C++‑модуля позволяет перенести ресурсоёмкие процедуры в среду статически типизированного кода, где они выполняются с заметно более высокой скоростью благодаря оптимизациям компилятора и прямому доступу к данным в памяти. При этом верхний уровень остаётся в Python, что сохраняет лаконичность сценариев, совместимость с экосистемой библиотек и удобство аналитической работы. Такая двуслойная архитектура стала отраслевым стандартом, поскольку обеспечивает одновременно производительность нативного кода и гибкость высокоуровневого API.

Подход «Python‑над C++» уже реализован в большинстве ключевых пакетов: NumPy и pandas вычисляют массивные операции через BLAS‑/LAPACK‑обёртки на C/Fortran; scikit‑learn обрабатывает деревья решений и к‑средних в Cython/C++.

**CPython C API**  
Базовый, низкоуровневый интерфейс самого интерпретатора Python (Python.h). Предоставляет полный контроль над объектами PyObject\* и позволяет достигать минимального вызовного оверхеда. Недостаток заключается в высокой трудоёмкости разработки: управление ссылками, ручной разбор аргументов и явная обработка ошибок заметно усложняют код. Такие трудозатраты делают его нецелесообразным выбором для данного проекта, поэтому этот вариант исключён из дальнейшего рассмотрения.

**Pybind11**

В данной дипломной работе был выбран pybind11 [5]. Эта заголовочная библиотека позволяет описывать обёртку (binding) для функции или класса буквально в несколько строк, без громоздкого ручного кода, который требуется при использовании чистогоC API. В ней уже реализованы преобразователи для контейнеров STL и, что особенно важно для задач машинного обучения, для массивов NumPy — данные передаются без дополнительных копирований. RAII‑механизмы автоматически управляют объектами Python, снижая риск утечек памяти и сегфолтов.Такое сочетание простоты, встроенной поддержки ключевых структур данных и высокой скорости делает pybind11 наиболее подходящим решением в рамках поставленной задачи.

## **2.2 Эффективное использование библиотеки Pybind11**

Поскольку в проекте планируется напрямую работать с массивами NumPy, прежде всего нужно понимать, как объект ndarray представлен в самом Python

Массив NumPy (ndarray) хранит элементы в одном непрерывном буфере, адресуемом указателем data, позиция любого элемента вычисляется как линейная комбинация индексов и смещений strides, задающих число байтов, на которое нужно сдвинуться вдоль каждой оси. Размещение буфера бывает row‑major (C‑order) или column‑major (Fortran‑order). Именно row‑major используется по умолчанию, тогда как column‑major создаётся только при явном параметре

Поскольку ndarray уже хранит данные в непрерывном буфере, при вызове C++‑функции из Python критично избежать лишнего копирования этого блока: дополнительная аллокация не только расходует память, но и нивелирует выгоду от использования нативного кода. В pybind11 эту задачу решает класс‑обёртка py::array\_t<T>, который принимает объект NumPy по ссылке, проверяет его непрерывность и просто сохраняет указатель data, а также метаданные shape и strides. Таким образом, C++‑функция получает прямой доступ к исходному буферу Python‑массива, выполняет вычисления in place или, при необходимости, возвращает тот же объект без создания дублей, полностью сохраняя нулевую стоимость копирования.

Если бы интерфейс C++ функции объявлялся как std::vector<std::vector<T>>, pybind11 при вызове из Python вынужден был бы сначала разбирать объект ndarray построчно: для каждой строки создавать новый вложенный std::vector, копировать в него элементы и затем помещать этот вектор во внешний контейнер. Такой каскад преобразований означает десятки или сотни отдельных аллокаций вместо одной, полное дублирование данных. Кроме того, получившаяся «разреженная» структура хранит строки в памяти раздельно, лишая их преимущества непрерывного буфера и затрудняя последующие низкоуровневые операции. Поэтому использование std::vector<std::vector<T>> для приема NumPy‑массивов не только провоцирует лишние копирования, но и делает дальнейшую работу с данными неудобной и менее производительной, гораздо рациональнее принимать массив через py::array\_t<T> и обрабатывать его как единый непрерывный блок.

Для проверки описанного выше факта, можно составить простую программу, которая принимала бы два std::vector<std::vector<T>>, а на стороне python code передавала две матрицы, каждая размером 500 на 500. В результате такого эксперимента, снимем флеймграф и посмотрим на рисунке 1, что действительно происходит много операций «конвертаций» и аллокаций памяти

Изображение выглядит как снимок экрана, Мультимедийное программное обеспечение, программное обеспечение, Графическое программное обеспечение

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 1. Флеймграф с использованием std::vector<std::vector<T>>

Также стоит отметить, что если мы выбрали для работы объект py::array\_t<T>, то необходимо самым наиболее быстрым способом получать доступ к элементам этого объекта. Есть два способа доступа к элементам. Первый способ это метод .at(i, j, …) или перегруженный оператор () включают проверку индексов и сверку числа измерений при **каждом** обращении; такая защитная логика полезна на этапе отладки, но в плотном численном цикле приводит к значительным потерям времени. Второй способ через методы unchecked<N> and mutable\_unchecked<N>, где N – заранее указанная размерность массива. При использовании этих методов проверки границ и согласованности измерений выполняются однократно до входа в критический участок, а дальнейшее обращение сводится к прямому вычислению адреса в памяти. Остановимся на втором способе, так совершать подобные проверки каждый раз не имеет смысла.

В самом коде, исключения перехватываться не будут: любое непредвиденное состояние (некорректный тип, неверная размерность массива, выход за границы) будет обнаруживаться лишь на уровне предварительной валидации данных в Python. Внутри самого алгоритма лишние проверки и обработчики исключений исключены, поскольку их активация приводит к развёртыванию стека и, следовательно, к значительным накладным расходам.

## **2.3 Выбор структуры данных для хранения дерева**

При проектировании бинарного дерева решений ключевым выбором становится формат хранения узлов. На практике используют две противоположных схемы.

1. Узлы‑структуры со связями‑указателями (связный список).

Каждый узел содержит значения порога, индексы признаков и два указателя left, right на дочерние структуры. Такой дизайн интуитивно ясен и упрощает динамические операции — например, инкрементальное добавление веток или «глубинную» обрезку: достаточно аллоцировать новый объект и переназначить пару ссылок, не затрагивая остальное дерево

1. Вектор (массив) с неявными отношениями индексов.

Альтернативно дерево хранят как один динамический массив структур, а родственные связи кодируют правилами: для узла с индексом *i* левый потомок располагается в 2\*i + 1, правый — в 2\*i + 2. Либо например узел с индексом i, хранит в своей структуре позиции в массиве левого и правого ребенка.

Способ хранения необходимо выбрать, с учетом того, чтоб минимизировать и построение дерева, и время предсказания. С учетом поставленной задачи, первый способ является не выгодным, так как обход сверху вниз превращается в «pointer chasing»: при каждой итерации процессор вынужден загружать случайную страницу памяти, что порождает частые кэш‑промахи. Во втором способе все узлы будут лежать рядом с друг другом. Исходя из этого, благодаря хранению вторым способом мы сможем минимизировать время работы предсказания модели.

## **2.4 Базовый способ подсчета критериев**

В качестве первого алгоритма подсчета наших критериев назовем его базовым или baseline. Основная идея будет заключаться, в том что в базовой реализации мы будем хранить массив best\_splits, состоящий из выбранных сплитов на предыдущем шаге и для вычисления F(S,R) для нового сплита, мы будем брать все сплиты из этого массива и заново переподсчитывать вероятности. Такой подход предельно прост и однозначно корректен, поскольку на каждой итерации использует актуальные распределения и не накапливает числовых ошибок; он удобен для пошаговой отладки, но имеет существенные издержки: частоты пересчитываются для всех узлов, что приводит к избыточным проходам по данным. Этот способ реализации скорее будет проверять другие методы, чем быть основным выбором.

## **2.5 Динамический подсчет критериев**

На самом деле подсчет частот в разбиении слева и справа можно реализовать динамически следующим образом. Для каждого упорядоченного признака храним, два массива — частоты классов слева и справа от текущего порога. Инициализируем правый массив числами по всему узлу, левый — нулями, затем, проходя по отсортированным объектам, при каждом сдвиге порога инкрементируем компоненту класса в левом массиве и декрементируем ту же компоненту в правом. Таким образом, условные вероятности обновляются без повторного сканирования выборки.

Заметим, что помимо вероятностей, сама формула подсчета критерия может считаться динамически. Предположим, что нам необходимо поделить наш узел еще на два узла, это означает, что мы хотим перейти от сплита к сплитам и

Предполагаем, что на прошлом шагу нам был известен значения критерия до разделения:

При разделении мы хотим посчитать нашу формулу уже с новыми сплитами и

- parent\_criteria + child\_left\_criteria + child\_right\_criteria

Таким образом в нашей программме мы можем держать глобальную переменную которая будет хранить , обновлять её если нашли новое разбиение. Также не забывать записывать parent\_criteria, child\_left\_criteria, child\_right\_criteria, чтоб на следующей итерации легко было пересчитывать. Все выше описанное, динамический подсчет частот и динамический расчет данной формулы, составляет решение, которое мы будем называть динамическим или dynamic

## **2.6 Остановка разбиения дерева решений**

Для обеспечения сопоставимости результатов самостоятельной реализации дерева решений и эталонного класса DecisionTreeClassifier [6] из библиотеки scikit-learn в обоих подходах должны использоваться идентичные критерии остановки роста дерева. В качестве базовых настроек в своей реализации я ввожу три широко применяемых гиперпараметра:

1. **max\_depth** — предельная глубина дерева. Определяет максимальное число последовательных сплитов от корня до листа; позволяет контролировать сложность модели и предотвращать её чрезмерный рост.
2. **min\_samples\_split** — минимальное число образцов в узле, необходимое для его дальнейшего разбиения. Если в узле меньше объектов, чем задано этим значением, сплит не выполняется, и узел становится листом.
3. **min\_samples\_leaf** — минимальное число образцов, которое должно оказаться в каждой из двух дочерних ветвей после разбиения. Гарантирует, что ни один из вновь образуемых листьев не будет «слишком маленьким», что препятствует переобучению на выбросах и шуме.

Кроме жёстких ограничений на глубину и размер узлов, современные реализации деревьев решений вводят дополнительный параметр, контролирующий минимальный вклад (уменьшение) меры неоднородности при каждом разбиении. В scikit-learn это параметр min\_impurity\_decrease, который требует, чтобы прирост был не менее заданного порога.

В собственной реализации я также ввожу аналогичный параметр min\_impurity\_decrease: это гарантирует, что дерево «останавливается» не только по размерам или глубине, но и когда дальнейшее разбиение не приносит статистически значимого улучшения качества. Такой критерий позволяет избежать чрезмерного ветвления на малых остаточных приростах, сохраняя модель компактной

## **2.7 Удобность создаваемого интерфейса**

При разработке пользовательского интерфейса исходил из задачи максимально упростить интеграцию собственного C++-модуля в привычный workflow специалистов по машинному обучению. Во-первых, я сделал так, чтобы все данные на входе и выходе воспринимались как стандартные numpy.ndarray: никаких специальных обёрток или промежуточных конвертаций — разработчику достаточно передать двумерные массивы признаков и вектор меток в привычном формате. Это позволяет сразу использовать класс дерева в любом скрипте, где уже задействован NumPy.

Во-вторых, была полностью сохранена сигнатура методов обучения и предсказания: .fit(X\_train, y\_train) и .predict(X\_test) названы так же, как в scikit-learn. Такой подход снимает все лишние вопросы: человек, знакомый со DecisionTreeClassifier из sklearn, вступает в работу с модулем без фазы «приучения» к новому API и сразу видит знакомые параметры и соглашения по передаче данных. Это решение действительно снижает время привыкания программиста, поскольку в интернете очень многой фреймворков, имеющие разную сигнатуру, что действительно становится сложно запомнить для каждого фреймфорка свою сигнатуру функций

Важным элементом удобства стали автодокументированные подсказки при инициализации объекта дерева. С помощью возможностей PyBind11 было обеспечено появление полного списка гиперпараметров (глубина, минимальный размер узла, критерии разбиения, порог по уменьшению impurity и др.) в подсказках IDE или в режиме быстрой справки Jupyter.

Поскольку в данной работе используется новые введенные критерии, то также в программме была добавлена возможность выбирать критерий построения дерева. Эта поддержка была реализована через параметр type\_function, в который достаточно передать значение от 1 до 4, в зависимости какую функцию разбиения хотим использовать.

## **2.8 Бенчмарк полученных алгоритмов**

В исследовании были сопоставлены три реализации алгоритма построения бинарного дерева решений: наивная базовая версия с полным пересчётом частот на каждом кандидате сплита, оптимизированная динамическая версия, в которой счётчики левой и правой частей — а вместе с ними и значение критерия F(S,R) — обновляются инкрементно, а также промышленная реализация DecisionTreeClassifier из библиотеки scikit‑learn.

Эксперимент для подсчета среднего времени построения дерева был осуществлен следующим образом. Для каждого значения n из набора {500,1000, 1500,…,5000} генерировалось num\_datasets случайных матриц признаков размера n×n (N=V) с равномерными вещественными значениями, целевой вектор формировался случайным образом из 21 классов. На каждом таком датасете измерялось время полного обучения дерева, затем вычислялось среднее значение по num\_datasets. Результаты можно увидеть на рисунке 2, по оси X — величина n; по оси Y — среднее время построения (сек).

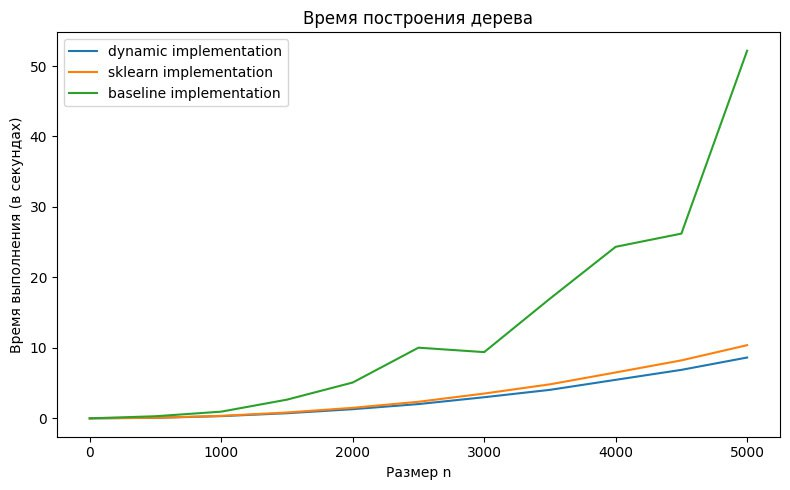


Рисунок 2. Время построения дерева для разных реализацией

Рисунок 2 демонстрирует резкое ухудшение производительности базового алгоритма: начиная примерно с n≈1500n растёт значительно быстрее, и к n=5000 превышает 50 секунд. Напротив, динамическая реализация и дерево из scikit‑learn показывают себя хорошо, причём динамическая реализация стабильно опережает scikit‑learn на 5–15 %. Полученные результаты подтверждают эффективность динамического пересчёта: он снижает затраты на порядок по сравнению с наивным подходом и обеспечивает скорость, сравнимую либо чуть лучшую, чем у зрелой библиотеки scikit‑learn, сохраняя при этом корректность алгоритма.

В рамках эксперимента по оценке времени предсказания дерева решений была проведена серия измерений. Для этого фиксировалась обучающая выборка размером 2000 × 2000, на которой обучались две модели: реализация из библиотеки sklearn с критерием gini и пользовательская реализация, основанная на собственной метрике 1. После обучения для каждого из алгоритмов генерировались тестовые выборки размером n×2000, где n варьировалось от 500 до 5000 с шагом 500. На каждой из этих выборок измерялось среднее время выполнения предсказания, что позволило оценить масштабируемость обеих реализаций в зависимости от объёма входных данных.

Изображение выглядит как текст, линия, График, снимок экрана

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 3. Время предсказания для разных реализаций

На полученном графике можно наблюдать устойчивый рост времени предсказания с увеличением размера тестовой выборки у обеих моделей. При этом реализация sklearn демонстрирует меньшие значения времени, что объясняется высокой степенью оптимизации её внутренней архитектуры. Тем не менее, разница между двумя подходами остаётся умеренной

## **2.9 Разные способы компиляции**

При компиляции C++‑модуля, который мы подключаем к Python через pybind11, существует несколько режимов сборки: именно от них зависит, насколько быстро будет работать итоговый код и насколько удобно его отлаживать. В основе различий лежит флаг оптимизации компиляторах [7], изменяя его, можно экспериментально добиться лучшего соотношения скорости выполнения и времени сборки.

Существуют следующие флаги оптимизации:

1. **-O0 (Debug)**  Все оптимизации отключены, что обеспечивает полную сохранность отладочной информации и минимальное время компиляции, однако приводит к низкой производительности получаемого кода; режим рекомендуется исключительно для пошаговой диагностики.
2. **-O2**  Включает агрессивный инлайн функций, развёртывание небольших циклов и улучшенное планирование инструкций. Данный уровень является стандартным выбором для быстрой сборки и даёт существенный прирост производительности при умеренном увеличении времени компиляции.
3. **-O3 (Release)**  Расширяет набор оптимизаций уровня -O2, добавляя векторизацию, инлайнинг крупных функций и специализированные преобразования счётных циклов. Обеспечивает максимальную скорость выполнения, но удлиняет процесс сборки и может увеличить размер итогового исполняемого файла.

Далее планируется провести дополнительный эксперимент: для каждого уровня оптимизации будет 50 раз сгенерирован датасет фиксированного размера, после чего будет вычислено среднее время построения дерева для каждого флага оптимизации. Сгенерированные датасеты имеют следующие параметры: число объектов 10000, число признаков 100, число лейблов 21. Результаты такого эксперимента вы можете видеть в таблице 1.

Таблица 1. Время построения дерева в зависимости от флагов оптимизации

|  |  |
| --- | --- |
| Флаги оптимизации | Время построения дерева (в секундах) |
| -O0 | 2.91 секунды |
| -O2 | 0.47 секунды |
| -O3 | 0.47 секунды |

Как можем заметить, сборка с флагом -O3 более приоритетнее, так как время построения дерева сокращается в 6 раз, как минимум по сравнению с флагом -O0.

## **2.10 Генерирование синтетических данных**

Для проведения точных экспериментов классификационнных алгоритмов, необходимо данные, которые могли бы варьироваться в зависиимости от разных параметров. Для этого подойдут гауссовы кластеры.В статье [8], говорится как построить гауссовы кластеры, причем их расположение относительно друг друга можно менять благодаря параметру . Причем также можно изменять размеры датасета, а именно менять число объектов N, число признаков V и число кластеров L.

Алгоритм генерации выглядит следующим образом:

1. На начальном этапе инициализируем наши параметры N, V, L, . Причем для каждого из кластеров определяем его размер. В случае если N mod L != 0, то нельзя во всех кластерах сделать равное число объектов, соотвественно у последнего кластер будет на N mod L объектов больше
2. Для каждого кластера l формируется вектор‑центр длины V. Его компоненты берутся из равномерного распределения на интервале (−1,  1−). Причем из интервала от 0 до 1.
3. Для каждого центра случайно задаётся вектор стандартных отклонений , где каждое ​ берётся из равномерного отрезка [0.05,  0.10]
4. Для каждого кластера l, генерируется векторов из нормального распределения N(0,) по каждой координате. После прибавляется к каждому вектор центр соотвествуюшего кластера . Полученная матрица для кластера l будет иметь размеры X V.
5. Проделав работу так для каждого класстера, финальный датасет состоит из конкатенаций соотвествующих массивов

Попробуем нарисовать данные кластеры с помощью библиотеки matlotplib. Для этого фиксируем значения N = 2000, L = 3, V = 2, при этом первые данные также будут построены при = 0.1. Распределение данных на плоскости можно увидеть на рисунке 4.

Изображение выглядит как снимок экрана, диаграмма, График

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 4. Расположение кластеров при N = 2000, L = 3, V = 2, = 0.1

Исходя из расположения кластеров на рисунке 1, можем заметить, что центры кластеров отстоят друг от друга на достаточно большие расстояния по обоим признакам, что приводит к практически полному разделению точек на уровне первого сплита. В таких условиях алгоритм дерева решений легко определяет порог независимо по X1​ или X2, сразу получая три почти однородных ветви: глубина дерева при этом остаётся минимальной, а accuracy близка к 100 %.

Теперь давайте нарисуем, расположение кластеров при почти таких же параметрах N = 2000, L = 3, V = 2, но с другим значением = 0.7. Расположение кластеров при таких параметрах показано на рисунке 5.

Изображение выглядит как снимок экрана, Красочность, График

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 5. Расположение кластеров при N = 2000, L = 3, V = 2, = 0.7

На рис. 5 показан тот же набор при = 0.7. Как можем заметить из рисунка, центры сближены, и ни один порог по одному признаку не может полностью отделить все три класса за один шаг. Здесь дерево решений столкнётся с необходимостью строить более глубокую и разветвлённую структуру: на первом уровне сплит даст только частичное разделение, а чтобы снизить impurity потребуются дополнительные узлы. В результате глубина модели вырастет, а при строгом ограничении глубины или минимального размера листа доля неверных классификаций (особенно около границ перекрытия) неизбежно увеличится.

# 3. Проведение эксперимента и анализ качества модели

## **3.1 Основные метрики эксперимента**

В рассматриваемой работе задача сводится к обучению модели на размеченных данных с целью корректного отнесения каждого объекта к одному из заранее заданных классов. Для количественной оценки качества классификации принято использовать ряд метрик, отражающих различные аспекты работы алгоритма.

Одним из наиболее интуитивно понятных и широко применяемых показателей является **точность** (accuracy). Она определяется как доля правильно классифицированных образцов во всём тестовом наборе:

Также необходимо рассмотреть еще одну метрику классификации. В качестве второй метрики классификации выберем ARI(Adjusted Rand Index).

Скорректированный индекс Рэнда (ARI [9], Adjusted Rand Index) представляет собой меру сходства двух разбиений одного и того же множества объектов, скорректированную на уровень совпадений, ожидаемых случайно. В основе ARI лежит сравнение пар объектов: учитываются все пары, которые либо одновременно попали в один и тот же кластер в обоих разбиениях, либо в разных кластерах в обоих. При этом ARI вычитает из фактического числа таких «согласованных» пар ожидаемое число совпадений при случайном распределении объектов и нормирует результат так, что значения индекса лежат в диапазоне от –1 до +1. Значение +1 означает полное совпадение разбиений, то есть идеальную классификацию; значение 0 соответствует уровню согласованности, который статистически равен случайному; отрицательные значения указывают на то, что два разбиения согласуются ещё хуже, чем случайное распределение.

Изображение выглядит как текст, Шрифт, белый, линия

Контент, сгенерированный ИИ, может содержать ошибки.

Данная комбинация метрик, обеспечивает полный анализ работы нашего алгоритма.

## **3.2 Результаты эксперимента**

После выбора наших метрик, необходимо сгенерировать синтетические данные, как это сделать было описано ранее в главе 2.10. Проведу соотвественно описание как мы хотим генерировать данные, с какими параметрами и как будем мерить точность:

1. Генерация данных. Необходимо для каждого набора (V, L, ) генерировать число датасетов равному 50, сами параметры могут пробегать следующие значение V равен 15 или 25, L равен 7 или 15, равен 0.5, 0.75 или 0.85
2. Подсчет метрик. Зафиксировав (V, L, ), начинаем генерировать 50 датасетов, для каждого датасета, мы делим его на тренировочные данные и тестовые, строим дерево на тренировочных данных, а на тестовым считаем наши критерии accuracy и ARI
3. Вывод информации. Необходимо вывести таблицу в следующем формате, по столбцам у нас будут типы критериев, которые мы рассматриваем, то есть 1, 2, 3, 4. По строкам у нас будет располагаться комбинации (V, L, ), а на пересечении будет располагаться среднее значение и стандартное отклонение для каждого из наших выбранных критериев

Стоит отметить, что в генерировать каждый датасет и запускать алгоритм не очень выгодно с точки зрения потраченного времени. Поэтому напишем соотвествующий скрипт, который реализует соотвествующий эксперимент. Планируется также провести сравнение собственной реализации дерева решений с эталонным алгоритмом из библиотеки scikit-learn, DecisionTreeClassifier. При этом важно отметить, что сравнение будет осуществляться не только по умолчанию, но и с учётом различных доступных критериев разбиения, реализованных в данном классе.

В частности, будут рассмотрены следующие встроенные критерии scikit-learn:

1. gini — классический критерий Джини, применяемый по умолчанию в алгоритме CART.
2. entropy — основан на информации Шеннона, реализует принцип максимального информационного выигрыша (используется в C4.5).

Каждый из этих критериев будет использован при построении дерева DecisionTreeClassifier, чтобы в дальнейшем сравнить его поведение и точность с четырьмя функциями разбиения, реализованными в рамках настоящей работы. Это позволит не только оценить абсолютное качество предложенных методов, но и поставить их в контекст современных индустриальных решений. В таблице 2, приведены результаты эксперимента для метрики accuracy для новых критериев

Таблица 2. Accuracy построенных критериев в формате mean/std

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| комбинации | = 0.5 | | | |  | | | | |  | | | | |
|  | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 1 | | 2 | 3 | 4 |
| L = 7,  V = 15 | 0.88/  0.12 | **0.93/**  **0.10** | 0.92/  0.09 | 0.89/  0.10 | 0.92/  0.06 | **0.93/**  **0.06** | **0.93/**  **0.06** | 0.92/  0.06 | 0.79/  0.06 | | 0.80/  0.06 | 0.80/  0.06 | 0.80/  0.06 |
| L = 7,  V = 25 | 0.82/  0.12 | **0.94/**  **0.09** | 0.89/  0.13 | 0.86/  0.10 | 0.93/  0.07 | **0.94/**  **0.06** | 0.94/  0.07 | 0.94  /0.06 | 0.83/  0.05 | | 0.84/  0.05 | 0.84  /0.05 | 0.84  /0.05 |
| L = 15,  V = 15 | 0.59/  0.09 | 0.63/  0.11 | 0.62/  0.11 | **0.65**  **/0.10** | 0.67/  0.07 | 0.68/  0.08 | 0.68/  0.08 | **0.68/**  **0.08** | 0.57/  0.05 | | 0.57  /0.05 | 0.57  /0.05 | 0.57/  0.05 |
| L = 15,  V = 25 | 0.55/  0.08 | 0.61/  0.09 | 0.61/  0.09 | **0.61/**  **0.08** | 0.68/  0.08 | 0.69/  0.08 | 0.69/  0.08 | **0.70/**  **0.07** | 0.58/  0.05 | | 0.59/  0.05 | 0.59/  0.05 | 0.59/  0.05 |

Таблица 3. Accuracy критериев из библиотеки sklearn в формате mean/std

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Комбинации | α = 0.5 | | α = 0.75 | | α = 0.85 | |
|  | gini | entropy | gini | entropy | Gini | entropy |
| L = 7,  V = 15 | 0.88/  0.12 | 1.00  /0.00 | 0.92/  0.06 | 0.97  /0.02 | 0.79/  0.06 | 0.81/  0.05 |
| L = 7,  V = 25 | 0.82/  0.12 | 1.00/  0.00 | 0.93/  0.07 | 0.98/  0.01 | 0.83/  0.05 | 0.86/  0.03 |
| L = 15,  V = 15 | 0.59/  0.09 | 0.93/  0.05 | 0.67/  0.07 | 0.79/  0.05 | 0.57/  0.05 | 0.58/  0.04 |
| L = 15,  V = 25 | 0.55/  0.08 | 0.96  /0.04 | 0.68/  0.08 | 0.83/  0.04 | 0.58/  0.05 | 0.62/  0.03 |

Таблица 4. ARI построенных критериев в формате mean/std

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Комбинации | α = 0.5 | | | | α = 0.75 | | | | α = 0.85 | | | |
|  | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 |
| L = 7, V = 15 | 0.87/  0.14 | **0.93/**  **0.09** | 0.93/  0.11 | 0.91  /0.11 | 0.86/  0.10 | **0.87/**  **0.09** | 0.87/  0.09 | 0.86/  0.10 | 0.60/  0.08 | 0.60/  0.08 | 0.60/  0.08 | 0.61/  0.07 |
| L = 7, V = 25 | 0.79/  0.13 | **0.93/**  **0.11** | 0.84/  0.14 | 0.82/  0.11 | 0.91/  0.07 | **0.91/**  **0.06** | 0.91/  0.06 | 0.91/  0.07 | 0.66/  0.06 | 0.65/  0.07 | 0.65/  0.07 | 0.66/  0.07 |
| L = 15, V = 15 | 0.51/  0.11 | 0.57/  0.11 | 0.57/  0.12 | **0.61/**  **0.13** | 0.57/  0.09 | 0.59/  0.08 | 0.58/  0.08 | 0.58/  0.09 | 0.36/  0.05 | 0.36/  0.05 | 0.36/  0.05 | 0.36/  0.05 |
| L = 15, V = 25 | 0.49/  0.12 | 0.57/  0.14 | 0.55/  0.12 | 0.54/  0.12 | 0.59/  0.09 | 0.59/  0.10 | 0.59/  0.10 | **0.60/**  **0.10** | 0.38/  0.06 | 0.38/  0.06 | 0.38/  0.06 | 0.39/  0.06 |

Таблица 5.ARI критериев из библиотеки sklearn в формате mean/std

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Комбинации | α = 0.5 | | α = 0.75 | | α = 0.85 | |
|  | gini | entropy | gini | entropy | gini | entropy |
| L = 7, V = 15 | 0.87/  0.14 | 0.99/  0.01 | 0.86/  0.10 | 0.92  /0.03 | 0.60/  0.08 | 0.62/  0.06 |
| L = 7, V = 25 | 0.79/  0.13 | 0.99/  0.01 | 0.91/  0.07 | 0.94/  0.03 | 0.66/  0.06 | 0.68/  0.06 |
| L = 15, V = 15 | 0.51/  0.11 | 0.92/  0.05 | 0.57/  0.09 | 0.68/  0.06 | 0.36/  0.05 | 0.38/  0.05 |
| L = 15, V = 25 | 0.49/  0.12 | 0.93/  0.04 | 0.59/  0.09 | 0.75/  0.05 | 0.38/  0.06 | 0.43/  0.04 |

## **3.3 Анализ полученных результатов**

Как показали численные эксперименты (см. табл. 2-5), критерий 1 демонстрирует совпадающие с индексом Джини значения всех исследованных показателей качества (accuracy, ARI) во всех комбинациях параметров (L,V,α). Более того, построенное на его основе дерево решений оказалось структурно тождественным дереву, сформированному классическим алгоритмом CART с критерием gini: идентичны не-только набор выбранных признаков на каждом уровне, но и пороговые значения, а также состав и количество листовых узлов. Таким образом, эксперименты эмпирически подтвердили теоретический вывод о полной эквивалентности критерия 1 и индекса Джини как в плане оптимизируемой целевой функции, так и в отношении результата построения дерева.

Показатели в табл. 2 и 4 демонстрируют нетипичную для решающих деревьев картину: при возрастании α от 0.5 до 0.75 точность всех построенных критериев возрастает, тогда как обычно ожидалось бы ухудшение классификации. Причина связана с тем, как формируются синтетические данные и как наши критерии реагируют на данные.

Поскольку при генерации для каждого кластера l, мы формируем вектор центров из интервала (−1,  1−), то при = 0.5, вероятность, что у нас получится какой – то класс отдельный от других (остров), больше чем при = 0.75. Данный факт можно заметить при генерации данных с параметром = 0.5 и при просмотре распределения относительно признака 24.

Изображение выглядит как текст, снимок экрана, число, линия

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 6. Распределение лейблов относительно признака под номером 24

В этом случае на рисунке 6, мы видим, что класс 8 очень легко отделяется если по признаку 24 задать порог 0.28. Это и делает критерий 1(критерий Gini). Для полного дальнейшего анализа давайте построим дерево с критерием 1, при глубине дерева = 4 и при N = 2000, = 0.5, L = 15, V = 25.

Представление такого дерева можно видеть на рисунке 7.



Рисунок 7. Дерево решений построенное для критерия 1 при N = 2000,

= 0.5, L = 15, V = 25, max\_depth = 4

На представленном дереве на рисунке 7, корневой узел сразу использует признак 24 в качестве порога: этот атрибут настолько хорошо отделяет один из 15 классов, что критерий 1 моментально «вычищает» соответствующий под-кластер. После столь резкого первого деления последующие уровни уже располагают лишь тремя позициями по каждому пути, и этого оказывается недостаточно, чтобы полностью развести оставшиеся метки: в итоге сформировано 10 терминальных узлов, предсказывающих всего 9 уникальных классов (6 меток так и не получили собственного листа). Картина явно демонстрирует структурное ограничение: при максимальной высоте 4 дерево способно иметь не более 16 листьев, но значительную их часть «съедает» доминирующая ветвь, тогда как остальные классы вынужденно «слипаются» в неоднородных листьях. Повышение max\_depth хотя бы до 5–6 увеличило бы потенциальное число терминальных узлов до 32–64, позволило бы выделить оставшиеся кластеры в отдельные листья и, следовательно, повысило бы классификационную точность модели.

При = 0.5 диапазон генерации центров кластеров сужается до интервала

(–0,25, 0,25), поэтому даже после добавления гауссового шума распределение каждого признака концентрируется вблизи нулевого значения. В результате кластеры оказываются существенно перекрытыми, и на первом уровне разбиения никакой порог по отдельному признаку не обеспечивает достаточного снижения критерия 1 для выделения «чистого» листа. В качестве подтверждения, показан рисунок 8, на котором изображено распределение лейблов относительно признака 9 при = 0.75

Изображение выглядит как текст, снимок экрана, число, линия

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 8. Распределение лейблов относительно признака под номером 9

В этом случае на рисунке 8, мы видим, что сложно выделить 1-2 чистых листа на первом шаге. Для полного анализа построим дерево использующее первый критерий на синтетических данных с параметрами N = 2000, = 0.75, L = 15, V = 25 при глубине дерева = 4. Получившиеся дерево можно видеть на рисунке 9.

Изображение выглядит как текст, диаграмма, План, Параллельный

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 9. Дерево решений построенное для критерия 1 при N = 2000,

= 0.75, L = 15, V = 25, max\_depth = 4

Дерево решений, построенное с использованием критерия 1 и α = 0.75, демонстрирует более глубокую и сбалансированную структуру: полностью «чистые» узлы не появляются уже на первых уровнях, и алгоритм продолжает разбиения до получения 16 терминальных листов. За счёт такого детального ветвления модель точнее разделяет выборку, что приводит к измеримому росту общей классификационной точности.

Тем не менее, ограничение по α и глубине не позволяет устранить остаточную неоднородность в самых глубоких ветвях: в одном из конечных узлов сохраняется заметный уровень примесей классов, указывающий на недостающую глубину для окончательного деления.

Также интересно отметить из полученных результатов, что критерий энтропии даже при α = 0.5, формирует сбалансированную и симметричную структуру: ни один из узлов не становится «чистым» уже после первого разбиения, поэтому алгоритм продолжает ветвление вплоть до глубины 4. Этой глубины оказывается достаточно, чтобы во всех терминальных узлах достигалась полная однородность классов, что свидетельствует о более равномерном распределении информации по уровням дерева.

При небольшом числе классов сводные данные таблиц 2 и 4 наглядно демонстрируют преимущество второго критерия над остальными. Так, при конфигурации L=7,  V=15 и = 0.5, данный критерий достигает в среднем 93 % accuracy и 0,93 ARI, что является лучшим результатом среди критериев 1–4. Аналогичная картина наблюдается и при L=7,  V=25 при той же α=0,5, второй критерий демонстрирует 94 % accuracy и 0,93 ARI, вновь уверенно опережая альтернативы.

При увеличении числа классов (то есть в более «трудной» постановке задачи) четвёртый критерий демонстрирует наилучшие результаты по сравнению с остальными тремя. Так, при конфигурации L=15,  V=15 и α=0.5, он показывает в среднем 65% accuracy и 0.61 ARI, обходя по этим показателям критерии 1-4. Четвертый критерий также показывает лучшие результаты среди критериев 1-4 и для конфигурации L=15,  V=25 и α=0.5, добиваясь 61% accuracy при меньшем стандартном отклонении 0.08.

При α = 0.85 наши критерии уже действительно не могут справиться, поскольку кластера максимально близки друг другу и качество для всех критериев примерно одинаково между собой.

Заметим, что все четыре критерия демонстрируют заметно более низкие результаты, чем стандартная мера entropy. Даже при использовании оптимального α=0,75 средняя точность наших моделей уступает эталонам, как продемонстрировано в таблицах 2 – 5. Как было ранее показано причина такому — ограниченная «высота» дерева, чтобы достичь сопоставимой чистоты листьев, нашим критериям требуется глубина, превышающая ту, при которой уже формируются чистые узлы у деревьев, построенных по entropy. При дальней­шем увеличении глубины действительно наблюдается улучшение — узлы постепенно «доочищаются» и разница в метриках сокращается.

## **3.4 Эксперимент на известных датасетах**

Для полноценной оценки предлагаемых критериев целесообразно расширить эксперимент за пределы искусственно сгенерированных данных и проверить их работу на классических датасетах.

В качестве первого датасета можно взять датасет Ирисов [10]. Состоящий из 150 образцов, каждый из которых описывается четырьмя числовыми признаками — длиной и шириной чашелистика и лепестка в сантиметрах. Целевой признак — вид растения. Необходимо отнести образец к одному из трёх классов: Iris setosa, Iris versicolor или Iris virginica.

Для обеспечения сопоставимости результатов мы зафиксировали параметры остановки дерева, максимальную глубину = 3, минимальное число объектов для разбиения узла = 4 и минимальное число объектов в листе = 2. В ходе эксперимента все четыре разработанных критерия наряду со стандартными Gini, Entropy продемонстрировали идентичные показатели: точность классификации составила 0.96666, а ARI - 0.8981.

Для дальнейшей проверки критериев возьмём популярный набор данных Wine. Этот датасет включает 178 образцов вина трёх различных сортов. Каждый экземпляр описывается 13 числовыми признаками: содержанием алкоголя, яблочной кислоты, зольных веществ и щёлочности золы, концентрацией магния, общим содержанием фенолов, флавоноидами и нефлавоноидными фенолами, проантоцианинами, интенсивностью цвета, показателем оттенка, отношением OD280/OD315 для разведённых образцов и содержанием пролина. Также проделаем эксперимент с фиксированными значением критерий остановок, максимальную глубину = 3, минимальное число объектов для разбиения узла = 5 и минимальное число объектов в листе = 2. Результаты по метрикам accuracy и ARI представлены в таблице 6.

Таблица 6. Значения метрик на датасете Wine

|  |  |  |
| --- | --- | --- |
| Критерий | Accuracy | ARI |
| 1 | 0.89 | 0.67 |
| 2 | 0.97 | 0.92 |
| 3 | 1.0 | 1.0 |
| 4 | 0.97 | 0.92 |
| Sklearn gini | 0.89 | 0.67 |
| Sklearn entropy | 1.0 | 1.0 |

На наборе Wine критерий 3 подтвердил свою наивысшую эффективность, продемонстрировав идеальные метрики (Accuracy = 1, ARI = 1), полностью повторяя результаты стандартного критерия Entropy. Критерии 2 и 4 также показали очень высокие показатели (Accuracy = 0,97, ARI = 0,92), существенно опережая базовый Gini (0,89/0,67) и критерий 1 (0,89/0,67)

Самое интересное почему же ошиблись критерий 2 и критерий 4. Оказывается для них было построено почти идентичные деревья. Начальное разбиение на тренировочных данных было слишком различным, в силу этого 2 и 4 критерии начали по другому разделять, и пытаться разбить 3 класс максимально чисто (в силу того факта что вероятность третьего класса мала), поэтому не удалось на данной глубине найти идеального разбиения

# Заключение

В рамках данной выпускной квалификационной работы была поставлена и решена задача построения и анализа новых критериев разбиения для алгоритма дерева решений. Целью исследования являлась разработка метрик и их экспериментальное сравнение с существующими решениями, представленными в библиотеке scikit-learn.

Были реализованы четыре критерия неопределённости, один из которых воспроизводит поведение классического CART, а другой — статистический подход CHAID, основанный на χ²-статистике. Два оставшихся критерия предложены как новые модификации с использованием альтернативных шкалирующих факторов. Все метрики были реализованы в производительном модуле на C++ с использованием библиотеки pybind11 и снабжены интерфейсом, совместимым с sklearn.

Для объективной оценки качества построенных моделей были проведены обширные эксперименты на синтетических данных с контролируемыми параметрами (число классов, признаков, степень размытости α). В качестве основных показателей качества использовались метрика точности (accuracy) и индекс согласованности ARI (Adjusted Rand Index). Полученные результаты показали, что предложенные критерии, особенно критерии 2 и 4, демонстрируют стабильное поведение в условиях высокой сложности задачи и во многих случаях выше по качеству чем метрика gini из sklearn.

Также было проведено сравнение по времени обучения и предсказания между реализованными методами и стандартной реализацией дерева решений из sklearn. Несмотря на незначительное увеличение времени предсказания, предложенная реализация показывает меньшее время построения дерева.

Таким образом, проведённое исследование подтверждает, что предложенные метрики могут служить полноценной альтернативой существующим критериям построения деревьев решений и обладают рядом преимуществ в условиях усложнённых многоклассовых задач. Полученные результаты могут быть полезны для дальнейшего развития методов интерпретируемого машинного обучения и оптимизации существующих алгоритмов построения деревьев.

# Приложение

Вся реализация доступна в репозитории <https://github.com/NickSterALPHA/TrofimovTree>, где в папке src сосредоточены исходные файлы на C++ вместе с обёртками на pybind11, обеспечивающими интеграцию с Python. Для удобства сборки проекта предусмотрён скрипт CMakeLists.txt, который автоматически конфигурирует и компилирует модуль расширения.

В рамках исследования подготовлены два Jupyter-ноутбука. Файл calculation.ipynb отвечает за генерацию синтетических данных и проведение экспериментальных прогонов по всем критериям. Ноутбук paint.ipynb содержит графики для исследования полученных данных

Единственной внешней зависимостью проекта выступает библиотека pybind11, используемая для связывания C++ реализации и Python-интерфейса.

# Список используемой литературы

[1] Breiman L., Friedman J.H., Olshen R.A., Stone C.J. Classification and Regression Trees. — Belmont: Wadsworth International Group, 1984

[2] Quinlan J.R. C4.5: Programs for Machine Learning. — San Mateo: Morgan Kaufmann, 1993.

[3] Green S.B., Salkind N.J. Using SPSS for Windows and Macintosh: Analyzing and Understanding Data. 3rd ed. — Upper Saddle River, NJ: Prentice Hall, 2003

[4] Mirkin B Core data analysis. - Springer, 2019

[5] pybind11 // URL: https://pybind11.readthedocs.io/en/stable/index.html (дата обращения: 03.03.24).

[6] DecisionTreeClassifier // scikit-learn URL: https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html (дата обращения: 02.03.2025).

[7] GCC: список опций командной строки компилятора [Электронный ресурс]. — Режим доступа: https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html (дата обращения: 15 мая 2025 г.).

[8] Inertia-based indices to determine the number of clusters in K-means:an experimental evaluation // [www.researchgate.net](http://www.researchgate.net).

[9] Hubert L., Arabie P. Comparing partitions. // Journal of Classification. — 1985.

[10] Fisher R.A. The use of multiple measurements in taxonomic problems // Annals of Eugenics. — 1936.