Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

“НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО”

Факультет	**Программной Инженерии и Компьютерной Техники**
Направление подготовки (специальность)	**Системное и прикладное программное обеспечение**
Дисциплина	**Системы искусственного интеллекта**

ЛАБОРАТОРНАЯ РАБОТА 6
ОТЧЕТ

Выполнил студент:	**Силинцев Владислав Витальевич (355273)**
Группа:	**P3314**
Преподаватель:	**Болдырева Елена Александровна (157150)**

г. Санкт-Петербург

2025

Содержание

ИНДИВИДУАЛЬНОЕ ЗАДАНИЕ

1. Выбор датасета:
    - Датасет о пассажирах Титаника.
    - Датасет о диабете.
2. Загрузите выбранный датасет и выполните предварительную обработку данных.
3. Получите и визуализируйте (графически) статистику по датасету (включая количество, среднее значение, стандартное отклонение, минимум, максимум и различные квантили).
4. Разделите данные на обучающий и тестовый наборы в соотношении, которое вы считаете подходящим.
5. Реализуйте логистическую регрессию "с нуля" без использования сторонних библиотек, кроме NumPy и Pandas. Ваша реализация логистической регрессии должна включать в себя:
    - Функцию для вычисления гипотезы (sigmoid function).
    - Функцию для вычисления функции потерь (log loss).
    - Метод обучения, который включает в себя градиентный спуск.
    - Возможность варьировать гиперпараметры, такие как коэффициент обучения (learning rate) и количество итераций.
6. Исследование гиперпараметров:
    - Проведите исследование влияния гиперпараметров на производительность модели. Варьируйте следующие гиперпараметры:
        - Коэффициент обучения (learning rate).
        - Количество итераций обучения.
        - Метод оптимизации (например, градиентный спуск или оптимизация Ньютона).
7. Оценка модели:
    - Для каждой комбинации гиперпараметров оцените производительность модели на тестовом наборе данных, используя метрики, такие как accuracy, precision, recall и F1-Score.

Сделайте выводы о том, какие значения гиперпараметров наилучшим образом работают для данного набора данных и задачи классификации. Обратите внимание на изменение производительности модели при варьировании гиперпараметров.

ОТЧЕТ О ХОДЕ ВЫПОЛНЕНИЯ

Выбор датасета

В соответствии с порядковым номером в группе (8 — чётное число) для выполнения работы был выбран датасет о пассажирах Титаника.

Предварительная обработка данных

```
# === Предварительная обработка данных ===
# --- Заполняем пропуски ---
# Age - заполняем медианой
train_df['Age'].fillna(train_df['Age'].median(), inplace=True)
test_df['Age'].fillna(test_df['Age'].median(), inplace=True)

# Embarked - заполняем модой
train_df['Embarked'].fillna(train_df['Embarked'].mode()[0], inplace=True)

# Fare (в тестовых данных есть 1 пропуск) - заполняем медианой
test_df['Fare'].fillna(test_df['Fare'].median(), inplace=True)

# --- Кодирование категориальных признаков ---
train_df = pd.get_dummies(train_df, columns=['Embarked'], prefix='Embarked')
test_df = pd.get_dummies(test_df, columns=['Embarked'], prefix='Embarked')

train_df = pd.get_dummies(train_df, columns=['Sex'], prefix='Sex')
test_df = pd.get_dummies(test_df, columns=['Sex'], prefix='Sex')

train_df = pd.get_dummies(train_df, columns=['Pclass'], prefix='Class')
test_df = pd.get_dummies(test_df, columns=['Pclass'], prefix='Class')

# --- Удаляем ненужные столбцы ---
columns_to_drop = ['Name', 'Ticket', 'Cabin', 'PassengerId']
train_df = train_df.drop(columns=[col for col in columns_to_drop if col in
train_df.columns])
test_df = test_df.drop(columns=[col for col in columns_to_drop if col in
test_df.columns])
```

Статистика по датасету

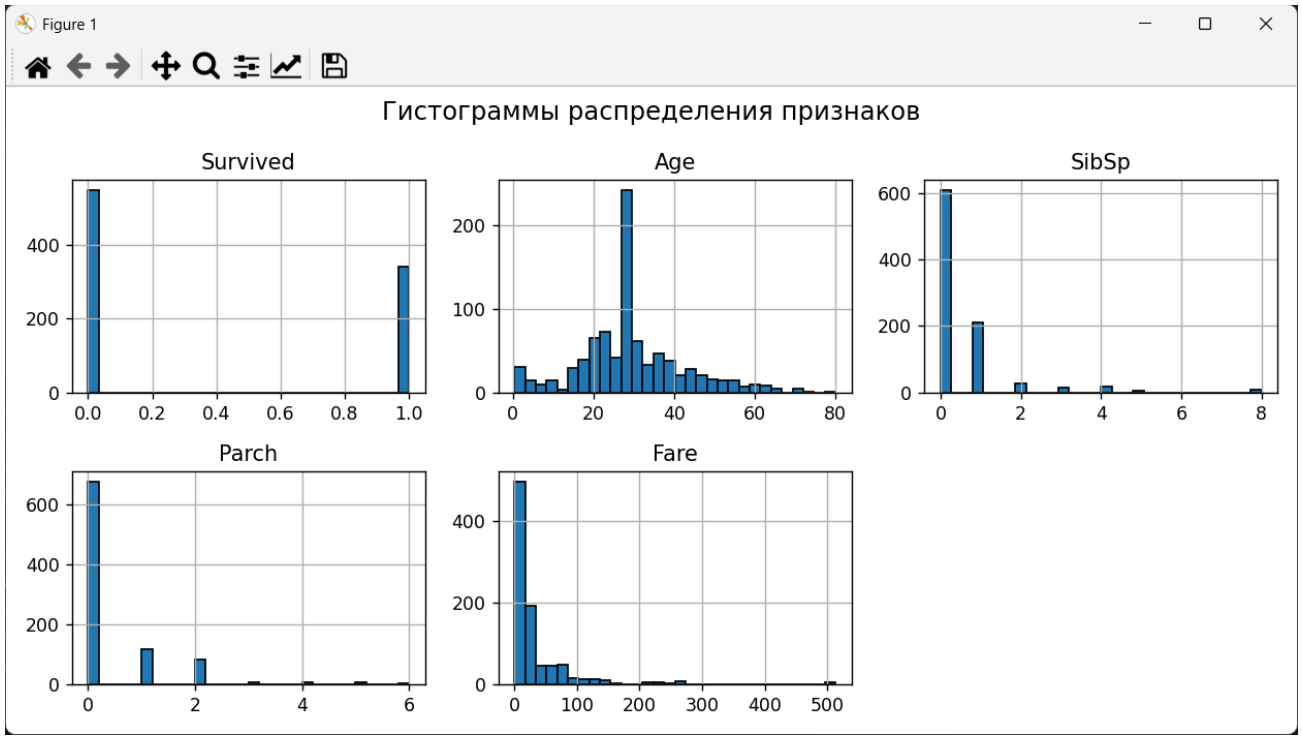

График 1 – Гистограммы распределения.

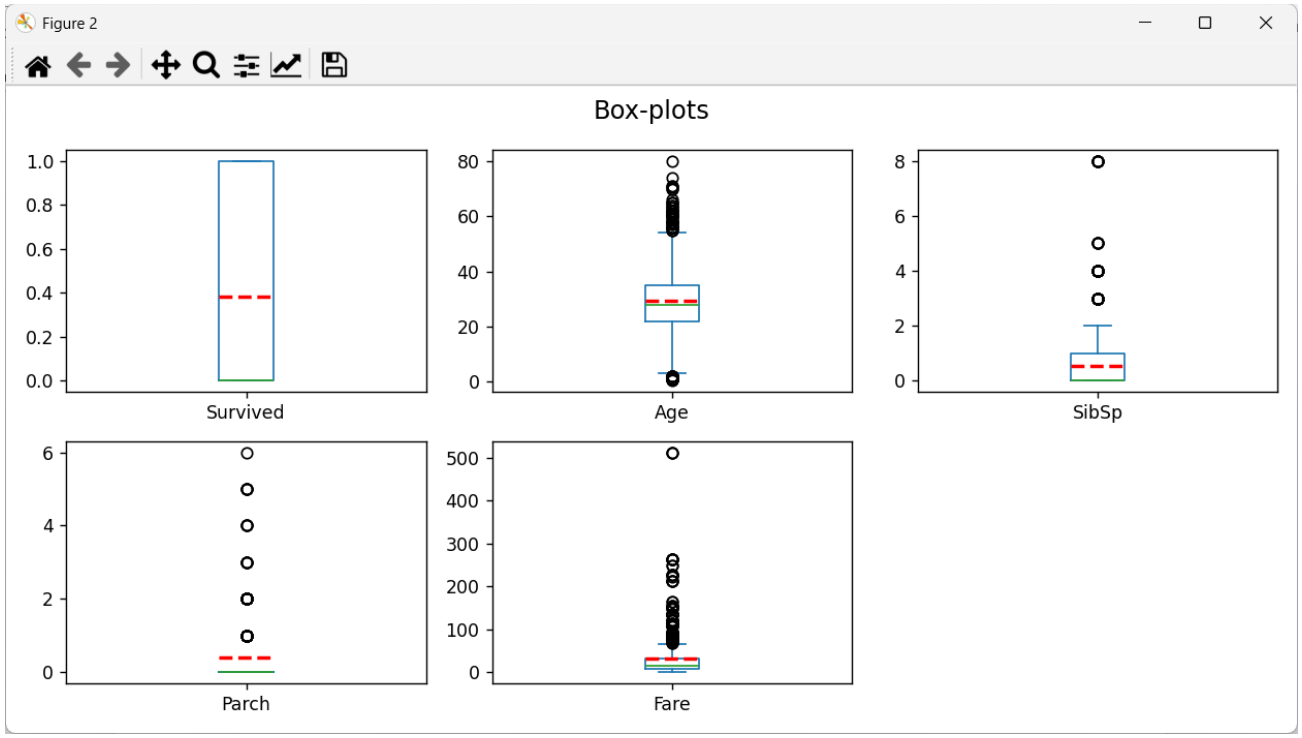

График 2 – Box-plots.

Разделение данных

```
# === Считываем датафреймы ===
train_df = read_csv_file("train.csv")
test_df = read_csv_file("test.csv")
test_result_df = read_csv_file("gender_submission.csv")
```

Реализация логистической регрессии

```
import numpy as np

from optimization_type import OptimizationType


def sigmoid(x):
    """Функция сигмоиды"""
    # Гарантируем, что это numpy array
    x = np.asarray(x, dtype=float)

    # Защита от переполнения
    x = np.clip(x, -500, 500)

    return 1 / (1 + np.exp(-x))


def log_loss(y_true, y_predicted_proba):
    """Функция потерь log loss"""
    n = len(y_true)
    result = 0
    for i in range(n):
        result += y_true[i] * np.log(y_predicted_proba[i]) + (1 - y_true[i]) *
np.log(1 - y_predicted_proba[i])
    result *= -1 / n
    return result


class LogisticRegression:
    """Класс для логистической регрессии"""

    def __init__(self, learning_rate=0.01, n_iterations=1000, opt_type=Optimiza-
tionType.GRADIENT_DESCENT):
        """Конструктор"""
        self.learning_rate = learning_rate
        self.n_iterations = n_iterations
        self.weights = None
        self.bias = None
        self.loss_history = []
        self.opt_type = opt_type

    def fit(self, x_train, y_train):
        """Обучение модели градиентным спуском"""
        x_train = np.array(x_train, dtype=float)
        y_train = np.array(y_train, dtype=float)

        n_samples, n_features = x_train.shape
        self.weights = np.zeros(n_features)
        self.bias = 0
```

```
        print(f"Начинаем обучение методом: {self.opt_type.name}")
        print(f"Скорость обучения: {self.learning_rate} ")
        print(f"Количество итераций: {self.n_iterations}")

        # Выбор метода оптимизации
        if self.opt_type == OptimizationType.GRADIENT_DESCENT:
            self._fit_gradient_descent(x_train, y_train, n_samples)
        elif self.opt_type == OptimizationType.NEWTON_METHOD:
            self._fit_newton_method(x_train, y_train, n_samples)
        else:
            raise ValueError(f"Неизвестный метод оптимизации: {self.opt_type}")

    def _fit_gradient_descent(self, x_train, y_train, n_samples):
        """Обучение градиентным спуском"""
        print_step = self.n_iterations / 5
        for i in range(1, self.n_iterations + 1):
            # Вычисляем линейную комбинацию
            linear_output = np.dot(x_train, self.weights) + self.bias

            # Применяем сигмоиду для получения вероятностей
            y_predicted_proba = sigmoid(linear_output)

            # Вычисление функции потерь
            loss = log_loss(y_train, y_predicted_proba)
            self.loss_history.append(loss)

            # Вычисление градиентов
            error = y_predicted_proba - y_train

            # Вычисляем градиенты (производные)
            dw = (1 / n_samples) * np.dot(x_train.T, error)
            db = (1 / n_samples) * np.sum(error)

            # Обновление параметров
            self.weights -= self.learning_rate * dw
            self.bias -= self.learning_rate * db

            if i % print_step == 0:
                print(f"  Итерация {i:4d}, Loss: {loss:.4f}")

        print(f"  Градиентный спуск завершен")

    def _fit_newton_method(self, x_train, y_train, n_samples):
        """Обучение методом Ньютона"""
        # Добавляем столбец единиц для bias
        x = np.hstack([x_train, np.ones((n_samples, 1))])

        # Инициализируем все параметры вместе
        theta = np.zeros(x.shape[1])  # [w1, w2, ..., wn, bias]

        print_step = self.n_iterations / 5
        for i in range(1, self.n_iterations + 1):
            # Вычисляем линейную комбинацию
            linear_output = np.dot(x, theta)

            # Применяем сигмоиду для получения вероятностей
            y_predicted_proba = sigmoid(linear_output)
```

```
            # Вычисление функции потерь
            loss = log_loss(y_train, y_predicted_proba)
            self.loss_history.append(loss)

            # Градиент (первая производная)
            # Формула: градиент = X^T * (p - y) / n
            gradient = np.dot(x.T, (y_predicted_proba - y_train)) / n_samples

            # Гессиан (вторая производная, матрица)
            # Формула: гессиан = X^T * D * X / n
            # где D = diag(p * (1 - p)) - диагональная матрица
            w = y_predicted_proba * (1 - y_predicted_proba)  # диагональные эле-
менты

            # Быстрый расчет: X^T * diag(W) * X
            hessian = np.dot(x.T * w, x) / n_samples

            # Метод Ньютона: theta_new = theta_old - H^(-1) * градиент
            # Решаем: H * delta = градиент
            delta = np.linalg.solve(hessian, gradient)
            theta -= delta

            # Разделяем обратно на weights и bias
            self.weights = theta[:-1]  # все кроме последнего
            self.bias = theta[-1]  # последний - это bias

            if i % print_step == 0:
                print(f"  Итерация {i:4d}, Loss: {loss:.4f}")

            # Останавливаемся, если loss почти не меняется
            if i > 1 and abs(self.loss_history[-1] - self.loss_history[-2]) <
1e-9:
                print(f"  Сошлось на итерации {i}")
                print(f"  Loss: {loss:.4f}")
                break

        print(f"  Метод Ньютона завершен")

    def predict_proba(self, x_test):
        """Предсказание вероятности принадлежности к классу 1"""

        # Линейная комбинация
        linear_output = np.dot(x_test, self.weights) + self.bias

        # Применяем сигмоиду
        return sigmoid(linear_output)

    def predict(self, x_test, threshold=0.5):
        """Предсказание классов (0 или 1)"""
        probabilities = self.predict_proba(x_test)

        return (probabilities >= threshold).astype(int)
```

Исследование влияния гиперпараметров

```
# --- Исследование влияния гиперпараметров на производительность модели ---
# Градиентный спуск, шаг 0.01, итераций 100
logistic_regression = LogisticRegression(learning_rate=0.01, n_iterations=100)
logistic_regression.fit(X_train, y_train)
y_predicted = logistic_regression.predict(X_test)
tp, tn, fp, fn = test_values(y_predicted, y_test)
accuracy, precision, recall, f1 = get_model_evaluation(tp, tn, fp, fn)
print(f"Accuracy: {accuracy}")
print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F1: {f1}")

print_sep()

# Градиентный спуск, шаг 1, итераций 100
logistic_regression = LogisticRegression(learning_rate=1, n_iterations=100)
logistic_regression.fit(X_train, y_train)
y_predicted = logistic_regression.predict(X_test)
tp, tn, fp, fn = test_values(y_predicted, y_test)
accuracy, precision, recall, f1 = get_model_evaluation(tp, tn, fp, fn)
print(f"Accuracy: {accuracy}")
print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F1: {f1}")

print_sep()

# Градиентный спуск, шаг 0.1, итераций 100
logistic_regression = LogisticRegression(learning_rate=0.1, n_iterations=100)
logistic_regression.fit(X_train, y_train)
y_predicted = logistic_regression.predict(X_test)
tp, tn, fp, fn = test_values(y_predicted, y_test)
accuracy, precision, recall, f1 = get_model_evaluation(tp, tn, fp, fn)
print(f"Accuracy: {accuracy}")
print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F1: {f1}")

print_sep()

# Градиентный спуск, шаг 0.1, итераций 1000
logistic_regression = LogisticRegression(learning_rate=0.1, n_iterations=1000)
logistic_regression.fit(X_train, y_train)
y_predicted = logistic_regression.predict(X_test)
tp, tn, fp, fn = test_values(y_predicted, y_test)
accuracy, precision, recall, f1 = get_model_evaluation(tp, tn, fp, fn)
print(f"Accuracy: {accuracy}")
print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F1: {f1}")

print_sep()

# Градиентный спуск, шаг 0.1, итераций 5000
logistic_regression = LogisticRegression(learning_rate=0.1, n_iterations=5000)
logistic_regression.fit(X_train, y_train)
y_predicted = logistic_regression.predict(X_test)
```

```
tp, tn, fp, fn = test_values(y_predicted, y_test)
accuracy, precision, recall, f1 = get_model_evaluation(tp, tn, fp, fn)
print(f"Accuracy: {accuracy}")
print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F1: {f1}")

print_sep()

# Метод Ньютона, итераций 100
logistic_regression = LogisticRegres-
sion(opt_type=OptimizationType.NEWTON_METHOD, n_iterations=100)
logistic_regression.fit(X_train, y_train)
y_predicted = logistic_regression.predict(X_test)
tp, tn, fp, fn = test_values(y_predicted, y_test)
accuracy, precision, recall, f1 = get_model_evaluation(tp, tn, fp, fn)
print(f"Accuracy: {accuracy}")
print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F1: {f1}")

print_sep()

# Метод Ньютона, итераций 1000
logistic_regression = LogisticRegres-
sion(opt_type=OptimizationType.NEWTON_METHOD, n_iterations=1000)
logistic_regression.fit(X_train, y_train)
y_predicted = logistic_regression.predict(X_test)
tp, tn, fp, fn = test_values(y_predicted, y_test)
accuracy, precision, recall, f1 = get_model_evaluation(tp, tn, fp, fn)
print(f"Accuracy: {accuracy}")
print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F1: {f1}")
```

Оценка моделей

```
==================================================
Начинаем обучение методом: GRADIENT_DESCENT
Скорость обучения: 0.01
Количество итераций: 100
  Итерация   20, Loss: 0.6759
  Итерация   40, Loss: 0.6606
  Итерация   60, Loss: 0.6476
  Итерация   80, Loss: 0.6364
  Итерация  100, Loss: 0.6266
  Градиентный спуск завершен
Accuracy: 0.715311004784689
Precision: 0.21710526315789475
Recall: 1.0
F1: 0.3567567567567568
==================================================
```

Рисунок 1 – Модель 1.

```
==================================================
Начинаем обучение методом: GRADIENT_DESCENT
Скорость обучения: 1
Количество итераций: 100
  Итерация   20, Loss: 0.4679
  Итерация   40, Loss: 0.4583
  Итерация   60, Loss: 0.4558
  Итерация   80, Loss: 0.4543
  Итерация  100, Loss: 0.4530
  Градиентный спуск завершен
Accuracy: 0.9688995215311005
Precision: 0.9868421052631579
Recall: 0.9316770186335404
F1: 0.9584664536741214
==================================================
```

Рисунок 2 – Модель 2.

```
==================================================
Начинаем обучение методом: GRADIENT_DESCENT
Скорость обучения: 0.1
Количество итераций: 100
  Итерация   20, Loss: 0.5929
  Итерация   40, Loss: 0.5495
  Итерация   60, Loss: 0.5229
  Итерация   80, Loss: 0.5052
  Итерация  100, Loss: 0.4931
  Градиентный спуск завершен
Accuracy: 0.9066985645933014
Precision: 0.743421052631579
Recall: 1.0
F1: 0.8528301886792453
==================================================
```

Рисунок 3 – Модель 3.

```
==================================================
Начинаем обучение методом: GRADIENT_DESCENT
Скорость обучения: 0.1
Количество итераций: 1000
  Итерация  200, Loss: 0.4678
  Итерация  400, Loss: 0.4583
  Итерация  600, Loss: 0.4558
  Итерация  800, Loss: 0.4542
  Итерация 1000, Loss: 0.4529
  Градиентный спуск завершен
Accuracy: 0.9688995215311005
Precision: 0.9868421052631579
Recall: 0.9316770186335404
F1: 0.9584664536741214
==================================================
```

Рисунок 4 – Модель 4.

```
==================================================
Начинаем обучение методом: GRADIENT_DESCENT
Скорость обучения: 0.1
Количество итераций: 5000
  Итерация 1000, Loss: 0.4529
  Итерация 2000, Loss: 0.4483
  Итерация 3000, Loss: 0.4455
  Итерация 4000, Loss: 0.4438
  Итерация 5000, Loss: 0.4428
  Градиентный спуск завершен
Accuracy: 0.9497607655502392
Precision: 0.9407894736842105
Recall: 0.9225806451612903
F1: 0.9315960912052118
==================================================
```

Рисунок 5 – Модель 5.

```
==================================================
Начинаем обучение методом: NEWTON_METHOD
Скорость обучения: 0.01
Количество итераций: 100
  Сошлось на итерации 6
  Loss: 0.4405
  Метод Ньютона завершен
Accuracy: 0.9449760765550239
Precision: 0.9342105263157895
Recall: 0.9161290322580645
F1: 0.9250814332247558
==================================================
```

Рисунок 6 – Модель 6.

```
==================================================
Начинаем обучение методом: NEWTON_METHOD
Скорость обучения: 0.01
Количество итераций: 1000
  Сошлось на итерации 6
  Loss: 0.4405
  Метод Ньютона завершен
Accuracy: 0.9449760765550239
Precision: 0.9342105263157895
Recall: 0.9161290322580645
F1: 0.9250814332247558
```

Рисунок 7 – Модель 7.

Разработанное приложение

Исходный код приложения: https://github.com/vvlaads/AI-systems-6.

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы была реализована и исследована модель логистической регрессии для решения задачи бинарной классификации на примере датасета "Titanic". Основной целью исследования был анализ влияния гиперпараметров алгоритма на его производительность, а также сравнение двух методов оптимизации: градиентного спуска и метода Ньютона.

Проведённый эксперимент показал существенную зависимость качества модели от выбора гиперпараметров. Наиболее критическим параметром для градиентного спуска оказалась скорость обучения. При слишком малом значении (0.01) алгоритм демонстрировал медленную сходимость даже после 100 итераций, что выражалось в высоком значении функции потерь (0.6266) и низкой точности модели (Accuracy = 0.715). При этом модель показывала максимальную полноту (Recall = 1.0), но крайне низкую точность предсказаний положительного класса (Precision = 0.217), что свидетельствует о чрезмерно "осторожном" поведении классификатора.

Оптимальное значение скорости обучения для данного набора данных составило 0.1. При этом параметре модель достигала хорошего баланса между точностью и полнотой уже после 100 итераций (F1 = 0.853). Дальнейшее увеличение скорости обучения до 1 привело к ускорению сходимости и незначительному улучшению метрик (F1 = 0.958 после 100 итераций), однако существует риск расходимости при столь высоком значении, что требует дополнительной проверки на других наборах данных.

Исследование влияния количества итераций выявило, что для градиентного спуска с learning_rate = 0.1 достаточно 1000 итераций для достижения стабильных результатов. Дальнейшее увеличение до 5000 итераций привело к незначительному снижению качества (F1 уменьшился с 0.958 до 0.932), что может свидетельствовать о начале переобучения или необходимости регулировки скорости обучения в процессе оптимизации.

Метод Ньютона продемонстрировал существенные преимущества в скорости сходимости, достигая оптимального решения всего за 6 итераций, что значительно быстрее градиентного спуска. При этом качество классификации оказалось стабильно высоким (Accuracy = 0.945, F1 = 0.925) и не зависело от заданного максимального количества итераций, так как алгоритм останавливался при достижении сходимости. Однако следует отметить, что метод Ньютона требует вычисления матрицы вторых производных, что может быть вычислительно затратно для задач большой размерности.

Важным наблюдением стало то, что метрики Recall и Precision находились в обратной зависимости: улучшение одной часто приводило к ухудшению другой. Наилучший баланс этих метрик был достигнут при использовании градиентного спуска с параметрами learning_rate = 1 и n_iterations = 100, что дало значение F1-меры 0.958 — наивысшее среди всех испытанных конфигураций.

Таким образом, можно сделать вывод, что для данного набора данных оптимальными параметрами градиентного спуска являются learning_rate = 0.1-1 и n_iterations = 100-1000. Метод Ньютона показал себя как более эффективный с точки зрения скорости сходимости, но его практическое применение может быть ограничено вычислительной сложностью для задач с большим количеством признаков. Полученные результаты подтверждают важность тщательного подбора гиперпараметров для достижения максимальной эффективности моделей машинного обучения.