Лабораторная работа №1 - KNN

Выбор набора данных и метрик

Я выбрал датасет, содержащий данные о химическом составе различных вин. Этот датасет можно использовать как для классификации, так и для регрессии. Для классификации мы можем выделить несколько классов качества вина (например, 0 - низкое качество, 1 - среднее, 2 - высокое качество). Также этот датасет не содержит пропусков, что упрощает работу.

Классификация: задача классификации качества вина. **Регрессия**: задача предсказания точного значения качества вина.

Метрики для оценки:

- Для классификации: Accuracy, F1-Score.
- Для регрессии: MSE, R^2, MAE.

Для классификации:

- `Accuracy (Точность)`: Эта метрика показывает долю правильно классифицированных примеров. Применяется, если классы сбалансированы и важна общая точность классификации.
- `F1-Score`: Этот показатель является средним гармоническим точности и полноты. Подходит, когда классы несбалансированы, и важно минимизировать как ложноположительные, так и ложносогласные ошибки.

Для регрессии:

- `Mean Squared Error (MSE)`: Среднеквадратичная ошибка используется для измерения разницы между предсказанными и реальными значениями. Это хорошая метрика для оценки точности модели в задачах регрессии.
- `R² (Коэффициент детерминации)`: Это метрика, которая оценивает, какая доля вариации целевой переменной объясняется моделью. R^2 близкий к 1 означает хорошую модель.
- `Mean Absolute Error (MAE)`: Средняя абсолютная ошибка также используется для оценки отклонений между предсказанными и реальными значениями.

Практическая значимость: Прогнозирование качества вина является реальной задачей, используемой в виноделии, чтобы оценить, какие химические компоненты оказывают влияние на качество.

Алгоритм KNN

Импортируем библиотеки

In [208…
```
import pandas as pd
import numpy as np
from collections import Counter
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.neighbors import KNeighborsClassifier, KNeighborsRegressor
from sklearn.metrics import accuracy_score, f1_score, mean_squared_error,
from sklearn.preprocessing import StandardScaler
```

Загрузка данных из датасета

In [209…
```
# Загрузка данных
df = pd.read_csv('winequality-red.csv')
df.head()
```

Out[209…

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulp
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	

Разделение данных:

- Для классификации целевая переменная y_class преобразована в бинарную (0 или 1) на основе того, если качество вина больше или равно 7, считаем его высококачественным.
- Для регрессии мы оставляем точное значение качества.

In [210…
```
# Разделение на признаки и целевую переменную для классификации и регресс
X = df.drop('quality', axis=1)

# Для классификации:
y_class = (df['quality'] >= 7).astype(int)
# Для регрессии:
y_reg = df['quality']
```

Разделение данных на обучающую и тестовую выборки (80% обучение, 20% тестирование)

In [211…
```
X_train, X_test, y_train_class, y_test_class = train_test_split(X, y_clas
y_train_reg, y_test_reg = train_test_split(y_reg, test_size=0.2, random_s
```

Классификация с использованием KNN

In [212…
```
knn_class = KNeighborsClassifier(n_neighbors=5)
knn_class.fit(X_train, y_train_class)
y_pred_class = knn_class.predict(X_test)
```

Оценка модели классификации

In [213…
```
accuracy_classic_class = accuracy_score(y_test_class, y_pred_class)
f1_classic_class = f1_score(y_test_class, y_pred_class)
print("Бейзлайн:")
print(f"Accuracy: {accuracy_classic_class:.4f}")
print(f"F1 Score: {f1_classic_class:.4f}")
```

```
Бейзлайн:
Accuracy: 0.8562
F1 Score: 0.3030
```

Подбор гиперпараметров с помощью GridSearchCV

In [214…
```
param_grid = {
    'n_neighbors': [3, 5, 7, 9, 11],
    'metric': ['euclidean', 'manhattan', 'minkowski']
}
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

knn_class = KNeighborsClassifier()
grid_search_class = GridSearchCV(knn_class, param_grid, cv=5)
grid_search_class.fit(X_train_scaled, y_train_class)

# Лучшие параметры
print("Best parameters for classification:", grid_search_class.best_param
```

```
Best parameters for classification: {'metric': 'manhattan', 'n_neighbors':
11}
```

Обучение модели с улучшениями

In [215…
```
best_knn_class = grid_search_class.best_estimator_

y_pred_class = best_knn_class.predict(X_test_scaled)
accuracy_impr_class = accuracy_score(y_test_class, y_pred_class)
f1_impr_class = f1_score(y_test_class, y_pred_class)
```

Вывод метрик

In [216…
```
print("Улучшенный бейзлайн:")
print(f"Accuracy: {accuracy_impr_class:.4f}")
print(f"F1 Score: {f1_impr_class:.4f}")

print("\nClassification Report:")
print(classification_report(y_test_class, y_pred_class))
```

```
Улучшенный бейзлайн:
Accuracy: 0.8812
F1 Score: 0.5128

Classification Report:
              precision    recall  f1-score   support

           0       0.91      0.96      0.93       273
           1       0.65      0.43      0.51        47

    accuracy                           0.88       320
   macro avg       0.78      0.69      0.72       320
weighted avg       0.87      0.88      0.87       320
```

Реализуем собственную версию KNN

In [217…

```python
class CustomKNN:
    def __init__(self, n_neighbors=5, metric='euclidean'):
        self.n_neighbors = n_neighbors
        self.metric = metric

    def fit(self, X, y):
        self.X_train = np.array(X)
        self.y_train = np.array(y)

    def _compute_distance(self, x1, x2):
        if self.metric == 'euclidean':
            return np.sqrt(np.sum((x1 - x2) ** 2))
        elif self.metric == 'manhattan':
            return np.sum(np.abs(x1 - x2))
        else:
            raise ValueError("Unsupported metric!")

    def _get_neighbors(self, x):
        distances = [self._compute_distance(x, x_train) for x_train in se
        neighbors = np.argsort(distances)[:self.n_neighbors]
        return neighbors

    def predict_classification(self, X):
        predictions = []
        for x in X:
            neighbors = self._get_neighbors(x)
            neighbor_labels = self.y_train[neighbors]
            most_common = Counter(neighbor_labels).most_common(1)[0][0]
            predictions.append(most_common)
        return np.array(predictions)

# Обучение
knn_custom_class = CustomKNN(n_neighbors=5)
knn_custom_class.fit(X_train_scaled, y_train_class)
y_pred_custom_class = knn_custom_class.predict_classification(X_test_scal

# Метрики
accuracy_custom_class = accuracy_score(y_test_class, y_pred_custom_class)
f1_custom_class = f1_score(y_test_class, y_pred_custom_class)
```

```
# Вывод результатов
print("Реализация Custom KNN:")
print(f"Accuracy: {accuracy_custom_class:.4f}")
print(f"F1 Score: {f1_custom_class:.4f}")
```

```
Реализация Custom KNN:
Accuracy: 0.8812
F1 Score: 0.5128
```

Сравним полученные резльтаты:

In [218…
```
print("Сравнение результатов:")
print(f"Бейзлайн Accuracy: {accuracy_classic_class:.4f}, Улучшенный Accur
print(f"Бейзлайн F1-Score: {f1_classic_class:.4f}, Улучшенный F1-Score: {
```

```
Сравнение результатов:
Бейзлайн Accuracy: 0.8562, Улучшенный Accuracy: 0.8812, Accuracy Custom KN
N : 0.8812
Бейзлайн F1-Score: 0.3030, Улучшенный F1-Score: 0.5128, F1-Score Custom KN
N: 0.5128
```

Регрессия с использованием KNN

In [219…
```
knn_reg = KNeighborsRegressor(n_neighbors=5)
knn_reg.fit(X_train, y_train_reg)
y_pred_reg = knn_reg.predict(X_test)
```

Оценка модели регрессии

In [220…
```
mse_classic = mean_squared_error(y_test_reg, y_pred_reg)
mae_classic = mean_absolute_error(y_test_reg, y_pred_reg)
r2_classic = r2_score(y_test_reg, y_pred_reg)

print("Regression Metrics:")
print(f"Mean Squared Error (MSE): {mse_classic:.4f}")
print(f"Mean Absolute Error (MAE): {mae_classic:.4f}")
print(f"R² Score: {r2_classic:.4f}")
```

```
Regression Metrics:
Mean Squared Error (MSE): 0.5320
Mean Absolute Error (MAE): 0.5788
R² Score: 0.1859
```

Подбор гиперпараметров с помощью GridSearchCV

In [221…
```
param_grid = {
    'n_neighbors': [3, 5, 7, 9, 11],
    'metric': ['euclidean', 'manhattan', 'minkowski']
}
grid_search_reg = GridSearchCV(KNeighborsRegressor(), param_grid, cv=5)
grid_search_reg.fit(X_train_scaled, y_train_reg)

# Лучшие параметры
print("Best parameters for classification:", grid_search_reg.best_params_
```

```
Best parameters for classification: {'metric': 'manhattan', 'n_neighbors':
11}
```

Обучение с лучшими параметрами

In [222…
```python
best_knn_reg = grid_search_reg.best_estimator_

y_pred_reg = best_knn_reg.predict(X_test_scaled)
mse_improved = mean_squared_error(y_test_reg, y_pred_reg)
mae_improved = mean_absolute_error(y_test_reg, y_pred_reg)
r2_improved = r2_score(y_test_reg, y_pred_reg)
```

Вывод метрик для улучшеного бейзлайна

In [223…
```python
print("Improved Regression Metrics:")
print(f"Mean Squared Error (MSE): {mse_improved:.4f}")
print(f"Mean Absolute Error (MAE): {mae_improved:.4f}")
print(f"R² Score: {r2_improved:.4f}")
```

```
Improved Regression Metrics:
Mean Squared Error (MSE): 0.3689
Mean Absolute Error (MAE): 0.4901
R² Score: 0.4355
```

Реализации своей версии KNN для регрессии

In [224…
```python
# Реализация KNN
class CustomKNN:
    def __init__(self, n_neighbors=5, metric='euclidean'):
        self.n_neighbors = n_neighbors
        self.metric = metric

    def fit(self, X, y):
        self.X_train = np.array(X)
        self.y_train = np.array(y)

    def _compute_distance(self, x1, x2):
        if self.metric == 'euclidean':
            return np.sqrt(np.sum((x1 - x2) ** 2))
        elif self.metric == 'manhattan':
            return np.sum(np.abs(x1 - x2))
        else:
            raise ValueError("Unsupported metric!")

    def _get_neighbors(self, x):
        distances = [self._compute_distance(x, x_train) for x_train in se
        neighbors = np.argsort(distances)[:self.n_neighbors]
        return neighbors

    def predict_regression(self, X):
        predictions = []
        for x in X:
            neighbors = self._get_neighbors(x)
            neighbor_values = self.y_train[neighbors]
            predictions.append(np.mean(neighbor_values))
        return np.array(predictions)
```

```
# Обучение
knn_custom_reg = CustomKNN(n_neighbors=11, metric='manhattan')
knn_custom_reg.fit(X_train_scaled, y_train_reg)
y_pred_custom_reg = knn_custom_reg.predict_regression(X_test_scaled)
```

Вывод метрик для регрессии

In [225…
```
# Регрессия
mse_custom = mean_squared_error(y_test_reg, y_pred_custom_reg)
mae_custom = mean_absolute_error(y_test_reg, y_pred_custom_reg)
r2_custom = r2_score(y_test_reg, y_pred_custom_reg)

print("\nCustom Regression Metrics:")
print(f"Mean Squared Error (MSE): {mse_custom:.4f}")
print(f"Mean Absolute Error (MAE): {mae_custom:.4f}")
print(f"R² Score: {r2_custom:.4f}")
```

```
Custom Regression Metrics:
Mean Squared Error (MSE): 0.3689
Mean Absolute Error (MAE): 0.4901
R² Score: 0.4355
```

In [226…
```
print("Сравнение результатов:")

print(f"Бейзлайн MSE: {mse_classic:.4f}, Улучшенный MSE: {mse_improved:.4
print(f"Бейзлайн MAE: {mae_classic:.4f}, Улучшенный MAE: {mae_improved:.4
print(f"Бейзлайн R²: {r2_classic:.4f}, Улучшенный R²: {r2_improved:.4f},
```

```
Сравнение результатов:
Бейзлайн MSE: 0.5320, Улучшенный MSE: 0.3689, Custom KNN MSE: 0.3689
Бейзлайн MAE: 0.5788, Улучшенный MAE: 0.4901, Custom KNN MAE: 0.4901
Бейзлайн R²: 0.1859, Улучшенный R²: 0.4355, Custom KNN R²: 0.4355
```

Лабораторная работа №2 - Лог и Лин рег.

In [227…
```
import numpy as np
from sklearn.linear_model import LogisticRegression, LinearRegression, Ri
from sklearn.metrics import accuracy_score, f1_score, mean_squared_error,
from sklearn.model_selection import GridSearchCV
```

In [228…
```
# Загрузка данных
df = pd.read_csv('winequality-red.csv')
df.head()
```

Out[228…

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulp
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	

Разделение данных:

- Для классификации целевая переменная y_class преобразована в бинарную (0 или 1) на основе того, если качество вина больше или равно 7, считаем его высококачественным.
- Для регрессии мы оставляем точное значение качества.

In [229…

```
# Разделение на признаки и целевую переменную для классификации и регресс
X = df.drop('quality', axis=1)

# Для классификации:
y_class = (df['quality'] >= 7).astype(int)
# Для регрессии:
y_reg = df['quality']
```

Разделение данных на обучающую и тестовую выборки (80% обучение, 20% тестирование)

In [230…

```
X_train, X_test, y_train_class, y_test_class = train_test_split(X, y_clas
y_train_reg, y_test_reg = train_test_split(y_reg, test_size=0.2, random_s
```

Логистическая регрессия (классификация)

Используем встроенный алгоритм

In [231…

```
# Логистическая регрессия для классификации
logreg_class = LogisticRegression(max_iter=200)
logreg_class.fit(X_train_scaled, y_train_class)
y_pred_class_logreg = logreg_class.predict(X_test_scaled)

# Оценка качества
accuracy_logreg = accuracy_score(y_test_class, y_pred_class_logreg)
f1_logreg = f1_score(y_test_class, y_pred_class_logreg)

print("Бейзлайн:")
print(f"Accuracy: {accuracy_logreg:.4f}")
print(f"F1 Score: {f1_logreg:.4f}")
print(classification_report(y_test_class, y_pred_class))
```

```
Бейзлайн:
Accuracy: 0.8656
F1 Score: 0.3768
              precision    recall  f1-score   support

           0       0.91      0.96      0.93       273
           1       0.65      0.43      0.51        47

    accuracy                           0.88       320
   macro avg       0.78      0.69      0.72       320
weighted avg       0.87      0.88      0.87       320
```

Оптимизация гиперпараметров для улучшения базовой модели

In [232…
```
param_grid_logreg = {
    'C': [0.1, 1, 10],
    'solver': ['lbfgs', 'liblinear'],
    'penalty': ['l2']
}

grid_search_logreg = GridSearchCV(LogisticRegression(max_iter=200), param
grid_search_logreg.fit(X_train_scaled, y_train_class)

# Лучшие параметры
print("Best parameters for Logistic Regression:", grid_search_logreg.best
```

```
Best parameters for Logistic Regression: {'C': 0.1, 'penalty': 'l2', 'solv
er': 'liblinear'}
```

Обучаем модель с лучшими параметрами и выводим метрики

In [233…
```
best_logreg = grid_search_logreg.best_estimator_

y_pred_class = best_logreg.predict(X_test_scaled)

accuracy_logreg_improved = accuracy_score(y_test_class, y_pred_class)
f1_logreg_improved = f1_score(y_test_class, y_pred_class)

print("Улучшенный Бейзлайн:")
print(f"Accuracy: {accuracy_logreg_improved:.4f}")
print(f"F1 Score: {f1_logreg_improved:.4f}")

print(classification_report(y_test_class, y_pred_class))
```

```
Улучшенный Бейзлайн:
Accuracy: 0.8594
F1 Score: 0.3077
              precision    recall  f1-score   support

           0       0.88      0.97      0.92       273
           1       0.56      0.21      0.31        47

    accuracy                           0.86       320
   macro avg       0.72      0.59      0.61       320
weighted avg       0.83      0.86      0.83       320
```

Собственная реализация логистической регрессии

In [234…
```python
class CustomLogisticRegression:
    def __init__(self, learning_rate=0.01, n_iterations=1000):
        self.learning_rate = learning_rate
        self.n_iterations = n_iterations
        self.weights = None
        self.bias = None

    def sigmoid(self, z):
        return 1 / (1 + np.exp(-z))

    def fit(self, X, y):
        n_samples, n_features = X.shape
        self.weights = np.zeros(n_features)
        self.bias = 0

        for _ in range(self.n_iterations):
            model = np.dot(X, self.weights) + self.bias
            predictions = self.sigmoid(model)

            dw = (1/n_samples) * np.dot(X.T, (predictions - y))
            db = (1/n_samples) * np.sum(predictions - y)

            self.weights -= self.learning_rate * dw
            self.bias -= self.learning_rate * db

    def predict(self, X):
        model = np.dot(X, self.weights) + self.bias
        predictions = self.sigmoid(model)
        return [1 if i > 0.5 else 0 for i in predictions]

# Обучение кастомной логистической регрессии
custom_logreg = CustomLogisticRegression()
custom_logreg.fit(X_train_scaled, y_train_class)
y_pred_class_custom_logreg = custom_logreg.predict(X_test_scaled)

# Оценка качества
accuracy_custom_logreg = accuracy_score(y_test_class, y_pred_class_custom
f1_custom_logreg = f1_score(y_test_class, y_pred_class_custom_logreg)

print("Custom Logistic Regression Accuracy:", accuracy_custom_logreg)
print("Custom Logistic Regression F1 Score:", f1_custom_logreg)
```

```
Custom Logistic Regression Accuracy: 0.8625
Custom Logistic Regression F1 Score: 0.3125
```

Сравнение результатов

In [235…
```python
print("Сравнение результатов:")
print(f"Бейзлайн Accuracy: {accuracy_logreg:.4f}, Улучшенный Accuracy: {a
print(f"Бейзлайн F1-Score: {f1_logreg:.4f}, Улучшенный F1-Score: {f1_logr
```

```
Сравнение результатов:
Бейзлайн Accuracy: 0.8656, Улучшенный Accuracy: 0.8594, Custom Accuracy:
0.8625
Бейзлайн F1-Score: 0.3768, Улучшенный F1-Score: 0.3077, Custom F1-Score:
0.3125
```

Линейная регрессия (регрессия)

Обучение встроенной реализации модели

In [236…
```
linreg = LinearRegression()
linreg.fit(X_train_scaled, y_train_reg)
y_pred_reg_linreg = linreg.predict(X_test_scaled)
```

Вывод метрик

In [237…
```
# Оценка качества
mse_linreg = mean_squared_error(y_test_reg, y_pred_reg_linreg)
mae_linreg = mean_absolute_error(y_test_reg, y_pred_reg_linreg)
r2_linreg = r2_score(y_test_reg, y_pred_reg_linreg)

print("Бейзлайн:")
print(f"Linear Regression MSE: {mse_linreg:.4f}")
print(f"Linear Regression MAE: {mae_linreg:.4f}")
print(f"Linear Regression R²: {r2_linreg:.4f}")
```

```
Бейзлайн:
Linear Regression MSE: 0.3900
Linear Regression MAE: 0.5035
Linear Regression R²: 0.4032
```

Применение Ridge регрессии для улучшения модели

In [238…
```
ridge = Ridge(alpha=1)
ridge.fit(X_train_scaled, y_train_reg)
y_pred_reg_ridge = ridge.predict(X_test_scaled)

# Оценка качества
mse_linreg_improved = mean_squared_error(y_test_reg, y_pred_reg_ridge)
mae_linreg_improved = mean_absolute_error(y_test_reg, y_pred_reg_ridge)
r2_linreg_improved = r2_score(y_test_reg, y_pred_reg_ridge)

print("Улучшенный Бейзлайн:")
print(f"Ridge Regression MSE: {mse_linreg_improved:.4f}")
print(f"Ridge Regression MAE: {mae_linreg_improved:.4f}")
print(f"Ridge Regression R²: {r2_linreg_improved:.4f}")
```

```
Улучшенный Бейзлайн:
Ridge Regression MSE: 0.3900
Ridge Regression MAE: 0.5036
Ridge Regression R²: 0.4032
```

Реализация собственной версии линейной регрессии

In [239…
```
class CustomLinearRegression:
```

```
    def __init__(self, learning_rate=0.01, n_iterations=1000):
        self.learning_rate = learning_rate
        self.n_iterations = n_iterations
        self.weights = None
        self.bias = None

    def fit(self, X, y):
        n_samples, n_features = X.shape
        self.weights = np.zeros(n_features)
        self.bias = 0

        for _ in range(self.n_iterations):
            y_pred = np.dot(X, self.weights) + self.bias
            dw = (1/n_samples) * np.dot(X.T, (y_pred - y))
            db = (1/n_samples) * np.sum(y_pred - y)

            self.weights -= self.learning_rate * dw
            self.bias -= self.learning_rate * db

    def predict(self, X):
        return np.dot(X, self.weights) + self.bias

# Обучение кастомной линейной регрессии
custom_linreg = CustomLinearRegression()
custom_linreg.fit(X_train_scaled, y_train_reg)
y_pred_reg_custom_linreg = custom_linreg.predict(X_test_scaled)

# Оценка качества
mse_custom_linreg = mean_squared_error(y_test_reg, y_pred_reg_custom_linr
mae_custom_linreg = mean_absolute_error(y_test_reg, y_pred_reg_custom_lin
r2_custom_linreg = r2_score(y_test_reg, y_pred_reg_custom_linreg)

print(f"Custom Linear Regression MSE: {mse_custom_linreg:.4f}")
print(f"Custom Linear Regression MAE: {mae_custom_linreg:.4f}")
print(f"Custom Linear Regression R²: {r2_custom_linreg:.4f}")
```

```
Custom Linear Regression MSE: 0.3899
Custom Linear Regression MAE: 0.5035
Custom Linear Regression R²: 0.4034
```

Сравнение результатов:

In [240…

```
print("Сравнение результатов:")
print(f"Бейзлайн MSE: {mse_linreg:.4f}, Улучшенный MSE: {mse_linreg_impro
print(f"Бейзлайн MAE: {mae_linreg:.4f}, Улучшенный MAE: {mae_linreg_impro
print(f"Бейзлайн R²: {r2_linreg:.4f}, Улучшенный R²: {r2_linreg_improved:
```

```
Сравнение результатов:
Бейзлайн MSE: 0.3900, Улучшенный MSE: 0.3900, Custom MSE: 0.3899
Бейзлайн MAE: 0.5035, Улучшенный MAE: 0.5036, Custom MAE: 0.5035
Бейзлайн R²: 0.4032, Улучшенный R²: 0.4032, Custom R²: 0.4034
```

Лабораторная работа №3 - Решающее дерево

Импортируем нужные библиотеки

In [241…
```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor
from sklearn.metrics import accuracy_score, f1_score, classification_repo
from collections import Counter
```

Загрузка данных

In [242…
```
df = pd.read_csv('winequality-red.csv')
df.head()
```

Out[242…

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulp
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	

Разделение данных:

- Для классификации целевая переменная y_class преобразована в бинарную (0 или 1) на основе того, если качество вина больше или равно 7, считаем его высококачественным.
- Для регрессии мы оставляем точное значение качества.

In [243…
```
# Разделение на признаки и целевую переменную для классификации и регресс
X = df.drop('quality', axis=1)

# Для классификации:
y_class = (df['quality'] >= 7).astype(int)
# Для регрессии:
y_reg = df['quality']
```

Разделение данных на обучающую и тестовую выборки (80% обучение, 20% тестирование)

In [244…
```
X_train, X_test, y_train_class, y_test_class = train_test_split(X, y_clas
y_train_reg, y_test_reg = train_test_split(y_reg, test_size=0.2, random_s
```

Масштабирование

In [245…
```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Классификация

Используем встроенную версию решающего дерева для обучения модели

In [246…
```
# Бейзлайн: Решающее дерево без настройки гиперпараметров
dt_clf_baseline = DecisionTreeClassifier(random_state=42)
dt_clf_baseline.fit(X_train_scaled, y_train_class)

# Предсказания и метрики
y_pred_baseline_class = dt_clf_baseline.predict(X_test_scaled)
accuracy_tree_class = accuracy_score(y_test_class, y_pred_baseline_class)
f1_tree_class = f1_score(y_test_class, y_pred_baseline_class, average="we
```

Вывод метрик

In [247…
```
print("Бейзлайн:")
print(f"Accuracy: {accuracy_tree_class:.4f}")
print(f"F1-Score: {f1_tree_class:.4f}")

print(classification_report(y_test_class, y_pred_baseline_class))
```

```
Бейзлайн:
Accuracy: 0.8719
F1-Score: 0.8689
              precision    recall  f1-score   support

           0       0.92      0.93      0.93       273
           1       0.57      0.51      0.54        47

    accuracy                           0.87       320
   macro avg       0.74      0.72      0.73       320
weighted avg       0.87      0.87      0.87       320
```

Реализуем улучшеный бейзлайн с помощи настройки гиперпараметров

In [248…
```
# Параметры для настройки гиперпараметров
param_grid_class = {
    'max_depth': [3, 5, 10, None],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}

# Решающее дерево с GridSearch
grid_search_clf = GridSearchCV(DecisionTreeClassifier(random_state=42), p
grid_search_clf.fit(X_train_scaled, y_train_class)

# Лучшие параметры и метрики
best_params_class = grid_search_clf.best_params_
y_pred_improved_class = grid_search_clf.best_estimator_.predict(X_test_sc
accuracy_tree_improved = accuracy_score(y_test_class, y_pred_improved_cla
```

```
f1_tree_improved = f1_score(y_test_class, y_pred_improved_class, average=
```

Вывод метрик

In [249…
```
print("Улучшенный Бейзлайн:")
print(f"Accuracy: {accuracy_tree_improved:.4f}")
print(f"F1-Score: {f1_tree_improved:.4f}")
print(classification_report(y_test_class, y_pred_improved_class))
```

```
Улучшенный Бейзлайн:
Accuracy: 0.8594
F1-Score: 0.8344
              precision    recall  f1-score   support

           0       0.88      0.97      0.92       273
           1       0.55      0.23      0.33        47

    accuracy                           0.86       320
   macro avg       0.72      0.60      0.62       320
weighted avg       0.83      0.86      0.83       320
```

Реализуем собственнуб версию решающего дерева

In [250…
```
class CustomDecisionTree:
    def __init__(self, max_depth=None, min_samples_split=2, task='classif
        """
        Универсальное дерево решений.

        Параметры:
        - max_depth: Максимальная глубина дерева.
        - min_samples_split: Минимальное число элементов для разделения.
        - task: Тип задачи ('classification' или 'regression').
        """
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.task = task
        self.tree = None

    def fit(self, X, y):
        """Обучение дерева решений."""
        self.tree = self._build_tree(X, y, depth=0)

    def predict(self, X):
        """Предсказание для входных данных."""
        return np.array([self._predict_single(dict(zip(X.columns, x)), se

    def _entropy(self, y):
        """Вычисление энтропии."""
        counts = np.bincount(y)
        probabilities = counts / len(y)
        return -np.sum([p * np.log2(p) for p in probabilities if p > 0])

    def _gini(self, y):
        """Вычисление индекса Джини."""
        counts = np.bincount(y)
        probabilities = counts / len(y)
```

```
        return 1 - np.sum(probabilities ** 2)

    def _mse(self, y):
        """Вычисление MSE."""
        mean = np.mean(y)
        return np.mean((y - mean) ** 2)

    def _criterion(self, y, y_left, y_right):
        """Вычисление критерия разбиения."""
        if self.task == 'classification':
            return self._entropy(y) - (
                len(y_left) / len(y) * self._entropy(y_left) + len(y_righ
            )
        elif self.task == 'regression':
            return self._mse(y) - (
                len(y_left) / len(y) * self._mse(y_left) + len(y_right) /
            )
        else:
            raise ValueError("Неподдерживаемая задача. Используйте 'class

    def _best_split(self, X, y):
        """Поиск наилучшего разбиения данных."""
        best_gain = -1
        best_split = None
        best_column = None

        for column in X.columns:
            values = X[column].unique()
            for value in values:
                y_left = y[X[column] <= value]
                y_right = y[X[column] > value]

                if len(y_left) == 0 or len(y_right) == 0:
                    continue

                gain = self._criterion(y, y_left, y_right)

                if gain > best_gain:
                    best_gain = gain
                    best_split = value
                    best_column = column

        return best_column, best_split, best_gain

    def _build_tree(self, X, y, depth):
        """Рекурсивное построение дерева."""
        if (self.max_depth is not None and depth >= self.max_depth) or le
            if self.task == 'classification':
                return {'leaf': True, 'prediction': Counter(y).most_commo
            elif self.task == 'regression':
                return {'leaf': True, 'prediction': np.mean(y)}

        column, split_value, gain = self._best_split(X, y)

        if gain == -1:
            if self.task == 'classification':
                return {'leaf': True, 'prediction': Counter(y).most_commo
```

```
            elif self.task == 'regression':
                return {'leaf': True, 'prediction': np.mean(y)}

        left_indices = X[column] <= split_value
        right_indices = X[column] > split_value

        left_tree = self._build_tree(X[left_indices], y[left_indices], de
        right_tree = self._build_tree(X[right_indices], y[right_indices],

        return {
            'leaf': False,
            'column': column,
            'split_value': split_value,
            'left': left_tree,
            'right': right_tree
        }

    def _predict_single(self, x, tree):
        """Предсказание для одного примера."""
        if tree['leaf']:
            return tree['prediction']

        if x[tree['column']] <= tree['split_value']:
            return self._predict_single(x, tree['left'])
        else:
            return self._predict_single(x, tree['right'])
```

Обучим модель и выведем метрики

In [251…
```
tree_classifier = CustomDecisionTree(max_depth=5, min_samples_split=10, t
tree_classifier.fit(X_train, y_train_class)

y_pred_custom_class = tree_classifier.predict(X_test)

accuracy_tree_custom = accuracy_score(y_test_class, y_pred_custom_class)
f1_tree_custom = f1_score(y_test_class, y_pred_custom_class)

print("Кастомный Бейзлайн:")
print(f"Accuracy: {accuracy_tree_custom:.4f}")
print(f"F1-Score: {f1_tree_custom:.4f}")
```

```
Кастомный Бейзлайн:
Accuracy: 0.8969
F1-Score: 0.5926
```

Вывод и сравнение всех полученных метрик

In [252…
```
print("\nСравнение результатов (Классификация):")
print(f"Бейзлайн Accuracy: {accuracy_tree_class:.4f}, Улучшенная Accuracy
print(f"Бейзлайн F1-Score: {f1_tree_class:.4f}, Улучшенная F1-Score: {f1_
```

```
Сравнение результатов (Классификация):
Бейзлайн Accuracy: 0.8719, Улучшенная Accuracy: 0.8594, Custom Accuracy:
0.8969
Бейзлайн F1-Score: 0.8689, Улучшенная F1-Score: 0.8344, Custom F1-Score:
0.5926
```

Регрессия

Используем встроенную версию для обучения модели и выведем метрики

In [253…
```
# Бейзлайн
dt_reg_baseline = DecisionTreeRegressor(random_state=42)
dt_reg_baseline.fit(X_train_scaled, y_train_reg)

# Предсказания и метрики
y_pred_baseline_reg = dt_reg_baseline.predict(X_test_scaled)
mse_tree = mean_squared_error(y_test_reg, y_pred_baseline_reg)
mae_tree = mean_absolute_error(y_test_reg, y_pred_baseline_reg)
r2_tree = r2_score(y_test_reg, y_pred_baseline_reg)

print("Бейзлайн:")
print(f"MSE: {mse_tree:.4f}")
print(f"MAE: {mae_tree:.4f}")
print(f"R²: {r2_tree:.4f}")
```

```
Бейзлайн:
MSE: 0.6125
MAE: 0.4625
R²: 0.0627
```

Настроим гиперпараметры и выведем полученные метрики

In [254…
```
# Параметры для настройки гиперпараметров
param_grid_reg = {
    'max_depth': [3, 5, 10, None],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}

# Решающее дерево с GridSearch
grid_search_reg = GridSearchCV(DecisionTreeRegressor(random_state=42), pa
grid_search_reg.fit(X_train_scaled, y_train_reg)

# Лучшие параметры и метрики
best_params_reg = grid_search_reg.best_params_
y_pred_improved_reg = grid_search_reg.best_estimator_.predict(X_test_scal
mse_tree_improved = mean_squared_error(y_test_reg, y_pred_improved_reg)
mae_tree_improved = mean_absolute_error(y_test_reg, y_pred_improved_reg)
r2_tree_improved = r2_score(y_test_reg, y_pred_improved_reg)

print("Улучшенный Бейзлайн:")
print(f"MSE: {mse_tree_improved:.4f}")
print(f"MAE: {mae_tree_improved:.4f}")
print(f"R²: {r2_tree_improved:.4f}")
```

```
Улучшенный Бейзлайн:
MSE: 0.4400
MAE: 0.5062
R²: 0.3267
```

Реализуем собственную версию решающего дерева

In [255…

```python
class CustomDecisionTree:
    def __init__(self, max_depth=None, min_samples_split=2, task='classif
        """
        Универсальное дерево решений.

        Параметры:
        - max_depth: Максимальная глубина дерева.
        - min_samples_split: Минимальное число элементов для разделения.
        - task: Тип задачи ('classification' или 'regression').
        """
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.task = task
        self.tree = None

    def fit(self, X, y):
        """Обучение дерева решений."""
        self.tree = self._build_tree(X, y, depth=0)

    def predict(self, X):
        """Предсказание для входных данных."""
        return np.array([self._predict_single(dict(zip(X.columns, x)), se

    def _entropy(self, y):
        """Вычисление энтропии."""
        counts = np.bincount(y)
        probabilities = counts / len(y)
        return -np.sum([p * np.log2(p) for p in probabilities if p > 0])

    def _gini(self, y):
        """Вычисление индекса Джини."""
        counts = np.bincount(y)
        probabilities = counts / len(y)
        return 1 - np.sum(probabilities ** 2)

    def _mse(self, y):
        """Вычисление MSE."""
        mean = np.mean(y)
        return np.mean((y - mean) ** 2)

    def _criterion(self, y, y_left, y_right):
        """Вычисление критерия разбиения."""
        if self.task == 'classification':
            return self._entropy(y) - (
                len(y_left) / len(y) * self._entropy(y_left) + len(y_righ
            )
        elif self.task == 'regression':
            return self._mse(y) - (
                len(y_left) / len(y) * self._mse(y_left) + len(y_right) /
            )
        else:
            raise ValueError("Неподдерживаемая задача. Используйте 'class

    def _best_split(self, X, y):
        """Поиск наилучшего разбиения данных."""
        best_gain = -1
        best_split = None
```

```
        best_column = None

        for column in X.columns:
            values = X[column].unique()
            for value in values:
                y_left = y[X[column] <= value]
                y_right = y[X[column] > value]

                if len(y_left) == 0 or len(y_right) == 0:
                    continue

                gain = self._criterion(y, y_left, y_right)

                if gain > best_gain:
                    best_gain = gain
                    best_split = value
                    best_column = column

        return best_column, best_split, best_gain

    def _build_tree(self, X, y, depth):
        """Рекурсивное построение дерева."""
        if (self.max_depth is not None and depth >= self.max_depth) or le
            if self.task == 'classification':
                return {'leaf': True, 'prediction': Counter(y).most_commo
            elif self.task == 'regression':
                return {'leaf': True, 'prediction': np.mean(y)}

        column, split_value, gain = self._best_split(X, y)

        if gain == -1:
            if self.task == 'classification':
                return {'leaf': True, 'prediction': Counter(y).most_commo
            elif self.task == 'regression':
                return {'leaf': True, 'prediction': np.mean(y)}

        left_indices = X[column] <= split_value
        right_indices = X[column] > split_value

        left_tree = self._build_tree(X[left_indices], y[left_indices], de
        right_tree = self._build_tree(X[right_indices], y[right_indices],

        return {
            'leaf': False,
            'column': column,
            'split_value': split_value,
            'left': left_tree,
            'right': right_tree
        }

    def _predict_single(self, x, tree):
        """Предсказание для одного примера."""
        if tree['leaf']:
            return tree['prediction']

        if x[tree['column']] <= tree['split_value']:
            return self._predict_single(x, tree['left'])
```

```
        else:
            return self._predict_single(x, tree['right'])
```

Обучим модель и выведем полученные метрики

In [256…
```
tree_regressor = CustomDecisionTree(max_depth=5, min_samples_split=10, ta
tree_regressor.fit(X_train, y_train_reg)

y_pred_custom_reg = tree_regressor.predict(X_test)

mse_tree_custom = mean_squared_error(y_test_reg, y_pred_custom_reg)
mae_tree_custom = mean_absolute_error(y_test_reg, y_pred_custom_reg)
r2_tree_custom = r2_score(y_test_reg, y_pred_custom_reg)

print("Кастомный Бейзлайн:")
print(f"MSE: {mse_tree_custom:.4f}")
print(f"MAE: {mae_tree_custom:.4f}")
print(f"R²: {r2_tree_custom:.4f}")
```

```
Кастомный Бейзлайн:
MSE: 0.4062
MAE: 0.4890
R²: 0.3784
```

Вывод и сравнение всех метрик

In [257…
```
print("\nСравнение результатов (Регрессия):")
print(f"Бейзлайн MSE: {mse_tree:.4f}, Улучшенная MSE: {mse_tree_improved:
print(f"Бейзлайн MAE: {mae_tree:.4f}, Улучшенная MAE: {mae_tree_improved:
print(f"Бейзлайн R²: {r2_tree:.4f}, Улучшенная R²: {r2_tree_improved:.4f}
```

```
Сравнение результатов (Регрессия):
Бейзлайн MSE: 0.6125, Улучшенная MSE: 0.4400, Custom MSE: 0.4062
Бейзлайн MAE: 0.4625, Улучшенная MAE: 0.5062, Custom MAE: 0.4890
Бейзлайн R²: 0.0627, Улучшенная R²: 0.3267, Custom R²: 0.3784
```

Лабораторная работа №4 - Случайный лес

Импортируем нужные библиотеки

In [258…
```
import pandas as pd
import numpy as np
from sklearn.ensemble import RandomForestClassifier, RandomForestRegresso
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import accuracy_score, f1_score, classification_repo
from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor
from sklearn.base import BaseEstimator
```

Загрузка данных

In [259…
```
df = pd.read_csv('winequality-red.csv')
df.head()
```

Out[259…

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulp
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	

Разделение данных:

- Для классификации целевая переменная y_class преобразована в бинарную (0 или 1) на основе того, если качество вина больше или равно 7, считаем его высококачественным.
- Для регрессии мы оставляем точное значение качества.

In [260…

```
# Разделение на признаки и целевую переменную для классификации и регресс
X = df.drop('quality', axis=1)

# Для классификации:
y_class = (df['quality'] >= 7).astype(int)
# Для регрессии:
y_reg = df['quality']
```

Разделение данных на обучающую и тестовую выборки (80% обучение, 20% тестирование)

In [261…

```
X_train, X_test, y_train_class, y_test_class = train_test_split(X, y_clas
y_train_reg, y_test_reg = train_test_split(y_reg, test_size=0.2, random_s
```

Классификация

Используем встроенную версию случайного леса для обучения модели

In [262…

```
# Встроенный случайный лес для классификации
rf_classifier = RandomForestClassifier(random_state=42)
rf_classifier.fit(X_train, y_train_class)

# Предсказания
y_pred_class = rf_classifier.predict(X_test)

# Оценка метрик
accuracy_random_tree = accuracy_score(y_test_class, y_pred_class)
f1_random_tree = f1_score(y_test_class, y_pred_class)

print("Встроенный случайный лес:")
print(f"Accuracy: {accuracy_random_tree:.4f}")
```

```
print(f"F1-Score: {f1_random_tree:.4f}")
```

```
Встроенный случайный лес:
Accuracy: 0.9000
F1-Score: 0.6000
```

Реализуем улучшеный бейзлайн с помощи настройки гиперпараметров

In [263…

```
# Параметры для GridSearch
param_grid_classifier = {
    'n_estimators': [50, 100, 200],
    'max_depth': [5, 10, None],
    'min_samples_split': [2, 5, 10]
}

grid_search_classifier = GridSearchCV(RandomForestClassifier(random_state
grid_search_classifier.fit(X_train, y_train_class)

# Лучшие параметры
print("Best parameters for RandomForest Classifier:", grid_search_classif

# Обучение модели с лучшими параметрами
best_rf_classifier = grid_search_classifier.best_estimator_
y_pred_class_best = best_rf_classifier.predict(X_test)

# Оценка метрик
accuracy_random_tree_improved = accuracy_score(y_test_class, y_pred_class
f1_random_tree_improved = f1_score(y_test_class, y_pred_class_best)

print("Улучшенный случайный лес:")
print(f"Accuracy: {accuracy_random_tree_improved:.4f}")
print(f"F1-Score: {f1_random_tree_improved:.4f}")
```

```
Best parameters for RandomForest Classifier: {'max_depth': None, 'min_samp
les_split': 2, 'n_estimators': 200}
Улучшенный случайный лес:
Accuracy: 0.9031
F1-Score: 0.6076
```

Реализуем собственную версию случайного леса

In [264…

```
class CustomRandomForest(BaseEstimator):
    def __init__(self, n_estimators=100, max_depth=None, min_samples_spli
        self.n_estimators = n_estimators
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.task = task
        self.trees = []

    def fit(self, X, y):
        """Обучение случайного леса."""
        for _ in range(self.n_estimators):
            # Бутстраппинг: случайная выборка данных с возвращением
            X_bootstrap, y_bootstrap = self._bootstrap(X, y)
            tree = self._create_tree(X_bootstrap, y_bootstrap)
            self.trees.append(tree)
```

```
    def predict(self, X):
        """Предсказание для входных данных."""
        if self.task == 'classification':
            # Классификация: голосование деревьев
            predictions = [tree.predict(X) for tree in self.trees]
            return np.array([self._majority_vote(pred) for pred in zip(*p
        else:
            # Регрессия: усреднение предсказаний
            predictions = [tree.predict(X) for tree in self.trees]
            return np.mean(predictions, axis=0)

    def _bootstrap(self, X, y):
        """Метод бутстраппинга: случайная выборка данных с возвращением."
        n_samples = len(X)
        indices = np.random.choice(n_samples, n_samples, replace=True)
        return X.iloc[indices], y.iloc[indices]

    def _create_tree(self, X, y):
        """Создание решающего дерева."""
        if self.task == 'classification':
            tree = DecisionTreeClassifier(max_depth=self.max_depth, min_s
        else:
            tree = DecisionTreeRegressor(max_depth=self.max_depth, min_sa
        tree.fit(X, y)
        return tree

    def _majority_vote(self, predictions):
        """Голосование деревьев (для классификации)."""
        values, counts = np.unique(predictions, return_counts=True)
        return values[np.argmax(counts)]
```

Обучение модели и вывод метрик

In [265…
```
# Собственная реализация случайного леса для классификации
custom_rf_classifier = CustomRandomForest(n_estimators=100, max_depth=5,
custom_rf_classifier.fit(X_train, y_train_class)

# Предсказания
y_pred_custom_class = custom_rf_classifier.predict(X_test)

# Оценка метрик
accuracy_random_tree_custom = accuracy_score(y_test_class, y_pred_custom_
f1_random_tree_custom = f1_score(y_test_class, y_pred_custom_class)

print("Собственная реализация случайного леса:")
print(f"Accuracy: {accuracy_random_tree_custom:.4f}")
print(f"F1-Score: {f1_random_tree_custom:.4f}")
```

```
Собственная реализация случайного леса:
Accuracy: 0.8562
F1-Score: 0.3429
```

Вывод и сравнение всех полученных ранее метрик

In [266…
```
print("\nСравнение результатов:")
print(f"Бейзлайн Accuracy: {accuracy_random_tree:.4f}, Улучшенная Accurac
print(f"Бейзлайн F1-Score: {f1_random_tree:.4f}, Улучшенная F1-Score: {f1
```

```
Сравнение результатов:
Бейзлайн Accuracy: 0.9000, Улучшенная Accuracy: 0.9031, Custom Accuracy:
0.8562
Бейзлайн F1-Score: 0.6000, Улучшенная F1-Score: 0.6076, Custom F1-Score:
0.3429
```

Регрессия

Используем встроенную версию случайного леса для обучения модели

In [267…

```python
# Встроенный случайный лес для регрессии
rf_regressor = RandomForestRegressor(random_state=42)
rf_regressor.fit(X_train, y_train_reg)

# Предсказания
y_pred_reg = rf_regressor.predict(X_test)

# Оценка метрик
mse_random_tree = mean_squared_error(y_test_reg, y_pred_reg)
mae_random_tree = mean_absolute_error(y_test_reg, y_pred_reg)
r2_random_tree = r2_score(y_test_reg, y_pred_reg)

print("Встроенный случайный лес:")
print(f"MSE: {mse_random_tree:.4f}")
print(f"MAE: {mae_random_tree:.4f}")
print(f"R²: {r2_random_tree:.4f}")
```

```
Встроенный случайный лес:
MSE: 0.3012
MAE: 0.4224
R²: 0.5390
```

Реализуем улучшеный бейзлайн с помощи настройки гиперпараметров

In [268…

```python
# Параметры для GridSearch
param_grid_regressor = {
    'n_estimators': [50, 100, 200],
    'max_depth': [5, 10, None],
    'min_samples_split': [2, 5, 10]
}

grid_search_regressor = GridSearchCV(RandomForestRegressor(random_state=4
grid_search_regressor.fit(X_train, y_train_reg)

# Лучшие параметры
print("Best parameters for RandomForest Regressor:", grid_search_regresso

# Обучение модели с лучшими параметрами
best_rf_regressor = grid_search_regressor.best_estimator_
y_pred_reg_best = best_rf_regressor.predict(X_test)

# Оценка метрик
mse_random_tree_improved = mean_squared_error(y_test_reg, y_pred_reg_best
mae_random_tree_improved = mean_absolute_error(y_test_reg, y_pred_reg_bes
r2_random_tree_improved = r2_score(y_test_reg, y_pred_reg_best)
```

```
print("Улучшенный случайный лес:")
print(f"MSE: {mse_random_tree_improved:.4f}")
print(f"MAE: {mae_random_tree_improved:.4f}")
print(f"R²: {r2_random_tree_improved:.4f}")
```

```
Best parameters for RandomForest Regressor: {'max_depth': None, 'min_sampl
es_split': 2, 'n_estimators': 200}
Улучшенный случайный лес:
MSE: 0.3059
MAE: 0.4251
R²: 0.5319
```

Реализуем собственную версию случайного леса

In [269…

```
class CustomRandomForest(BaseEstimator):
    def __init__(self, n_estimators=100, max_depth=None, min_samples_spli
        self.n_estimators = n_estimators
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.task = task
        self.trees = []

    def fit(self, X, y):
        """Обучение случайного леса."""
        for _ in range(self.n_estimators):
            # Бутстраппинг: случайная выборка данных с возвращением
            X_bootstrap, y_bootstrap = self._bootstrap(X, y)
            tree = self._create_tree(X_bootstrap, y_bootstrap)
            self.trees.append(tree)

    def predict(self, X):
        """Предсказание для входных данных."""
        if self.task == 'classification':
            # Классификация: голосование деревьев
            predictions = [tree.predict(X) for tree in self.trees]
            return np.array([self._majority_vote(pred) for pred in zip(*p
        else:
            # Регрессия: усреднение предсказаний
            predictions = [tree.predict(X) for tree in self.trees]
            return np.mean(predictions, axis=0)

    def _bootstrap(self, X, y):
        """Метод бутстраппинга: случайная выборка данных с возвращением."
        n_samples = len(X)
        indices = np.random.choice(n_samples, n_samples, replace=True)
        return X.iloc[indices], y.iloc[indices]

    def _create_tree(self, X, y):
        """Создание решающего дерева."""
        if self.task == 'classification':
            tree = DecisionTreeClassifier(max_depth=self.max_depth, min_s
        else:
            tree = DecisionTreeRegressor(max_depth=self.max_depth, min_sa
        tree.fit(X, y)
        return tree
```

```
    def _majority_vote(self, predictions):
        """Голосование деревьев (для классификации)."""
        values, counts = np.unique(predictions, return_counts=True)
        return values[np.argmax(counts)]
```

Обучение модели и вывод метрик

In [270…
```
# Собственная реализация случайного леса для регрессии
custom_rf_regressor = CustomRandomForest(n_estimators=100, max_depth=5, m
custom_rf_regressor.fit(X_train, y_train_reg)

# Предсказания
y_pred_custom_reg = custom_rf_regressor.predict(X_test)

# Оценка метрик
mse_random_tree_custom = mean_squared_error(y_test_reg, y_pred_custom_reg
mae_random_tree_custom = mean_absolute_error(y_test_reg, y_pred_custom_re
r2_random_tree_custom = r2_score(y_test_reg, y_pred_custom_reg)

print("Собственная реализация случайного леса:")
print(f"MSE: {mse_random_tree_custom:.4f}")
print(f"MAE: {mae_random_tree_custom:.4f}")
print(f"R²: {r2_random_tree_custom:.4f}")
```

```
Собственная реализация случайного леса:
MSE: 0.3731
MAE: 0.4916
R²: 0.4291
```

Вывод и сравнение всех полученных ранее метрик

In [271…
```
print("Сравнение результатов:")
print(f"Бейзлайн MSE: {mse_random_tree:.4f}, Улучшенная MSE: {mse_random_
print(f"Бейзлайн MAE: {mae_random_tree:.4f}, Улучшенная MAE: {mae_random_
print(f"Бейзлайн R²: {r2_random_tree:.4f}, Улучшенная R²: {r2_random_tree
```

```
Сравнение результатов:
Бейзлайн MSE: 0.3012, Улучшенная MSE: 0.3059, Custom MSE: 0.3731
Бейзлайн MAE: 0.4224, Улучшенная MAE: 0.4251, Custom MAE: 0.4916
Бейзлайн R²: 0.5390, Улучшенная R²: 0.5319, Custom R²: 0.4291
```

Лабораторная работа №5 - Градиентный бустинг

Импортируем нужные библиотеки

In [272…
```
import pandas as pd
import numpy as np
from sklearn.ensemble import GradientBoostingClassifier, GradientBoosting
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import accuracy_score, f1_score, classification_repo
from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor
from sklearn.base import BaseEstimator
```

Загрузка данных

In [273…
```
df = pd.read_csv('winequality-red.csv')
df.head()
```

Out[273…

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulp
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	

Разделение данных:

- Для классификации целевая переменная y_class преобразована в бинарную (0 или 1) на основе того, если качество вина больше или равно 7, считаем его высококачественным.
- Для регрессии мы оставляем точное значение качества.

In [274…
```
# Разделение на признаки и целевую переменную для классификации и регресс
X = df.drop('quality', axis=1)

# Для классификации:
y_class = (df['quality'] >= 7).astype(int)
# Для регрессии:
y_reg = df['quality']
```

Разделение данных на обучающую и тестовую выборки (80% обучение, 20% тестирование)

In [275…
```
# Разделение на обучающие и тестовые данные
X_train, X_test, y_train_class, y_test_class = train_test_split(X, y_clas
X_train_reg, X_test_reg, y_train_reg, y_test_reg = train_test_split(X, y_
```

Классификация

Используем встроенную версию для обучения модели

In [276…
```
# Встроенный градиентный бустинг для классификации
gb_classifier = GradientBoostingClassifier(random_state=42)
gb_classifier.fit(X_train, y_train_class)

# Предсказания
y_pred_class = gb_classifier.predict(X_test)
```

```
# Оценка метрик
accuracy_gradient = accuracy_score(y_test_class, y_pred_class)
f1_gradient = f1_score(y_test_class, y_pred_class)

print("Встроенный градиентный бустинг:")
print(f"Accuracy: {accuracy_gradient:.4f}")
print(f"F1-Score: {f1_gradient:.4f}")
```

```
Встроенный градиентный бустинг:
Accuracy: 0.8812
F1-Score: 0.5250
```

Реализуем улучшеный бейзлайн с помощи настройки гиперпараметров

In [277…

```
# Параметры для GridSearch
param_grid_classifier = {
    'n_estimators': [50, 100, 200],
    'learning_rate': [0.05, 0.1, 0.2],
    'max_depth': [3, 5, 10]
}

grid_search_classifier = GridSearchCV(GradientBoostingClassifier(random_s
grid_search_classifier.fit(X_train, y_train_class)

# Лучшие параметры
print("Best parameters for GradientBoosting Classifier:", grid_search_cla

# Обучение модели с лучшими параметрами
best_gb_classifier = grid_search_classifier.best_estimator_
y_pred_class_best = best_gb_classifier.predict(X_test)

# Оценка метрик
accuracy_gradient_improved = accuracy_score(y_test_class, y_pred_class_be
f1_gradient_improved = f1_score(y_test_class, y_pred_class_best)

print("Улучшенный градиентный бустинг:")
print(f"Accuracy: {accuracy_gradient_improved:.4f}")
print(f"F1-Score: {accuracy_gradient_improved:.4f}")
```

```
Best parameters for GradientBoosting Classifier: {'learning_rate': 0.1, 'm
ax_depth': 5, 'n_estimators': 200}
Улучшенный градиентный бустинг:
Accuracy: 0.8938
F1-Score: 0.8938
```

Реализуем собственную версию градиентного бустинга

In [278…

```
class CustomGradientBoosting(BaseEstimator):
    def __init__(self, n_estimators=100, learning_rate=0.1, max_depth=3,
        self.n_estimators = n_estimators
        self.learning_rate = learning_rate
        self.max_depth = max_depth
        self.task = task
        self.trees = []
        self.init_value = None

    def fit(self, X, y):
```

```
        """Обучение градиентного бустинга."""
        # Определяем начальное значение
        if self.task == 'classification':
            self.init_value = np.log(np.mean(y) / (1 - np.mean(y)))  # Дл
        else:
            self.init_value = np.mean(y)  # Для регрессии

        # Инициализация предсказаний
        y_pred = np.full(y.shape, self.init_value, dtype=np.float64)

        for _ in range(self.n_estimators):
            # Вычисление ошибок
            if self.task == 'classification':
                residuals = y - (1 / (1 + np.exp(-y_pred)))  # Используем
            else:
                residuals = y - y_pred  # Для регрессии

            # Выбор типа дерева
            if self.task == 'classification':
                tree = DecisionTreeRegressor(max_depth=self.max_depth)
            else:
                tree = DecisionTreeRegressor(max_depth=self.max_depth)

            # Обучение дерева на остатках
            tree.fit(X, residuals)
            self.trees.append(tree)

            # Обновление предсказаний
            y_pred += self.learning_rate * tree.predict(X)


    def predict(self, X):
        """Предсказание значений."""
        y_pred = np.full(X.shape[0], self.init_value, dtype=np.float64)
        for tree in self.trees:
            y_pred += self.learning_rate * tree.predict(X)

        if self.task == 'classification':
            return (y_pred > 0).astype(int)  # Бинарная классификация
        return y_pred  # Для регрессии

    def predict_proba(self, X):
        """Предсказание вероятностей для классификации."""
        y_pred = self.predict(X)
        return 1 / (1 + np.exp(-y_pred))  # Логистическая функция
```

Обучение модели и вывод полученных метрик

In [279…

```
# Собственная реализация градиентного бустинга для классификации
custom_gb_classifier = CustomGradientBoosting(n_estimators=100, learning_
custom_gb_classifier.fit(X_train, y_train_class)

# Предсказания
y_pred_custom_class = custom_gb_classifier.predict(X_test)

# Оценка метрик
```

```
accuracy_gradient_custom = accuracy_score(y_test_class, y_pred_custom_cla
f1_gradient_custom = f1_score(y_test_class, y_pred_custom_class)

print("Собственная реализация градиентного бустинга:")
print(f"Accuracy: {accuracy_gradient_custom:.4f}")
print(f"F1-Score: {accuracy_gradient_custom:.4f}")
```

```
Собственная реализация градиентного бустинга:
Accuracy: 0.8750
F1-Score: 0.8750
```

Вывод и сравнение полученных метрик

In [280…
```
print("\nСравнение результатов:")
print(f"Бейзлайн Accuracy: {accuracy_gradient:.4f}, Улучшенная Accuracy:
print(f"Бейзлайн F1-Score: {f1_gradient:.4f}, Улучшенная F1-Score: {f1_gr
```

```
Сравнение результатов:
Бейзлайн Accuracy: 0.8812, Улучшенная Accuracy: 0.8938, Custom Accuracy:
0.8750
Бейзлайн F1-Score: 0.5250, Улучшенная F1-Score: 0.6047, Custom F1-Score:
0.3548
```

Регрессия

Используем встроенную версию для обучения модели

In [281…
```
# Встроенный градиентный бустинг для регрессии
gb_regressor = GradientBoostingRegressor(random_state=42)
gb_regressor.fit(X_train, y_train_reg)

# Предсказания
y_pred_reg = gb_regressor.predict(X_test)

# Оценка метрик
mse_gradient = mean_squared_error(y_test_reg, y_pred_reg)
mae_gradient = mean_absolute_error(y_test_reg, y_pred_reg)
r2_gradient = r2_score(y_test_reg, y_pred_reg)

print("Встроенный градиентный бустинг:")
print(f"MSE: {mse_gradient:.4f}")
print(f"MAE: {mae_gradient:.4f}")
print(f"R²: {r2_gradient:.4f}")
```

```
Встроенный градиентный бустинг:
MSE: 0.3623
MAE: 0.4849
R²: 0.4456
```

Реализуем улучшеный бейзлайн с помощи настройки гиперпараметров

In [282…
```
# Параметры для GridSearch
param_grid_regressor = {
    'n_estimators': [50, 100],
    'learning_rate': [0.05, 0.1],
    'max_depth': [3, 5]
```

```
}

grid_search_regressor = GridSearchCV(GradientBoostingRegressor(random_sta
grid_search_regressor.fit(X_train, y_train_reg)

# Лучшие параметры
print("Best parameters for GradientBoosting Regressor:", grid_search_regr

# Обучение модели с лучшими параметрами
best_gb_regressor = grid_search_regressor.best_estimator_
y_pred_reg_best = best_gb_regressor.predict(X_test)

# Оценка метрик
mse_gradient_improved = mean_squared_error(y_test_reg, y_pred_reg_best)
mae_gradient_improved = mean_absolute_error(y_test_reg, y_pred_reg_best)
r2_gradient_improved = r2_score(y_test_reg, y_pred_reg_best)

print("Улучшенный градиентный бустинг:")
print(f"MSE: {mse_gradient_improved:.4f}")
print(f"MAE: {mae_gradient_improved:.4f}")
print(f"R²: {r2_gradient_improved:.4f}")
```

```
Best parameters for GradientBoosting Regressor: {'learning_rate': 0.1, 'ma
x_depth': 5, 'n_estimators': 100}
Улучшенный градиентный бустинг:
MSE: 0.3454
MAE: 0.4554
R²: 0.4715
```

Реализуем собственную версию градиентного бустинга

In [283…

```
class CustomGradientBoosting(BaseEstimator):
    def __init__(self, n_estimators=100, learning_rate=0.1, max_depth=3,
        self.n_estimators = n_estimators
        self.learning_rate = learning_rate
        self.max_depth = max_depth
        self.task = task
        self.trees = []
        self.init_value = None

    def fit(self, X, y):
        """Обучение градиентного бустинга."""
        # Определяем начальное значение
        if self.task == 'classification':
            self.init_value = np.log(np.mean(y) / (1 - np.mean(y)))  # Дл
        else:
            self.init_value = np.mean(y)  # Для регрессии

        # Инициализация предсказаний
        y_pred = np.full(y.shape, self.init_value, dtype=np.float64)

        for _ in range(self.n_estimators):
            # Вычисление ошибок
            if self.task == 'classification':
                residuals = y - (1 / (1 + np.exp(-y_pred)))  # Используем
            else:
                residuals = y - y_pred  # Для регрессии
```

```
            # Выбор типа дерева
            if self.task == 'classification':
                tree = DecisionTreeRegressor(max_depth=self.max_depth)
            else:
                tree = DecisionTreeRegressor(max_depth=self.max_depth)

            # Обучение дерева на остатках
            tree.fit(X, residuals)
            self.trees.append(tree)

            # Обновление предсказаний
            y_pred += self.learning_rate * tree.predict(X)


    def predict(self, X):
        """Предсказание значений."""
        y_pred = np.full(X.shape[0], self.init_value, dtype=np.float64)
        for tree in self.trees:
            y_pred += self.learning_rate * tree.predict(X)

        if self.task == 'classification':
            return (y_pred > 0).astype(int)  # Бинарная классификация
        return y_pred  # Для регрессии

    def predict_proba(self, X):
        """Предсказание вероятностей для классификации."""
        y_pred = self.predict(X)
        return 1 / (1 + np.exp(-y_pred))  # Логистическая функция
```

Обучение модели и вывод полученных метрик

In [284…
```
# Собственная реализация градиентного бустинга для регрессии
custom_gb_regressor = CustomGradientBoosting(n_estimators=100, learning_r
custom_gb_regressor.fit(X_train, y_train_reg)

# Предсказания
y_pred_custom_reg = custom_gb_regressor.predict(X_test)

# Оценка метрик
mse_gradient_custom = mean_squared_error(y_test_reg, y_pred_custom_reg)
mae_gradient_custom = mean_absolute_error(y_test_reg, y_pred_custom_reg)
r2_gradient_custom = r2_score(y_test_reg, y_pred_custom_reg)

print("Собственная реализация градиентного бустинга:")
print(f"MSE: {mse_gradient_custom:.4f}")
print(f"MAE: {mae_gradient_custom:.4f}")
print(f"R²: {r2_gradient_custom:.4f}")
```

```
Собственная реализация градиентного бустинга:
MSE: 0.3630
MAE: 0.4858
R²: 0.4445
```

Вывод и сравнение всех метрик

In [285…
```
print("Сравнение результатов:")
```

```
print(f"Бейзлайн MSE: {mse_gradient:.4f}, Улучшенная MSE: {mse_gradient_i
print(f"Бейзлайн MAE: {mae_gradient:.4f}, Улучшенная MAE: {mae_gradient_i
print(f"Бейзлайн R²: {r2_gradient:.4f}, Улучшенная R²: {r2_gradient_impro
```

```
Сравнение результатов:
Бейзлайн MSE: 0.3623, Улучшенная MSE: 0.3454, Custom MSE: 0.3630
Бейзлайн MAE: 0.4849, Улучшенная MAE: 0.4554, Custom MAE: 0.4858
Бейзлайн R²: 0.4456, Улучшенная R²: 0.4715, Custom R²: 0.4445
```