МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №4

по дисциплине «Основы машинного обучения»

Тема: Классификация

Вариант 2

Студентка гр. 1304 Чернякова А.Д.

Преподаватель Жангиров Т.Р.

Санкт-Петербург

2024

Цель работы.

Изучить различные методы классификации на одном наборе данных и проанализировать полученные результаты.

Задание.

1. **Загрузка данных**
   - 1.1. Загрузите данные вашего варианта. Учтите, что метки классов являются текстом.
   - 1.2. Визуализируйте данные при помощи диаграммы рассеяния с выделением различных классов ней.
   - 1.3. Оцените сбалансированность классов
   - 1.4. Проведите предобработку данных при необходимости
   - 1.5. Разделите выборку на обучающую и тестовую. На обучающей проводите обучение модели, на тестовой расчет значений метрик.
2. **kNN**
   - 2.1. Проведите классификацию методом k ближайших соседей, подобрав параметры: количество соседей, необходимость взвешивани. Постройте графики зависимости точности (Accuracy) в зависимости от количества соседей (по 1-й линии для взвешенного и не взвешенного метода)
   - 2.2. Рассчитайте значения Precision,Recall,F1 для полученных результатов. Сопоставьте с таблицей ошибок.
3. **Логистическая регрессия**
   - 3.1. Проведите классификацию методом логистической регрессии при различных параметрах: без регуляризации, с l1 регуляризацией, с l2 регуляризацией. Постройте столбчатую диаграмму зависимости точности (Accuracy) от наличия регуляризации. Дальнейшие пункты 3.* выполняйте для лучшего параметра.
   - 3.2. Рассчитайте значения Precision,Recall,F1 для полученных результатов. Сопоставьте с таблицей ошибок.

4. Метод опорных векторов

4.1. Проведите классификацию методом опорных векторах при различных параметрах ядра: “linear”, “poly” (нужно выбрать степень), “rbf”. Постройте столбчатую диаграмму зависимости точности (Accuracy) от вида ядра. Дальнейшие пункты 4.* выполняйте для лучшего параметра.

4.2. Рассчитайте значения Precision,Recall,F1 для полученных результатов. Сопоставьте с таблицей ошибок.

5. Решающие деревья

5.1. Проведите классификацию используя решающие деревья, подобрав параметры при которых получается лучшее обобщение (максимальная глубина/максимальное количество листьев/метрика загрязнения и т.д.).

5.2. Рассчитайте значения Precision,Recall,F1 для полученных результатов. Сопоставьте с таблицей ошибок.

6. Выбор классификатора

6.1. Постройте таблицу с метриками Precision, Recall, AUC для полученных результатов каждым классификатором.

6.2. Сделайте выводы о том, какие классификаторы лучше всего подходят для вашего набора данных и в каких случаях.

Выполнение работы.

1. **Загрузка данных**

1.1. Загрузим набор данных lab4_5.csv с помощью метода read_csv, который принимает в качестве аргумента путь до файла. Убедимся, что загрузка прошла корректно с помощью метода head, который по умолчанию выводит первые 5 строк набора данных (см. листинг 1.1.1 и таблицу 1.1.1).

Листинг 1.1.1 - Загрузка данных

```
df = pd.read_csv('lab4_5.csv')
df.head()
```

Таблица 1.1.1 - Результат работы метода head

	X1	X2	Class
0	-1.115138	0.738108	DOWN
1	-1.836181	1.041943	DOWN
2	-0.771741	0.248164	DOWN
3	-2.500027	1.273076	DOWN
4	-0.291992	0.117160	DOWN

Данные загружены корректно.

Так как метки классов являются текстом, то с помощью LabelEncoder заменим их на числовые значения (см. листинг 1.1.2 и таблицу 1.1.2).

Листинг 1.1.2 - Замена меток классов на числовые значения

```
le = LabelEncoder()
df['Class'] = le.fit_transform(df['Class'])
df.head()
```

Таблица 1.1.2 - Результат работы метода head после замены

	X1	X2	Class
0	-1.115138	0.738108	0
1	-1.836181	1.041943	0
2	-0.771741	0.248164	0
3	-2.500027	1.273076	0
4	-0.291992	0.117160	0

Видно, что столбец Class теперь числовой.

1.2. Визуализируем данные при помощи диаграммы рассеяния с выделением различных классов ней (для это будем использовать библиотеку seaborn) (см. листинг 1.2 и рисунок 1.2). За выделение разным цветом разных классов отвечает параметр hue, цветовая палитра выбрана Set1.

Листинг 1.2 - Построение диаграммы рассеяния с выделением различных классов

```
scatter_plot = sns.scatterplot(data=df, x='X1', y='X2',
hue='Class', palette='Set1')
scatter_plot.set_title('Диаграмма рассеяния с выделением
классов')
plt.show()
```

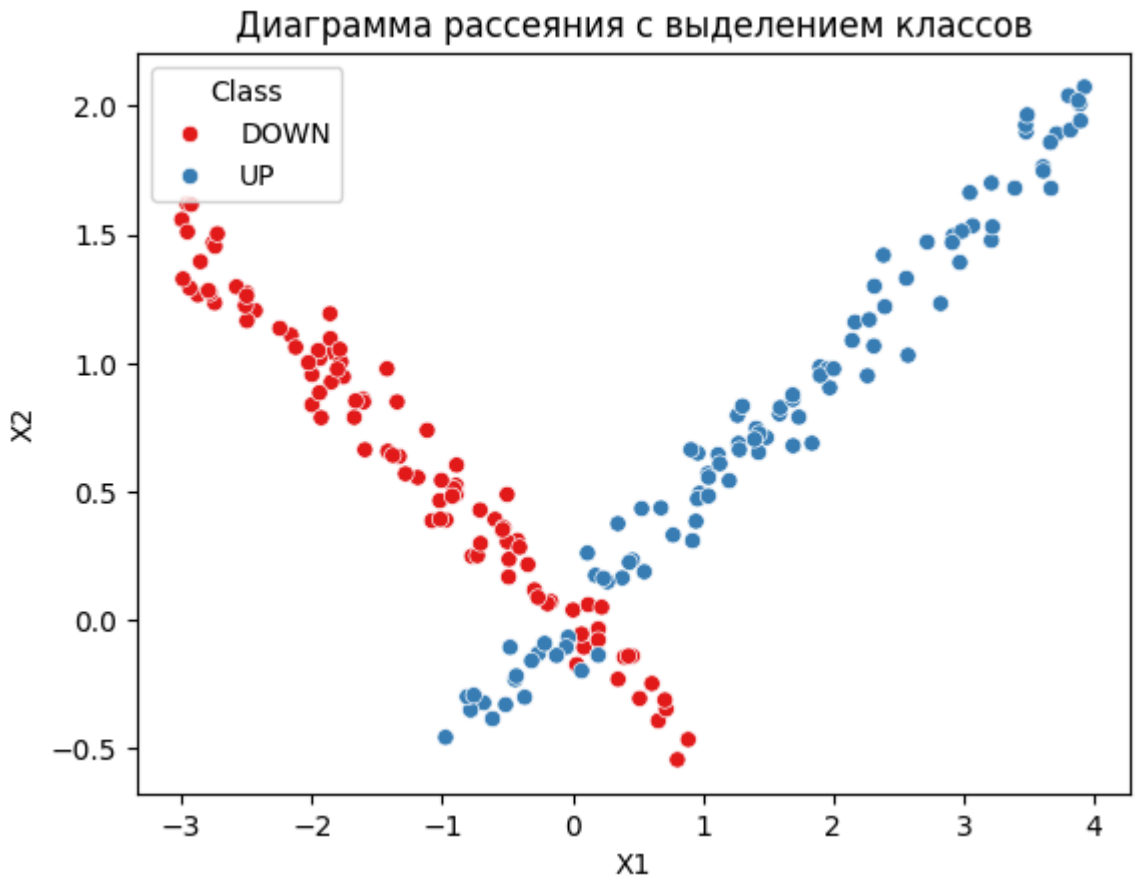

Рисунок 1.2 - Диаграмма рассеяния с выделением различных классов

1.3. Оценим сбалансированность классов. Чтобы оценить сбалансированность классов, нужно посчитать количество экземпляров каждого класса в данных. Для этого воспользуемся встроенным методом value_counts() (см. листинг 1.3 и рисунок 1.3).

Листинг 1.3 - Оценка сбалансированности классов

```
class_counts = df['Class'].value_counts()
class_counts
```

```
Class
0    100
1    100
Name: count, dtype: int64
```

Рисунок 1.3 - Оценка сбалансированности классов

Количество экземпляров каждого класса равно 100. Классы сбалансированы.

1.4. Разделим выборку на обучающую и тестовую (см. листинг 1.4). Здесь создается переменная X, содержащая признаки X1 и X2, путем удаления столбца "Class" из исходного DataFrame df. Это делается с помощью метода drop, указывая axis=1, чтобы удалить столбец, и передавая список столбцов для удаления. Создается переменная y, содержащая метки классов. В данном случае она содержит столбец "Class" из исходного DataFrame df. Далее разделим данные на обучающий и тестовый наборы с помощью функции train_test_split. Параметр test_size=0.3 указывает на то, что 30% данных будет отложено для тестирования, а оставшиеся 70% будут использованы для обучения.

На обучающей будем проводить обучение модели, на тестовой расчет значений метрик.

Листинг 1.4 - Разделение выборки на обучающую и тестовую

```
X = df.drop(["Class"], axis = 1)
y = df["Class"]
X_train, X_test, y_train, y_test =
train_test_split(X,y,test_size=0.3,random_state=42)
```

2. kNN

2.1. Проведем классификацию методом k ближайших соседей, подобрав параметры: количество соседей, необходимость взвешивания. Построим графики зависимости точности (Accuracy) в зависимости от количества соседей (см. листинг 2.1).

accuracy_score — это функция из библиотеки scikit-learn, используемая для оценки точности классификационной модели. Точность (accuracy) определяется как доля правильных предсказаний среди всех предсказаний. Это

одна из наиболее простых и часто используемых метрик для оценки качества классификационных моделей.

Листинг 2.1 - Построение графиков зависимости точности от количества соседей

```
# Списки для хранения точности
k_values = range(1, 21)
accuracies_uniform = []
accuracies_distance = []

# Проведение классификации для различных значений k
for k in k_values:
    # Невзвешенный метод
    knn_uniform = KNeighborsClassifier(n_neighbors=k,
weights='uniform')
    knn_uniform.fit(X_train, y_train)
    y_pred_uniform = knn_uniform.predict(X_test)
    accuracy_uniform = accuracy_score(y_test, y_pred_uniform)
    accuracies_uniform.append(accuracy_uniform)

    # Взвешенный метод
    knn_distance = KNeighborsClassifier(n_neighbors=k,
weights='distance')
    knn_distance.fit(X_train, y_train)
    y_pred_distance = knn_distance.predict(X_test)
    accuracy_distance = accuracy_score(y_test, y_pred_distance)
    accuracies_distance.append(accuracy_distance)

# Построение графиков
plt.plot(k_values, accuracies_uniform, label='Uniform')
plt.plot(k_values, accuracies_distance, label='Distance')
plt.xlabel('Number of Neighbors (k)')
plt.ylabel('Accuracy')
plt.xticks(k_values)  # Установка меток на оси x
plt.legend()
plt.grid()
plt.show()
```

Этот код выполняет следующие шаги:

Создаются списки k_values, accuracies_uniform, и accuracies_distance для хранения значений k (количество соседей) и точностей моделей для невзвешенного и взвешенного методов соответственно.

Проводится классификация для различных значений k (от 1 до 20) с использованием метода k ближайших соседей (k-NN).

Для каждого значения k:

Создается модель k-NN с невзвешенными соседями (weights='uniform') и обучается на тренировочных данных.

Создается модель k-NN с взвешенными соседями (weights='distance') и также обучается на тренировочных данных.

Вычисляется точность для каждой модели на тестовых данных с помощью функции accuracy_score.

Точности добавляются в соответствующие списки.

Строятся два графика, один для невзвешенного метода (Uniform) и другой для взвешенного метода (Distance), где по оси x отображается количество соседей (k), а по оси y — точность (Accuracy). Метки на оси x устанавливаются с помощью plt.xticks(k_values).

На рисунке 2.1 представлены графики зависимости точности от количества соседей со взвешиванием и без.

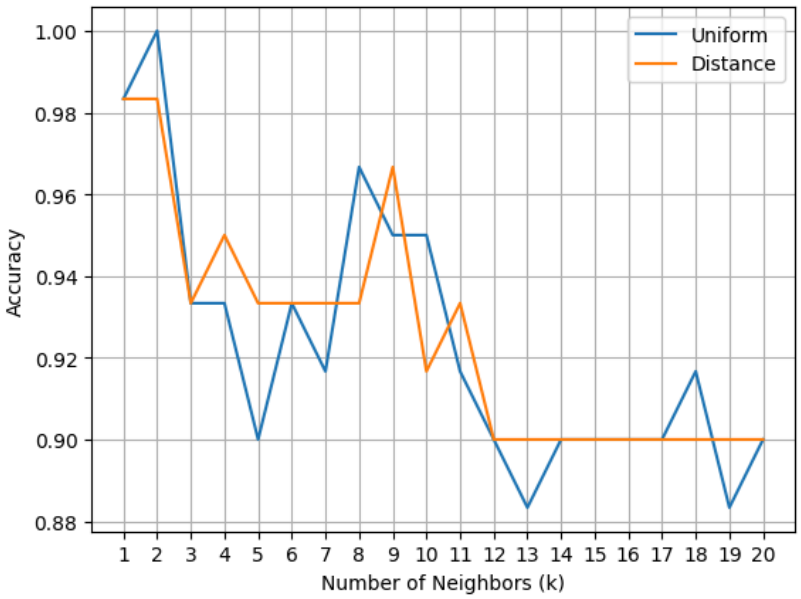

Рисунок 2.1 - Графики зависимости точности от количества соседей

По рисунку 2.1 можно заметить,что наивысшая точность (и с учетом весов, и без учета) достигается при количестве соседей равном 2. Но взвешенный метод дает точность лучше. Итак получаем, что количество соседей - 2, необходимость взвешивания - не нужно.

2.2. Результаты классификации можно представить в виде матрицы ошибок/несоответствий (см. рис. 2.2.1). Данная матрица показывает количество классифицированных тем или иным образом точек:

TP - правильно отнесены к классу

TN - правильно не отнесены к классу

FN - неправильно не отнесены к классу

FP - неправильно отнесены к классу

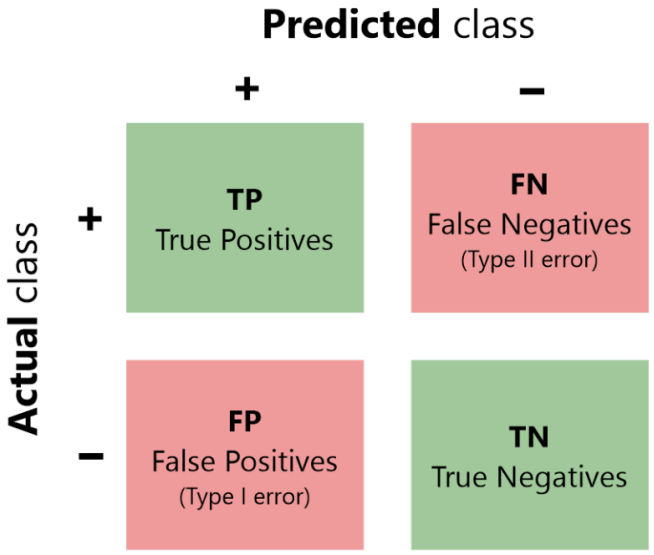

Рисунок 2.2.1 - Матрица ошибок/несоответствий

Точность/Precision (sklearn.metrics.precision_score) - TP/(TP+FP) - показывает сколько из истинных предсказаний являются реально истинными. Для несбалансированных классов не очень показательная метрика.

Полнота/Recall (sklearn.metrics.recall_score) - TP/(TP+FN) - показывает сколько реально истинных значений были правильно определены как истинные

F1-мера (sklearn.metrics.f1_score) - 2*TP/(2*TP+FN+FP) - среднее гармоническое из Precision и Recall

Рассчитаем значения Precision,Recall,F1 для полученных результатов. Для этого напишем функцию, представленную в листинге 2.2.1. Так классы сбалансированы, то пусть average=’macro’. Вызов функции представлен в листинге 2.2.2. Результат работы смотреть на рисунке 2.2.2.

Листинг 2.2.1 - Функция вычисления Precision,Recall,F1

```
# Функция для вычисления метрик и построения таблицы ошибок
def metrics (y_test, y_pred):
    precision = precision_score(y_test, y_pred, average='macro')
    recall = recall_score(y_test, y_pred, average='macro')
    f1 = f1_score(y_test, y_pred, average='macro')
    cm = confusion_matrix(y_test, y_pred)
    print(f"Precision: {precision:.4f}")
    print(f"Recall: {recall:.4f}")
    print(f"F1 Score: {f1:.4f}")
    print("Confusion Matrix:")
    print(cm)
```

Листинг 2.2.2 - Вызов функции metrics

```
knn_uniform = KNeighborsClassifier(n_neighbors=2,
weights='uniform')
knn_uniform.fit(X_train, y_train)
y_pred_uniform = knn_uniform.predict(X_test)
metrics(y_test, y_pred_uniform)
```

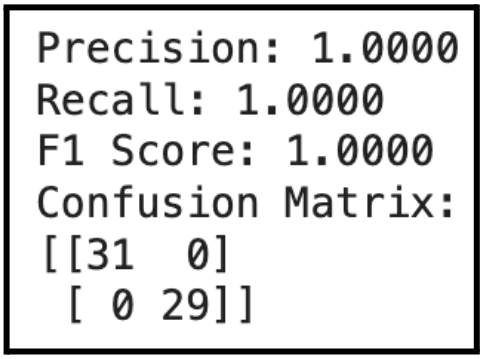

Рисунок 2.2.2 - Вычисление Precision,Recall,F1

Значения метрик сходятся с результатами таблицы ошибок. Можно заметить, что все определено верно, поэтому метод с данным набором данных справился очень хорошо.

3. Логистическая регрессия

В контексте логистической регрессии из библиотеки scikit-learn, параметры penalty, solver и max_iter используются для настройки модели. Вот краткое объяснение каждого из них:

penalty (указывает тип регуляризации, применяемой к модели. Регуляризация нужна для предотвращения переобучения и улучшения обобщающей способности модели):

'none': Без регуляризации.

'l1': L1-регуляризация (Lasso)

'l2': L2-регуляризация (Ridge)

'elasticnet': Сочетание L1 и L2 регуляризаций (только для некоторых солверов).

solver (указывает алгоритм, используемый для оптимизации задачи логистической регрессии):

'lbfgs': Алгоритм Broyden-Fletcher-Goldfarb-Shanno (BFGS).

'liblinear': Линейный солвер для малых датасетов.

'newton-cg': Метод Ньютона-Канторовича.

'sag': Стохастический усредненный градиент. Эффективен для больших датасетов.

'saga': Модификация SAG. Работает с большими датасетами.

max_iter (задает максимальное количество итераций, которое алгоритм будет выполнять для достижения сходимости. Если алгоритм не сходится до

указанного числа итераций, он остановится и вернет результат). Значение по умолчанию: 100.

3.1. Проведем классификацию методом логистической регрессии при различных параметрах: без регуляризации, с l1 регуляризацией, с l2 регуляризацией (см. листинг 3.1). Построим столбчатую диаграмму зависимости точности (Accuracy) от наличия регуляризации (см. рисунок 3.1).

Листинг 3.1 - Классификация методом логистической регрессии при различных параметрах

```
# Без регуляризации (penalty='none')
log_reg_none = LogisticRegression(penalty= None, solver='saga',
max_iter=1000)
log_reg_none.fit(X_train, y_train)
y_pred_none = log_reg_none.predict(X_test)
accuracy_none = accuracy_score(y_test, y_pred_none)

# С L1 регуляризацией (penalty='l1')
log_reg_l1 = LogisticRegression(penalty='l1', solver='saga',
max_iter=1000)
log_reg_l1.fit(X_train, y_train)
y_pred_l1 = log_reg_l1.predict(X_test)
accuracy_l1 = accuracy_score(y_test, y_pred_l1)

# С L2 регуляризацией (penalty='l2')
log_reg_l2 = LogisticRegression(penalty='l2', solver='saga',
max_iter=1000)
log_reg_l2.fit(X_train, y_train)
y_pred_l2 = log_reg_l2.predict(X_test)
accuracy_l2 = accuracy_score(y_test, y_pred_l2)

# Подготовка данных для диаграммы
methods = ['No Regularization', 'L1 Regularization', 'L2
Regularization']
accuracies = [accuracy_none, accuracy_l1, accuracy_l2]

# Построение столбчатой диаграммы
plt.bar(methods, accuracies, color=['blue', 'green', 'red'])
plt.xlabel('Regularization Method')
plt.ylabel('Accuracy')
plt.ylim(0, 1)  # Устанавливаем диапазон значений по оси y от 0
до 1
plt.grid()
plt.show()
```

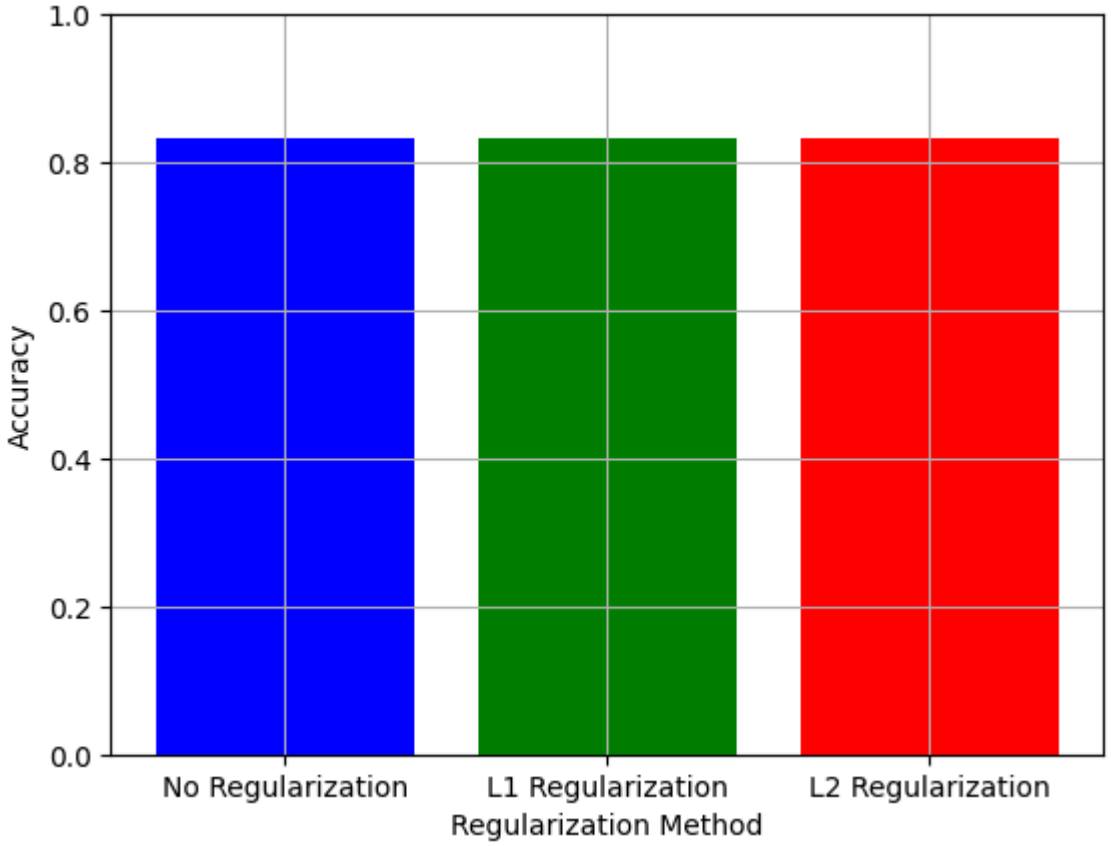

Рисунок 3.1 - Столбчатая диаграмма зависимости точности от наличия регуляризации

По рисунку 3.1 видно, что в нашем случае точность (Accuracy) одинакова для всех регуляризаций, поэтому выберем No Regularization.

3.2. Рассчитаем значения Precision,Recall,F1 для полученных результатов. Для этого воспользуемся функцией, представленную в листинге 2.2.1. Вызов функции представлен в листинге 3.2. Результат работы смотреть на рисунке 3.2.

Листинг 3.2 - Вызов функции metrics

```
log_reg_none = LogisticRegression(penalty= None, solver='saga',
max_iter=1000)
log_reg_none.fit(X_train, y_train)
y_pred_none = log_reg_none.predict(X_test)
```

```
metrics(y_test, y_pred_none)
```

```
Precision: 0.8384
Recall: 0.8354
F1 Score: 0.8331
Confusion Matrix:
[[24  7]
 [ 3 26]]
```

Рисунок 3.2 - Вычисление Precision,Recall,F1

Значения метрик сходятся с результатами таблицы ошибок.

Стоит обратить внимание на немалое количество ложноотрицательных результатов (False Negatives), поэтому метод с данным набором данных справился не очень хорошо.

4. Метод опорных векторов (SVM)

Виды ядер:

1) Линейное (linear)
2) Полиномиальное (poly) - позволяет строить кривые линии в линейном пространстве
3) Гаусово (rbf) - позволяет оценивать близость точек на основе соответствия нормально распределению
4) Сигмоидальное (sigmoid) - не удовлетворяет всем условиям ядра, но на практике работает хорошо

4.1. Проведем классификацию методом опорных векторов при различных параметрах ядра: “linear”, “poly” (нужно выбрать степень), “rbf” (см. листинг 4.1 с комментариями к коду). Построим столбчатую диаграмму зависимости точности (Accuracy) от вида ядра (см. рисунок 4.1).

Листинг 4.1 - Классификация методом опорных векторов при различных параметрах ядра

```
# Создание списка параметров ядра для SVM
kernels = ['linear', 'poly', 'rbf']
poly_degrees = [2, 3, 4, 5]  # Список степеней для
полиномиального ядра

# Списки для хранения точности и меток
accuracies = []
x_labels = []

# Обучение моделей SVM с различными ядрами
for kernel in kernels:
    if kernel == 'poly':
        for degree in poly_degrees:
            # создание модели
            svm_model = SVC(kernel=kernel, degree=degree)
            # обучение
            svm_model.fit(X_train, y_train)
            # предсказание
            y_pred = svm_model.predict(X_test)
            # расчет точности
            accuracy = accuracy_score(y_test, y_pred)
            accuracies.append(accuracy)
            x_labels.append(f'{kernel} (degree={degree})')
    else:
        svm_model = SVC(kernel=kernel)
        svm_model.fit(X_train, y_train)
        y_pred = svm_model.predict(X_test)
        accuracy = accuracy_score(y_test, y_pred)
        accuracies.append(accuracy)
        x_labels.append(kernel)

# Построение диаграммы
x_positions = np.arange(len(x_labels)) # для выравнивания
столбцов
plt.figure(figsize=(10, 6))
# выравнивание столбцов по центру относительно x_positions,
высота accuracies
plt.bar(x_positions, accuracies, align='center')
plt.xticks(x_positions, x_labels)
plt.xlabel('Kernel Type')
plt.ylabel('Accuracy')
plt.grid()
plt.show()
```

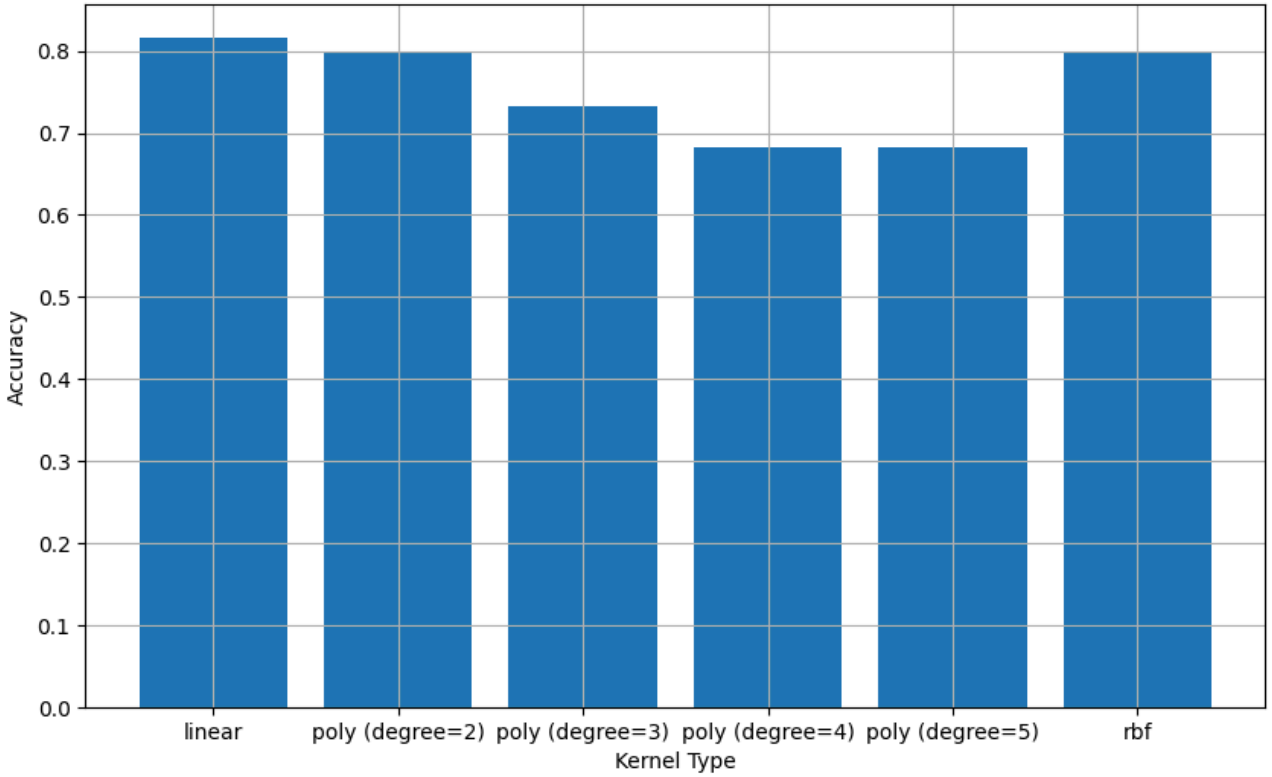

Рисунок 4.1 - Столбчатая диаграмма зависимости точности от вида ядра

Самая высокая точность получилась при параметре ядра linear (более 0.8).

4.2. Рассчитаем значения Precision,Recall,F1 для полученных результатов. Для этого воспользуемся функцией, представленную в листинге 2.2.1. Вызов функции представлен в листинге 4.2. Результат работы смотреть на рисунке 4.2.

Листинг 4.2 - Вызов функции metrics

```
svm_model = SVC(kernel='linear')
svm_model.fit(X_train, y_train)
y_pred = svm_model.predict(X_test)
metrics(y_test, y_pred)
```

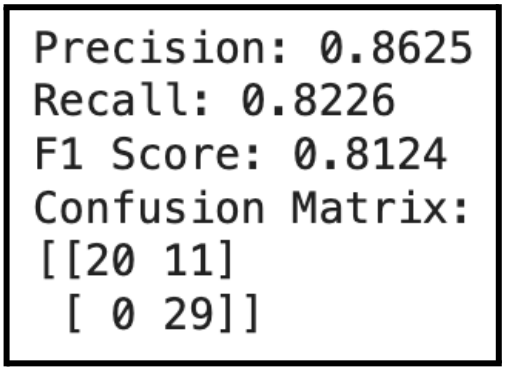

Рисунок 4.2 - Вычисление Precision,Recall,F1

Значения метрик сходятся с результатами таблицы ошибок.

Стоит обратить внимание на большое количество ложноотрицательных результатов (False Negatives), поэтому метод с данным набором данным справился плохо.

5. Решающие деревья

Объект GridSearchCV из библиотеки scikit-learn — это инструмент для поиска наилучших гиперпараметров модели машинного обучения. Он выполняет систематический перебор (grid search) по указанным наборам значений гиперпараметров и выбирает комбинацию, которая дает наилучшие результаты.

5.1. Проведем классификацию используя решающие деревья, подобрав параметры при которых получается лучшее обобщение (максимальная глубина/максимальное количество листьев/метрика загрязнения и т.д.). Будем использовать 4 параметра: Максимальная глубина, Минимальное/максимальное кол-во наблюдений в листе, Минимальное кол-во наблюдений в узле для разбиения, Достигнут удовлетворяющий уровень загрязнения. Реализация поиска лучших параметров с пояснениями представлена в листинге 5.1.

Листинг 5.1 - Подбор параметров для классификации (решающие деревья)

```
params = {
    'max_depth': [5, 10, 15],
    'min_samples_split': [2, 3, 4, 5],
    'min_samples_leaf': [2, 3, 4],
    'criterion': ['gini', 'entropy']
}

# Создание модели решающего дерева
tree_classifier = DecisionTreeClassifier()

# Создание объекта GridSearchCV для поиска лучших параметров
grid_search = GridSearchCV(tree_classifier, params,
scoring='accuracy')

# Обучение модели на обучающих данных
grid_search.fit(X_train, y_train)

# Получение лучших параметров
best_params = grid_search.best_params_
print("Лучшие параметры:", best_params)

# Использование лучшей модели для предсказания на тестовых данных
best_classifier = grid_search.best_estimator_
y_pred = best_classifier.predict(X_test)

# Оценка точности на тестовых данных
test_accuracy = accuracy_score(y_test, y_pred)
print("Точность:", test_accuracy)
```

Результат поиска представлен на рисунке 5.1.

```
Лучшие параметры: {'criterion': 'gini', 'max_depth': 10, 'min_samples_leaf': 2, 'min_samples_split': 3}
Точность: 0.95
```

Рисунок 5.1 - Лучшие параметры и точность

Точность вышла хорошая.

5.2. Рассчитаем значения Precision,Recall,F1 для полученных результатов. Для этого воспользуемся функцией, представленную в листинге 2.2.1. Вызов функции представлен в листинге 5.2. Результат работы смотреть на рисунке 5.2.

Листинг 5.2 - Вызов функции metrics

```
tree_classifier = DecisionTreeClassifier(criterion='gini',
max_depth=10, min_samples_leaf=2, min_samples_split=3)
tree_classifier.fit(X_train, y_train)
y_pred = tree_classifier.predict(X_test)
metrics(y_test, y_pred)
```

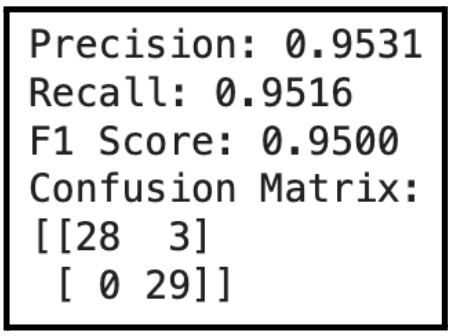

Рисунок 5.2 - Вычисление Precision,Recall,F1

Значения метрик сходятся с результатами таблицы ошибок.

По рисунку 5.2 видно, что есть небольшое количество ложноотрицательных результатов (False Negatives), но все равно метод с данным набором данных справился хорошо.

6. Выбор классификатора

6.1. Построим таблицу с метриками для полученных результатов каждым классификатором (см. таблицу 6.1).

Таблица 6.1 - Метрики, полученные каждым классификатором

Классификатор	Precision	Recall	F1 Score
kNN	1.0000	1.0000	1.0000
Логистическая регрессия	0.8384	0.8354	0.8331
Метод опорных векторов (SVM)	0.8625	0.8226	0.8124
Решающие деревья	0.9531	0.9516	0.9500

6.2. Сделаем выводы о том, какие классификаторы лучше всего подходят для вашего набора данных. Анализируя таблицу 6.1, можно сделать вывод, что для нашего нашего набора данных лучше всего подошел классификатор kNN, затем решающие деревья также справился неплохо, а уже не очень хорошо справились логистическая регрессия и SVM.

Вывод.

В ходе выполнения работы работы были изучены различные методы классификации: KNN, Логистическая регрессия, Метод опорных векторов и Решающие деревья. Для каждого классификатора были рассчитаны точность и метрики. Для нашего набора данных самым успешным оказался метод классификации KNN.