Факультет программной инженерии и компьютерной техники

Системы искусственного интеллекта

Отчёт по Модулю 2

Преподаватель: Кугаевских Александр Владимирович
Выполнил: Ле Чонг Дат
Группа: P33302

2023 г.

Лабораторная 4. Линейная регрессия

Линейная регрессия. Теоретическая часть

Задание

- Выбор датасетов:
  - Студенты с **нечетным** порядковым номером в группе должны использовать про обучение студентов
- Получите и визуализируйте (графически) статистику по датасету (включая количество, среднее значение, стандартное отклонение, минимум, максимум и различные квантили).
- Проведите предварительную обработку данных, включая обработку отсутствующих значений, кодирование категориальных признаков и нормировка.
- Разделите данные на обучающий и тестовый наборы данных.
- Реализуйте линейную регрессию с использованием метода наименьших квадратов без использования сторонних библиотек, кроме NumPy и Pandas (для использования коэффициентов использовать библиотеки тоже нельзя). Использовать минимизацию суммы квадратов разностей между фактическими и предсказанными значениями для нахождения оптимальных коэффициентов.
- Постройте **три модели** с различными наборами признаков.
- Для каждой модели проведите оценку производительности, используя метрику коэффициент детерминации, чтобы измерить, насколько хорошо модель соответствует данным.
- Сравните результаты трех моделей и сделайте выводы о том, какие признаки работают лучше всего для каждой модели.
- Бонусное задание
  - Ввести синтетический признак при построении модели

Step 1: Installing Libraries

This step is usually done once. If you already have these libraries installed, you can skip this step.

In [1]:
```
# Install necessary libraries
# !pip install pandas matplotlib numpy
```

Step 2: Importing Libraries and define helper functions

In [2]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

```
--------------------------------------------------------------------------------
---------------
Handling Missing Values
--------------------------------------------------------------------------------
---------------
Display the first few rows of the processed dataset
   Hours Studied  Previous Scores  Extracurricular Activities  Sleep Hours  \
0          0.750         1.000000                         1.0          1.0
1          0.375         0.711864                         0.0          0.0
2          0.875         0.186441                         1.0          0.6
3          0.500         0.203390                         1.0          0.2
4          0.750         0.593220                         0.0          0.8

   Sample Question Papers Practiced  Performance Index
0                          0.111111           0.900000
1                          0.222222           0.611111
2                          0.222222           0.388889
3                          0.222222           0.288889
4                          0.555556           0.622222
--------------------------------------------------------------------------------
---------------
Display the sizes of the processed dataset
(10000, 6)
```

After completing the preliminary data processing, the size of the dataset remains unchanged, indicating that the dataset does not contain any null values. Additionally, the 'Yes' and 'No' values in the 'Extracurricular Activities' column have been successfully converted to numerical values 0 and 1, respectively.

Task 3: Разделите данные на обучающий и тестовый наборы данных.

In [9]:
```python
# Assuming 'Performance Index' is the target variable
X = df.drop('Performance Index', axis=1)  # Features
y = df['Performance Index']               # Target

# Split the data into training and testing sets
# test_size defines the proportion of the test set, usually between 0.2 and 0.3
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta

# Display the sizes of the training and test sets
print("Training set size:", X_train.shape)
print("Test set size:", X_test.shape)
```

```
Training set size: (8000, 5)
Test set size: (2000, 5)
```

Task 4: Реализуйте линейную регрессию с использованием метода наименьших квадратов без использования сторонних библиотек, кроме NumPy и Pandas (для использования коэффициентов использовать библиотеки тоже нельзя). Использовать минимизацию суммы квадратов разностей между фактическими и предсказанными значениями для нахождения оптимальных коэффициентов.

In [10]:
```
# We need to add a column of ones to X_train for the intercept term.
X_train = np.append(np.ones((X_train.shape[0], 1)), X_train, axis=1)
X_test = np.append(np.ones((X_test.shape[0], 1)), X_test, axis=1)
```

In [11]:
```
# Calculate the Coefficients Using the Least Squares Method
def least_squares(X, y):
    X_transpose = X.T
    coefficients = np.linalg.inv(X_transpose.dot(X)).dot(X_transpose).dot(y)
    return coefficients

b = least_squares(X_train, y_train)
```

In [12]:
```
# Making Predictions
def predict(X, b):
    return X.dot(b)

y_test_pred = predict(X_test, b)
```

In [13]:
```
# Evaluating the model on the test set
mse_test = np.mean((y_test - y_test_pred)**2)
r_squared_test = r_squared(y_test, y_test_pred)

print(f"Test Mean Squared Error: {mse_test}")
print(f"Test R-squared: {r_squared_test}")
```

```
Test Mean Squared Error: 0.0005040281973483783
Test R-squared: 0.9889832909573145
```

Task 5: Постройте три модели с различными наборами признаков.

In [14]:
```
# Assuming these are the column names in your dataframe 'df'
features_set_1 = ['Hours Studied', 'Previous Scores']
features_set_2 = ['Hours Studied', 'Previous Scores', 'Extracurricular Activities']
features_set_3 = ['Hours Studied', 'Previous Scores', 'Sleep Hours', 'Sample Questi
```

In [15]:
```
# Splitting the first set of features
X1 = df[features_set_1]
X1_train, X1_test, y1_train, y1_test = train_test_split(X1, y, test_size=0.2, rando

# Splitting the second set of features
X2 = df[features_set_2]
X2_train, X2_test, y2_train, y2_test = train_test_split(X2, y, test_size=0.2, rando

# Splitting the third set of features
X3 = df[features_set_3]
X3_train, X3_test, y3_train, y3_test = train_test_split(X3, y, test_size=0.2, rando
```

In [16]:
```
# Training the Models
# Function for adding a column of ones (for the intercept) and computing coefficien
def prepare_and_train(X, y):
    X = np.append(np.ones((X.shape[0], 1)), X, axis=1)
    return least_squares(X, y)

# Training each model
```

Лабораторная 5. **Метод k-ближайших соседей**

Задание

- Выбор датасета:
  - Нечетный номер в группе - Датасет про диабет
- Проведите предварительную обработку данных, включая обработку отсутствующих значений, кодирование категориальных признаков и масштабирование.
- Получите и визуализируйте (графически) статистику по датасету (включая количество, среднее значение, стандартное отклонение, минимум, максимум и различные квантили), постройте 3d-визуализацию признаков.
- Реализуйте метод k-ближайших соседей ****без использования сторонних библиотек, кроме NumPy и Pandas.
- Постройте две модели k-NN с различными наборами признаков:
  - Модель 1: Признаки случайно отбираются .
  - Модель 2: Фиксированный набор признаков, который выбирается заранее.
- Для каждой модели проведите оценку на тестовом наборе данных при разных значениях k. Выберите несколько различных значений k, например, k=3, k=5, k=10, и т. д. Постройте матрицу ошибок.

Step 1: Installing Libraries

This step is usually done once. If you already have these libraries installed, you can skip this step.

In [1]:
```
# Install necessary libraries
# !pip install pandas numpy sklearn
```

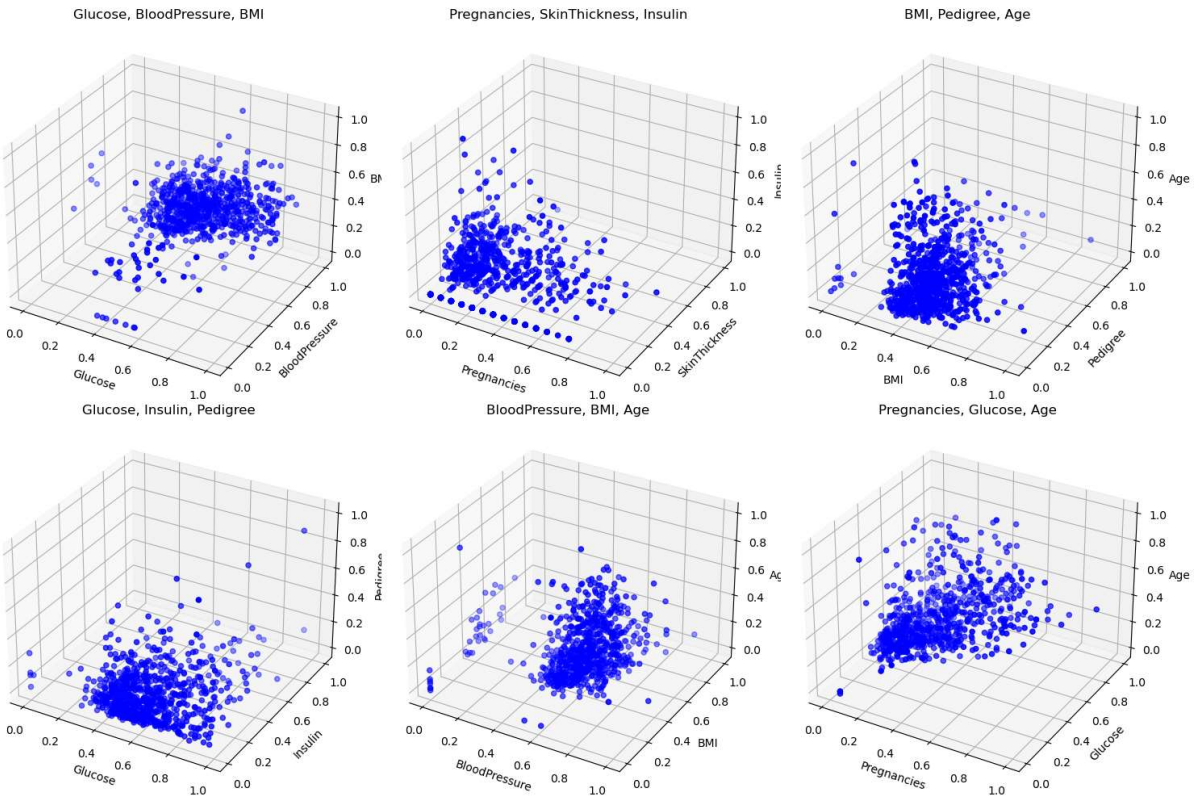
Step 2: Importing Libraries and define helper functions

In [2]:
```
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.impute import SimpleImputer
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats
```

In [3]:
```
def print_bold(text):
    BOLD = '\033[1m'
    END = '\033[0m'
    print('-' * 100 + f"\n{BOLD}{text}{END}")
```

```
# Create 3D plots for each combination in the grid
for i, features in enumerate(feature_combinations):
    ax = fig.add_subplot(n_rows, n_cols, i + 1, projection='3d')
    ax.scatter(df[features[0]], df[features[1]], df[features[2]], c='blue', marker=
    ax.set_xlabel(features[0])
    ax.set_ylabel(features[1])
    ax.set_zlabel(features[2])
    ax.set_title(f"{features[0]}, {features[1]}, {features[2]}")

# Adjust layout
plt.tight_layout()
plt.show()
```



Task 3: Реализуйте метод k-ближайших соседей без использования сторонних библиотек, кроме NumPy и Pandas.

In [12]:
```
def euclidean_distance(row1, row2):
    # Calculating the Euclidean distance between two rows
    return np.sqrt(np.sum((row1 - row2) ** 2))

def k_nearest_neighbors(train, test_row, k):
    # Finding the k nearest neighbors
    distances = []
    for train_row in train:
        dist = euclidean_distance(test_row[:-1], train_row[:-1])  # Ignore the labe
        distances.append((train_row, dist))
    distances.sort(key=lambda tup: tup[1])

    neighbors = []
    for i in range(k):
        neighbors.append(distances[i][0])
```

```
    # Aggregating the labels of neighbors
    labels = [neighbor[-1] for neighbor in neighbors]

    # Majority vote for the most common label
    prediction = max(set(labels), key=labels.count)
    return prediction
```

Task 4:

- Постройте две модели k-NN с различными наборами признаков:
  - Модель 1: Признаки случайно отбираются .
  - Модель 2: Фиксированный набор признаков, который выбирается заранее.

In [13]:
```
# Splitting the data into training and testing sets
train_df = df.sample(frac=0.8, random_state=1)  # 80% of data for training
test_df = df.drop(train_df.index)  # Remaining 20% for testing

# Model 1: Random Feature Selection
# Randomly select 3 features (excluding the last one, which is assumed to be the la
random_features = np.random.choice(df.columns[:-1], size=3, replace=False)
print_bold("Random features:")
print(random_features)
# Create training and testing sets with these features and the 'Outcome' label
train_random = train_df[random_features].join(train_df['Outcome'])
test_random = test_df[random_features].join(test_df['Outcome'])

# Model 2: Fixed Feature Selection
# Selecting specific features based on domain knowledge or prior analysis
# Glucose: Typically, higher glucose levels are a significant indicator of diabetes

# BMI (Body Mass Index): BMI is a key factor because it is often correlated with th

# Age: Age is another important factor. The risk of developing type 2 diabetes incr
fixed_features = ['Glucose', 'BMI', 'Age']
print_bold("Fixed features:")
print(fixed_features)
# Create training and testing sets with these fixed features and the 'Outcome' labe
train_fixed = train_df[fixed_features].join(train_df['Outcome'])
test_fixed = test_df[fixed_features].join(test_df['Outcome'])

# Function to test a model's accuracy
def test_model(train_set, test_set, k):
    correct = 0
    for test_row in test_set.values:
        # Predict the class of each test row
        prediction = k_nearest_neighbors(train_set.values, test_row, k)
        # Check if the prediction matches the actual label
        if prediction == test_row[-1]:  # Assuming the class label is the last colu
            correct += 1
    # Calculate accuracy as the proportion of correct predictions
    accuracy = correct / len(test_set)
    return accuracy

# Testing both models with k=3 neighbors
```

```
k = 3
accuracy_random = test_model(train_random, test_random, k)
accuracy_fixed = test_model(train_fixed, test_fixed, k)

# Print the accuracy of both models
print(f"Accuracy of Model 1 (Random Features): {accuracy_random}")
print(f"Accuracy of Model 2 (Fixed Features): {accuracy_fixed}")
```

```
--------------------------------------------------------------------------------
----------------
Random features:
['Age' 'Insulin' 'Glucose']
--------------------------------------------------------------------------------
----------------
Fixed features:
['Glucose', 'BMI', 'Age']
Accuracy of Model 1 (Random Features): 0.6688311688311688
Accuracy of Model 2 (Fixed Features): 0.6948051948051948
```

Task 5: Для каждой модели проведите оценку на тестовом наборе данных при разных значениях k. Выберите несколько различных значений k, например, k=3, k=5, k=10, и т. д. Постройте матрицу ошибок.

In [14]:
```
def confusion_matrix(true, pred):
    # Calculate confusion matrix components
    true = np.array(true)
    pred = np.array(pred)
    TP = sum((true == 1) & (pred == 1))  # True Positives
    TN = sum((true == 0) & (pred == 0))  # True Negatives
    FP = sum((true == 0) & (pred == 1))  # False Positives
    FN = sum((true == 1) & (pred == 0))  # False Negatives

    # Create a DataFrame for the confusion matrix
    conf_matrix_df = pd.DataFrame(
        [[TP, FP],
         [FN, TN]],
        columns=['Predicted Positive', 'Predicted Negative'],
        index=['Actual Positive', 'Actual Negative']
    )
    return conf_matrix_df

def evaluate_model(train_set, test_set, k_values):
    # Evaluate the model for different values of k and return confusion matrices
    true_labels = test_set[:, -1]
    confusion_matrices = []

    for k in k_values:
        # Make predictions for each test row
        predictions = [k_nearest_neighbors(train_set, test_row, k) for test_row in
        # Calculate and store the confusion matrix for this k value
        confusion_matrices.append(confusion_matrix(true_labels, predictions))

    return confusion_matrices

# Define different values of k
```

Лабораторная 6. Деревья решений

Деревья решений. Теоретическая часть (1)

Задание

- Для студентов с четным порядковым номером в группе – датасет с классификацией грибов, а нечетным – датасет с данными про оценки студентов инженерного и педагогического факультетов (для данного датасета нужно ввести метрику: студент успешный/неуспешный на основании грейда)
- Отобрать **случайным** образом sqrt(n) признаков
- Реализовать без использования сторонних библиотек построение дерева решений (дерево не бинарное, numpy и pandas использовать можно, использовать списки для реализации дерева - нельзя)
- Провести оценку реализованного алгоритма с использованием Accuracy, precision и recall
- Построить AUC-ROC и AUC-PR (в пунктах 4 и 5 использовать библиотеки нельзя)

Step 1: Installing libraries

This step is usually done once. If you already have these libraries installed, you can skip this step

In [1]:
```
# !pip install pandas numpy scikit-learn matplotlib
```

Step 2: Importing libraries and defining helper functions

In [2]:
```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
from sklearn.preprocessing import label_binarize
```

In [3]:
```
def print_bold(text):
    BOLD = '\033[1m'
    END = '\033[0m'
    print('-' * 100 + f"\n{BOLD}{text}{END}")
```

Step 3: Performing tasks

Task 1: Отобрать случайным образом sqrt(n) признаков

In [4]:
```
import pandas as pd
```

[5 rows x 32 columns]

In [5]:
```
# First, separate the features and the target
X = df.drop('GRADE', axis=1)
y = df['GRADE']

# Calculate the number of features to select: sqrt(n)
num_features = int(np.sqrt(X.shape[1]) * 2)

# Randomly select sqrt(n) features
selected_features = np.random.choice(X.columns, num_features, replace=False)
X_selected = X[selected_features]

# Displaying the selected features
X_selected.head()
```

Out[5]:

STUDENT ID	12	7	COURSE ID	21	1	14	24	9	3	26	20
STUDENT1	2	2		1	1	2	1	1	1	3	2	1
STUDENT2	3	2		1	1	2	1	1	1	3	2	1
STUDENT3	2	2		1	1	2	1	1	4	2	2	1
STUDENT4	2	1		1	1	1	1	2	1	1	2	1
STUDENT5	3	1		1	1	2	1	1	1	1	2	1

Task 2: Реализовать без использования сторонних библиотек построение дерева решений (дерево не бинарное, numpy и pandas использовать можно, использовать списки для реализации дерева - нельзя)

In [6]:
```
class DecisionNode:
    def __init__(self, feature_index=None, branches=None, value=None, most_common=N
        self.feature_index = feature_index  # Index of the feature for splitting
        self.branches = branches            # Dictionary of branches
        self.value = value                  # Value at a leaf node (if it is a leaf
        self.most_common = most_common
        self.class_probabilities = class_probabilities
        self.num_classes = num_classes

class DecisionTree:
    def __init__(self, max_depth=None):
        self.max_depth = max_depth
        self.root = None

    def fit(self, X, y):
        unique_classes = np.unique(y)
        self.root = self._build_tree(X, y, depth=0, num_classes=len(unique_classes)

    def _build_tree(self, X, y, depth, num_classes):
        num_samples, num_features = X.shape
```

```
        unique_classes = np.unique(y)

        # Stopping conditions
        if depth >= self.max_depth or len(unique_classes) == 1 or num_samples == 0:
            leaf_value = self._calculate_leaf_value(y)
            class_probabilities = self._calculate_class_probabilities(y, num_classe
            return DecisionNode(value=leaf_value, class_probabilities=class_probabi

        # Find the best split
        best_feature = self._find_best_split(X, y, num_features)

        # Creating branches for a non-binary tree
        branches = {}
        most_common = self._calculate_most_common_value(y)
        class_probabilities = self._calculate_class_probabilities(y, num_classes)
        for feature_value in np.unique(X[:, best_feature]):
            indices = X[:, best_feature] == feature_value
            X_subset, y_subset = X[indices], y[indices]
            branches[feature_value] = self._build_tree(X_subset, y_subset, depth +

        return DecisionNode(feature_index=best_feature, branches=branches, most_com

    def _calculate_most_common_value(self, y):
        unique_classes, class_counts = np.unique(y, return_counts=True)
        return unique_classes[class_counts.argmax()]

    def _calculate_class_probabilities(self, y, num_classes):
        class_counts = np.bincount(y, minlength=num_classes)
        class_probabilities = class_counts / np.sum(class_counts)
        return class_probabilities

    def _calculate_leaf_value(self, y):
        unique_classes, class_counts = np.unique(y, return_counts=True)
        return unique_classes[class_counts.argmax()]

    def _find_best_split(self, X, y, num_features):
        best_feature = None
        best_gain = -float('inf')

        for feature_index in range(num_features):
            X_column = X[:, feature_index]
            gain = self._information_gain(X_column, y)

            if gain > best_gain:
                best_gain = gain
                best_feature = feature_index

        return best_feature

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

    def _information_gain(self, X_column, y):
        # Total entropy before the split
        total_entropy = self._entropy(y)
```

```
        # Values and counts in this column
        values, counts = np.unique(X_column, return_counts=True)

        # Weighted entropy of branches
        weighted_entropy = sum(
            (counts[i] / len(X_column)) * self._entropy(y[X_column == value])
            for i, value in enumerate(values)
        )

        # Information gain is the total entropy minus the weighted entropy of the b
        return total_entropy - weighted_entropy

    def predict(self, X):
        predictions = []
        for data_point in X:
            node = self.root
            while node.value is None:
                feature_value = data_point[node.feature_index]
                if feature_value in node.branches:
                    node = node.branches[feature_value]
                else:
                    # Handling missing branches
                    node = self._handle_missing_branch(node)
            predictions.append(node.value)
        return np.array(predictions)

    def predict_proba(self, X):
        proba_predictions = []
        for data_point in X:
            node = self.root
            while node.value is None:
                feature_value = data_point[node.feature_index]
                if feature_value in node.branches:
                    node = node.branches[feature_value]
                else:
                    node = self._handle_missing_branch(node)
            proba_predictions.append(node.class_probabilities)
        return np.array(proba_predictions)


    def _handle_missing_branch(self, node):
        # Return the most common value of the current node
        return DecisionNode(value=node.most_common, class_probabilities=node.class_
```

In [7]:

```
# Splitting the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_selected, y, test_size=0.2, r

# Initialize the Decision Tree with a maximum depth of 3
tree = DecisionTree(max_depth=3)

# Fit the Decision Tree model to the training data
tree.fit(X_train.values, y_train.values)
```

Task 3: Провести оценку реализованного алгоритма с использованием Accuracy, precision и recall

In [8]:
```
def accuracy(y_true, y_pred):
    correct_predictions = np.sum(y_true == y_pred)
    total_predictions = len(y_true)
    return correct_predictions / total_predictions

def precision_recall_per_class(y_true, y_pred, class_label):
    true_positives = np.sum((y_true == class_label) & (y_pred == class_label))
    predicted_positives = np.sum(y_pred == class_label)
    actual_positives = np.sum(y_true == class_label)

    precision = true_positives / predicted_positives if predicted_positives > 0 els
    recall = true_positives / actual_positives if actual_positives > 0 else 0

    return precision, recall

def average_precision_recall(y_true, y_pred):
    unique_classes = np.unique(y_true)
    precision_sum = 0
    recall_sum = 0

    for class_label in unique_classes:
        precision, recall = precision_recall_per_class(y_true, y_pred, class_label)
        precision_sum += precision
        recall_sum += recall

    average_precision = precision_sum / len(unique_classes)
    average_recall = recall_sum / len(unique_classes)

    return average_precision, average_recall
```

In [9]:
```
y_pred = tree.predict(X_test.values)
y_proba = tree.predict_proba(X_test.values)

accuracy_value = accuracy(y_test, y_pred)

precision_value, recall_value = average_precision_recall(y_test, y_pred)

print(f"Accuracy: {accuracy_value}")
print(f"Precision: {precision_value}")
print(f"Recall: {recall_value}")
```

```
Accuracy: 0.1724137931034483
Precision: 0.22916666666666666
Recall: 0.19791666666666666
```

Task 4: Построить AUC-ROC и AUC-PR (в пунктах 4 и 5 использовать библиотеки нельзя)

In [10]:
```
def calculate_tpr_fpr(y_true, y_scores, thresholds):
    tpr_values = []
    fpr_values = []
```

Лабораторная 7. Логистическая регрессия

логистическая регрессия.docx

1. Выбор датасета:

- Датасет о пассажирах Титаника: Titanic Dataset
- Датасет о диабете: Diabetes Dataset
- Загрузите выбранный датасет и выполните предварительную обработку данных.
- Получите и визуализируйте (графически) статистику по датасету (включая количество, среднее значение, стандартное отклонение, минимум, максимум и различные квантили).
- Разделите данные на обучающий и тестовый наборы в соотношении, которое вы считаете подходящим.
- 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.

1. 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).
2. 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.

Step 1: Installing libraries

```
   Pregnancies   Glucose  BloodPressure  SkinThickness   Insulin       BMI  \
0     0.352941  0.743719       0.590164       0.353535  0.000000  0.500745
1     0.058824  0.427136       0.540984       0.292929  0.000000  0.396423
2     0.470588  0.919598       0.524590       0.000000  0.000000  0.347243
3     0.058824  0.447236       0.540984       0.232323  0.111111  0.418778
4     0.000000  0.688442       0.327869       0.353535  0.198582  0.642325

   DiabetesPedigreeFunction       Age  Outcome
0                  0.234415  0.483333        1
1                  0.116567  0.166667        0
2                  0.253629  0.183333        1
3                  0.038002  0.000000        0
4                  0.943638  0.200000        1
```

Task 2: Разделите данные на обучающий и тестовый наборы в соотношении, которое вы считаете подходящим.

In [11]:
```
X = df.drop('Outcome', axis=1)  # Features
y = df['Outcome']  # Target variable

# Splitting the dataset into the Training set and Test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_sta
```

Task 3:

- 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.

In [12]:
```
import numpy as np

class LogisticRegression:
    def __init__(self, learning_rate=0.01, max_iter=100, optimization_method='gradi
        self.learning_rate = learning_rate
        self.max_iter = max_iter
        self.optimization_method = optimization_method
        self.weights = None

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

    def _cost_function(self, X, y):
        m = len(y)
        h = self._sigmoid(X @ self.weights)
        return -1/m * np.sum(y * np.log(h) + (1 - y) * np.log(1 - h))

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

```
        m = len(y)
        for _ in range(self.max_iter):
            h = self._sigmoid(X @ self.weights)
            gradient = np.dot(X.T, (h - y)) / m
            self.weights -= self.learning_rate * gradient

    def _newton_method(self, X, y):
        m = len(y)
        for _ in range(self.max_iter):
            h = self._sigmoid(X @ self.weights)
            gradient = np.dot(X.T, (h - y)) / m
            # Create a diagonal matrix for element-wise multiplication
            S = np.diag(h * (1 - h))
            H = X.T @ S @ X / m
            self.weights -= np.linalg.inv(H) @ gradient

    def fit(self, X, y):
        X = np.insert(X, 0, 1, axis=1)  # Add intercept
        self.weights = np.zeros(X.shape[1])
        if self.optimization_method == 'gradient_descent':
            self._gradient_descent(X, y)
        elif self.optimization_method == 'newton':
            self._newton_method(X, y)

    def predict(self, X):
        X = np.insert(X, 0, 1, axis=1)  # Add intercept
        probabilities = self._sigmoid(X @ self.weights)
        return np.where(probabilities >= 0.5, 1, 0)
```

In [13]:

```
def calculate_accuracy(y_true, y_pred):
    correct = sum([1 for true, pred in zip(y_true, y_pred) if true == pred])
    return correct / len(y_true)

def calculate_precision(y_true, y_pred):
    true_positives = sum([1 for true, pred in zip(y_true, y_pred) if true == pred =
    predicted_positives = sum([1 for pred in y_pred if pred == 1])
    return true_positives / predicted_positives if predicted_positives != 0 else 0

def calculate_recall(y_true, y_pred):
    true_positives = sum([1 for true, pred in zip(y_true, y_pred) if true == pred =
    actual_positives = sum([1 for true in y_true if true == 1])
    return true_positives / actual_positives if actual_positives != 0 else 0

def calculate_f1_score(y_true, y_pred):
    precision = calculate_precision(y_true, y_pred)
    recall = calculate_recall(y_true, y_pred)
    return 2 * (precision * recall) / (precision + recall) if (precision + recall)

def evaluate_model(model, X_test, y_test):
    predictions = model.predict(X_test)
    accuracy = calculate_accuracy(y_test, predictions)
    precision = calculate_precision(y_test, predictions)
    recall = calculate_recall(y_test, predictions)
    f1 = calculate_f1_score(y_test, predictions)
    return accuracy, precision, recall, f1
```

```python
# Hyperparameters to test
learning_rates = [0.01, 0.1, 0.5]
iterations = [100, 500, 1000]
optimization_methods = ['gradient_descent', 'newton']

# Store the results
results = []

# from sklearn.linear_model import LogisticRegression

for lr in learning_rates:
    for itr in iterations:
        for om in optimization_methods:
            model = LogisticRegression(learning_rate=lr, max_iter=itr, optimization
            # model = LogisticRegression(C=lr, max_iter=itr, solver='lbfgs')
            model.fit(X_train, y_train)

            # Evaluate the model
            accuracy, precision, recall, f1 = evaluate_model(model, X_test, y_test)
            results.append((lr, itr, accuracy, precision, recall, f1, om))


# Convert the results into a pandas DataFrame
results_df = pd.DataFrame(results, columns=['Learning Rate', 'Iterations', 'Accurac

# Sorting the DataFrame by the 'Accuracy' column in descending order
sorted_results_df = results_df.sort_values(by='Accuracy', ascending=False)

# Set display options to avoid line breaks in columns
pd.set_option('display.max_colwidth', None)
pd.set_option('display.expand_frame_repr', False)

# Display the sorted DataFrame
print(sorted_results_df)
```

Conclusion

This report has presented a detailed examination and comparison of four distinct machine learning methods: Linear Regression, k-Nearest Neighbors (k-NN), Decision Trees, and Logistic Regression. Each method has its unique strengths and limitations, making them suitable for various applications.

1. **Linear Regression** is particularly effective in scenarios where a continuous outcome needs to be predicted based on linear relationships. Its simplicity and interpretability make it a valuable tool for predictive modeling in areas like economics and real estate. However, its reliance on the assumption of a linear relationship between variables can be a limitation in complex scenarios where such relationships do not exist.

2. **k-Nearest Neighbors (k-NN)** excels in classification problems where the decision boundary is highly irregular. Due to its non-parametric nature, it is versatile and can be applied in various contexts, including recommendation systems and image recognition. The primary limitation of k-NN is its computational intensity, especially with large datasets, and its sensitivity to the choice of 'k' and the distance metric used.

3. **Decision Trees** are powerful for classification and regression tasks, offering ease of interpretation and handling both numerical and categorical data efficiently. They are particularly useful in operational decision-making models like credit scoring and medical diagnosis. However, they are prone to overfitting, especially in cases with numerous features.

4. **Logistic Regression** is well-suited for binary classification problems, such as email spam detection or disease diagnosis. It provides the probability of class memberships, which is informative in risk assessment. However, its performance can be limited in non-linear decision boundaries, and it requires careful consideration of feature selection to avoid overfitting or underfitting.

In conclusion, while no single method is universally superior, the choice of method depends largely on the specific characteristics of the problem at hand, including the nature of the data, the complexity of the relationships within it, and the type of prediction or classification required. Understanding the strengths and limitations of each method allows for more informed and effective application in various real-world scenarios.