# Report on Approach - Logistic Regression

In this assignment, I used Logistic Regression to predict whether patients have diabetes, based on several medical predictor variables. Below is a simple explanation of my steps:

1. **Data Loading and Exploration**
   I began by loading the dataset using Pandas and checked the data structure. The dataset contains various features, such as glucose levels, BMI, and insulin levels, which can help in predicting diabetes. I briefly inspected the dataset by checking its info and sampling some rows.

2. **Correlation Analysis**
   Next, I calculated the correlation between the features to understand how they relate to each other and the target variable (whether a patient has diabetes). This was visualized using a heatmap, which gave me an overview of the relationships and using dabl (Data Analysis Baseline Library) to visualize the relations between each variable. Features like glucose had a strong correlation with the outcome, making them important for prediction.

3. **Data Preprocessing**
   I split the dataset into training and testing sets (80% training, 20% testing) to evaluate the model's performance on unseen data. To ensure the model works effectively, I standardized the feature values using StandardScaler. This step helps by normalizing the data, ensuring that no single feature dominates the prediction due to its scale.

4. **Model Implementation**
   Using the Logistic Regression model from Scikit-Learn, I trained the model on the scaled training data. After training, I used the test data to evaluate the model's predictions.

5. **Evaluation**
   To assess the model's performance, I used metrics such as F1-score, precision, recall, and accuracy. The classification report showed that the

model performed reasonably well, especially considering this is a binary classification problem. I also plotted a confusion matrix to visualize where the model did well and where it made mistakes.

6. **Conclusion**
   Overall, Logistic Regression provided good results, with features like glucose playing a critical role in predicting diabetes. Standardization and proper evaluation helped ensure that the model was fair and accurate.

# Report on KNN Regression for Diabetes Dataset

In this project, I worked with the **Diabetes dataset** to explore how K-Nearest Neighbors (KNN) regression performs and how the number of features and neighbors (`k`) affects the model's predictions. Since I'm new to this, I aimed to keep things simple and clear while gradually building my understanding.

### 1: Loading and Preparing the Data

I began by loading the **Diabetes dataset** using `sklearn`. The dataset comes with various features, such as age, sex, BMI, and blood pressure, all of which are related to predicting a target variable—essentially a health metric for diabetes.

I first converted the sex feature to binary values (from `[1, 2]` to `[0, 1]`), which simplifies the model's work. I then added the target variable (the diabetes progression metric) to the dataset and took a quick peek at the first few rows.

### 2: Data Visualization

To get a sense of how the features relate to each other, I created a **correlation matrix** using `seaborn`. This heatmap showed me how strongly each feature correlates with the target variable. Some features had a stronger relationship with the target, which hinted that they might be more important for predictions.

Additionally, I used the `dabl` library to plot a quick overview of the dataset. This visualization gave me a deeper understanding of the data's distribution and helped spot potential outliers.

### 3: Scaling the Data

Before jumping into KNN regression, I scaled the features using `StandardScaler`. Scaling is important because KNN works based on distances, and features with larger values could dominate the model if not scaled.

### 4: KNN Regression with Different k Values

Next, I split the data into training and testing sets. The idea was to train the model on one portion of the data and then test how well it performs on unseen data.

I tried out different values of k (the number of neighbors considered by KNN) to see how it affects the **Root Mean Squared Error (RMSE)**. RMSE tells us how far off the model's predictions are from the true values, with lower numbers being better.

For example, for k = 2 , the RMSE was 54.88, while for k = 7, the RMSE was 53.43. I plotted RMSE vs. k and found that the performance varied, with the best value of k minimizing the RMSE.

### 5: Feature Reduction

I also wanted to see how the number of features affects the model's performance. I started with all features (like age, BMI, etc.) and gradually removed them one by one, training the model each time. For every reduced feature set, I measured the RMSE and plotted the results.

Interestingly, even with fewer features, the model's RMSE didn't skyrocket, suggesting that some features may not be that important. This also aligns with what I saw in the correlation matrix.

### 6: Conclusion

This exercise helped me understand how KNN regression works and how different factors, like k and the number of features, impact the model's accuracy. Visualizing the results gave me insights into the data and the model's behavior, which is super helpful as a beginner.

# Report: Classifying Newsgroups Using Multinomial Naive Bayes

In this assignment, I worked on classifying documents from the 20 Newsgroups dataset using the Multinomial Naive Bayes (MNB) algorithm. The dataset contains news articles categorized into 20 different topics, but I selected 5 categories to keep things manageable. The categories I picked were:

- alt.atheism
- comp.graphics
- sci.med
- rec.autos
- talk.politics.guns

**1: Preprocessing the Data**

To start, I loaded the dataset using `fetch_20newsgroups`, which is a built-in function from Scikit-Learn. The text data was converted into lowercase, and I also removed common words (called "stopwords") that don't add much meaning to the text, like "the" or "and." I used `TfidfVectorizer` for this, which not only helps with the preprocessing but also converts the text into a format that the model can understand. It transforms the text into something called "Tf-Idf," which basically gives weight to words based on how frequently they appear in the text but also how unique they are across documents.

**2: Building the Model**

Next, I built the Naive Bayes model. For this, I used `MultinomialNB` from Scikit-Learn. I set the smoothing parameter (`alpha`) to 0.01, which helps the model deal with rare words better. I then split the data into a training set (80% of the data) and a test set (20%). The training data is used to teach the model, and the test data helps us see how well it performs.

**3: Model Performance**

After training the model, I evaluated it on the test data. To do this, I looked at several metrics, including accuracy, precision, recall, and F1-score. The accuracy tells us how many documents were correctly classified, while precision and recall

give us more insights into how well the model is performing for each category. I also generated a confusion matrix, which helped me understand where the model made mistakes.

Overall, the model performed well, with a decent accuracy score. One challenge was ensuring the dataset was balanced so that each category had a similar number of documents. This is important because an imbalance could lead to the model being biased toward the category with more documents.

**4: Conclusion**

Using Multinomial Naive Bayes with Tf-Idf worked well for this text classification task. It's a relatively simple yet effective method for handling text data, especially when dealing with a large number of documents and categories. I found this approach easy to implement, and it gave me good results without too much complexity.