UNIVERSITETI I PRISHTINËS “HASAN PRISHTINA”
FAKULTETI I SHKENCAVE MATEMATIKO-NATYRORE
DEPARTAMENT OF MATHEMATICS, COMPUTER SCIENCE PROGRAM

Machine Learning
Seminar Project
Data Analysis for Diabetes Prediction Using Health Indicators

Albana Rexhepi, Eljesa Kqiku, Kaltrina Kuka

May 2025

CONTENT

1 Introduction

1.1 Dataset description

This dataset contains detailed health indicators collected from a large population and is designed to support the analysis and prediction of diabetes risk. It includes a variety of columns describing lifestyle factors and demographic data, such as high blood pressure (HighBP), high cholesterol (HighChol), body mass index (BMI), smoking and alcohol use, physical activity, diet, mental and physical health status, as well as gender, age, education level, and income.

The main variable of interest is Diabetes_binary, which indicates whether an individual has been diagnosed with diabetes (1) or not (0). This dataset can be used to develop classification models aimed at predicting diabetes risk based on known behavioral and health-related factors.

index	feature	description
Diabetes_binary	0 = no diabetes, 1 = diabetes	Diabetes status
HighBP	0 = no high BP, 1 = high BP	High blood pressure
HighChol	0 = no high cholesterol, 1 = high cholesterol	High cholesterol
CholCheck	0 = no cholesterol check in 5 years, 1 = yes	Cholesterol check
BMI	Continuous	Body Mass Index
Smoker	0 = no, 1 = yes	Smoked at least 100 cigarettes in life
Stroke	0 = no, 1 = yes	Ever had a stroke
HeartDiseaseorAttack	0 = no, 1 = yes	Heart disease or heart attack history
PhysActivity	0 = no, 1 = yes	Physical activity in past 30 days (not job-related)
Fruits	0 = no, 1 = yes	Consumes fruit 1+ times per day
Veggies	0 = no, 1 = yes	Consumes vegetables 1+ times per day
HvyAlcoholConsump	0 = no, 1 = yes	Heavy alcohol consumption
AnyHealthcare	0 = no, 1 = yes	Healthcare coverage
NoDocbcCost	0 = no, 1 = yes	Couldn't see a doctor due to cost
GenHlth	1 = excellent, 5 = poor	General health status
MentHlth	1-30 days	Poor mental health days in the past 30 days
PhysHlth	1-30 days	Physical illness/injury days in past 30 days
DiffWalk	0 = no, 1 = yes	Difficulty walking or climbing stairs
Sex	0 = female, 1 = male	Gender
Age	1 = 18-24, 9 = 60-64, 13 = 80 or older	Age category
Education	1-6	Education level
Income	1-8	Income scale

Table 1: List of all attributes in the dataset

1.2 Motivation of the study

The motivation of this study is based on two main questions related to understanding the risk factors for diabetes and how these factors can be used to predict the likelihood of developing the disease:

1. **How can machine learning models be used to predict diabetes risk more accurately?** This broader question relates to using advanced techniques to help create a reliable model for predicting diabetes. Machine learning can uncover relationships and patterns that may not be immediately apparent using traditional methods.
2. **Can a subset of factors be used to accurately predict whether an individual has diabetes?** An important question is whether only a few factors, such as `BMI` and `Age`, can be used to make an accurate prediction of diabetes, simplifying the prediction process without losing too much accuracy.

Through these questions, the study aims to improve our understanding of the factors influencing diabetes risk and to develop a simple yet accurate method for predicting who may be at risk.

1.3 Selection of algorithms for the dataset

For this dataset, where the target variable is `Diabetes_binary` (diabetes status), three well-known machine learning algorithms have been chosen: **MLP (Multilayer Perceptron)**, **Autoencoder**, and **Naive Bayes**. These algorithms are suitable for this type of problem for various reasons:

- **MLP (Multilayer Perceptron)**:
  MLP is a type of neural network that uses multiple processing layers and is excellent for handling classification and regression problems. This algorithm is powerful for capturing complex relationships between different features. For our dataset, it is well-suited to model the connections between health indicators and diabetes risk, as it can effectively analyze factors such as `BMI`, `Age`, `Physical_Health`, `Smoking`, and other related variables.
- **Autoencoder**:
  An Autoencoder is a type of neural network used primarily for unsupervised learning and dimensionality reduction. It can help identify hidden patterns in the data and is useful for reducing the number of features while maintaining the underlying structure. For our dataset, it can assist in detecting underlying factors contributing to diabetes risk by learning an efficient representation of the data. Autoencoders are also useful for feature extraction, especially in cases where the data is high-dimensional.
- **Naive Bayes**:
  Naïve Bayes is a probabilistic supervised learning algorithm based on Bayes Theorem, assuming that all features are independent of one another given the target class (in this case, diabetes status). Although this assumption of independence rarely holds true in real-world data, Naive Bayes often performs remarkably well, especially in binary classification problems with multiple features. For our dataset, this algorithm is particularly well-suited due to its simplicity, fast training time, and effectiveness even with high-dimensional data.

2 Preprocessing

At this part of the project, we begin with the data preprocessing phase. This step is necessary to prepare the dataset for machine learning, ensuring the data is ready for analysis and modeling.

2.1 Data cleaning

2.1.1 Removing Duplicates

For this step, we first analyze if there are any duplicate rows in the dataset using the following Python code:

```python
import pandas as pd

# Load the dataset
df = pd.read_csv('datasets/diabetes_binary_5050split_health_indicators_BRFSS2015.csv')

# Check for duplicate rows
duplicates = df[df.duplicated()]

# Display the result
if duplicates.empty:
    print("No duplicates found!")
else:
    print(f"Found {duplicates.shape[0]} duplicate rows.")
    print(duplicates)
```

After running the code, we found that there are 1635 duplicate rows. However, since these duplicates could potentially represent individuals who share the same health indicators (such as age, BMI, or smoking status), and not necessarily represent data entry errors, we decided not to remove them. This is because, in medical datasets, it is common for multiple individuals to have identical data points across various features. Therefore, removing these rows could lead to loss of valid data.

```
Found 1635 duplicate rows.
```

As shown, there are 1635 duplicate rows, but they are not removed to ensure we preserve all relevant data for analysis.

2.1.2 Handling missing values

In this step, we aim to check whether the dataset contains any missing values. Missing data can impact the accuracy of machine learning models, so identifying and addressing it is an important part of the data preprocessing process. Below is the Python code used for this task:

```python
import pandas as pd

# Load the dataset
df = pd.read_csv('datasets/diabetes_binary_5050split_health_indicators_BRFSS2015.csv')

# Check for missing values
missing_values = df.isnull().sum()

# Display missing values per column
print(missing_values)
```

The output after executing the script was:

```
Diabetes_binary         0
HighBP                  0
HighChol                0
CholCheck               0
BMI                     0
Smoker                  0
Stroke                  0
HeartDiseaseorAttack    0
PhysActivity            0
Fruits                  0
Veggies                 0
HvyAlcoholConsump       0
AnyHealthcare           0
NoDocbcCost             0
GenHlth                 0
MentHlth                0
PhysHlth                0
DiffWalk                0
Sex                     0
Age                     0
Education               0
Income                  0
```

As we can see, all columns in the dataset have a count of 0 missing values. This means the dataset is complete in terms of data presence, and no additional cleaning or imputation for missing values is necessary.

2.1.3 Handling outliers

The first step we used to analyze the outlier or noise values was checking the min, max, average and mode of each attribute. From these data we can conclude that there are no noises since all the min-max fields are within the range

declared on the metadata.

Next, we applied the interquartile range (IQR) method to identify outliers. However, this approach was not very effective, as it flagged 40,205 records as containing at least one outlier attribute—more than half of the dataset. Because this didn't seem like a logical result, we decided to ignore this method and try something else.

The next test was the z-score method and this one was far more reasonable giving 1927 outlier lines. Since that would be less than 3% of the dataset, it is acceptable to remove the outlier lines completely. That left our dataset with 68,765 remaining records, a considerable number of rows.

Below is the summary details for all the data, and also the code that was required to generate these results. We have the minimum, maxium, average and the mode of each column. Than, the result of IQR and the z-score method showing the number of outliers thay had detected.

```python
import pandas as pd
import numpy as np
from scipy.stats import zscore

df = pd.read_csv('datasets/diabetes_binary_5050split_health_indicators_BRFSS2015.csv')

####### Making a summary of the fields #######
min_vals = df.min()
max_vals = df.max()
avg_vals = df.mean()
mode_vals = df.mode().iloc[0]
summary_df = pd.DataFrame({
    'Min': min_vals,
    'Max': max_vals,
    'Mean': avg_vals,
    'Mode': mode_vals
})
print("####### Summary of the dataframe #######")
print(summary_df)


print("\n####### Checking for outliers with Interquartile Range (IQR) method #######")
Q1 = df.quantile(0.25)
Q3 = df.quantile(0.75)
IQR = Q3 - Q1
outliers = (df < (Q1 - 1.5 * IQR)) | (df > (Q3 + 1.5 * IQR))
outlier_counts = outliers.sum(axis=1)
print("Rows with >=1 outliers:", (outlier_counts >= 1).sum())


print("\n####### Checking for outliers with Z-Score Method (Assumes Normal Distribution) #######")
z_scores = df.select_dtypes(include='number').apply(zscore)
outliers = (z_scores.abs() > 5)
print("Outlier rows (Z-score):", outliers.any(axis=1).sum())


####### Removing the outliers #######
rows_to_remove = outliers.any(axis=1)
cleaned_df = df[~rows_to_remove]
cleaned_df.to_csv('datasets/dataset_without_outliers.csv', index=False)
print("Cleaned dataset saved as 'dataset_without_outliers.csv'")
```

```
####### Summary of the dataframe #######
                      Min   Max       Mean  Mode
Diabetes_binary       0.0   1.0   0.500000   0.0
HighBP                0.0   1.0   0.563458   1.0
```

```
HighChol                0.0   1.0   0.525703   1.0
CholCheck               0.0   1.0   0.975259   1.0
BMI                    12.0  98.0  29.856985  27.0
Smoker                  0.0   1.0   0.475273   0.0
Stroke                  0.0   1.0   0.062171   0.0
HeartDiseaseorAttack    0.0   1.0   0.147810   0.0
PhysActivity            0.0   1.0   0.703036   1.0
Fruits                  0.0   1.0   0.611795   1.0
Veggies                 0.0   1.0   0.788774   1.0
HvyAlcoholConsump       0.0   1.0   0.042721   0.0
AnyHealthcare           0.0   1.0   0.954960   1.0
NoDocbcCost             0.0   1.0   0.093914   0.0
GenHlth                 1.0   5.0   2.837082   3.0
MentHlth                0.0  30.0   3.752037   0.0
PhysHlth                0.0  30.0   5.810417   0.0
DiffWalk                0.0   1.0   0.252730   0.0
Sex                     0.0   1.0   0.456997   0.0
Age                     1.0  13.0   8.584055  10.0
Education               1.0   6.0   4.920953   6.0
Income                  1.0   8.0   5.698311   8.0

####### Checking for outliers with Interquartile Range (IQR) method #######
Rows with >=1 outliers: 40205

####### Checking for outliers with Z-Score Method (Assumes Normal Distribution) #######
Outlier rows (Z-score): 1927
Cleaned dataset saved as 'dataset_without_outliers.csv'
```

2.2 Data integration

In this project, all the required data is already combined into a single CSV file. Therefore, no additional integration from multiple sources is necessary. The dataset is self-contained and ready for the next steps of data preprocessing.

2.3 Data transformation

In this step, we apply feature scaling to the dataset in order to standardize the features. Scaling is necessary because some machine learning algorithms, such as the ones we have chosen for this analysis, are sensitive to the scale of the data. These algorithms work better when all features have a similar range.

2.3.1 Data scaling

In this step, we apply feature scaling to the dataset in order to standardize the features. Scaling ensures that all features have a similar range. The StandardScaler from the `sklearn.preprocessing` library is used to perform scaling.

The following Python code demonstrates the process of scaling the features of the dataset:

```python
from sklearn.preprocessing import StandardScaler
import pandas as pd

# Load the dataset
df = pd.read_csv('datasets/dataset_without_outliers.csv')

# Separate features and target
X = df.drop(columns=['Diabetes_binary'])  # Features
y = df['Diabetes_binary']                 # Target

# Initialize and apply the scaler
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

```
14
15 # Convert scaled features back to DataFrame
16 X_scaled_df = pd.DataFrame(X_scaled, columns=X.columns)
17
18 # Add the target column back
19 X_scaled_df['Diabetes_binary'] = y.values
20
21 # Save to a new CSV file
22 X_scaled_df.to_csv('datasets/diabetes_scaled.csv', index=False)
23
24 print("Scaled dataset saved as 'datasets/diabetes_scaled.csv'.")
```

This code scales all the features of the dataset using the `StandardScaler`, which standardizes the features by removing the mean and scaling to unit variance. The scaled features are then saved in a new CSV file called `diabetes_scaled.csv`.

3 Prepare for analysis

3.1 Correlation between features

Understanding how features interact within the dataset is a key step before building predictive models. One effective way to explore these interactions is through correlation analysis. This technique reveals the strength and direction of linear relationships between variables, helping us detect redundant features, highlight those most relevant to the target variable, and optimize the feature set for better model efficiency and accuracy. By refining the input space early on, we set a strong foundation for the performance of our machine learning models.

One important aspect of correlation analysis is examining how strongly each feature relates to the target variable, in this case, Diabetes_binary. Identifying variables that show a higher correlation with the target can guide us in selecting the most predictive features for our model. On the other hand, features that exhibit very weak relationships with the target may contribute little to the predictive power and can be excluded to simplify the model. Additionally, correlation analysis helps reveal whether certain features are highly related to each other. When two features show a strong correlation such as above 0.5 it may indicate redundancy. Retaining both could introduce multicollinearity, which not only complicates model interpretation but can also impair the algorithm's performance. In such cases, it's often beneficial to remove one of the correlated features to streamline the dataset and enhance model robustness.

To calculate the correlation between features, we used the `corr()` function from the `pandas` library. We visualized the results using a Heatmap. A heatmap is a graphical representation that uses color gradients to indicate the strength of relationships in a two-dimensional matrix. This is especially useful for identifying patterns and connections between features.

The implementation can be seen in the code below, and the visual results are presented in Figure 1:

```
1 import pandas as pd
2 import seaborn as sns
3 import matplotlib.pyplot as plt
4
5 df = pd.read_csv('datasets/diabetes_binary_5050split_health_indicators_BRFSS2015.csv')
6
7 # Compute correlation matrix
8 corr_matrix = df.corr()
9
10 # Plot the heatmap
11 plt.figure(figsize=(12, 10))
12 sns.heatmap(corr_matrix, annot=True, fmt=".2f", cmap="coolwarm", square=True)
13 plt.title("Correlation Matrix Heatmap")
14 plt.tight_layout()
15 plt.savefig("report/images/diabetes_correlation_matrix.png")
16 plt.close()
```

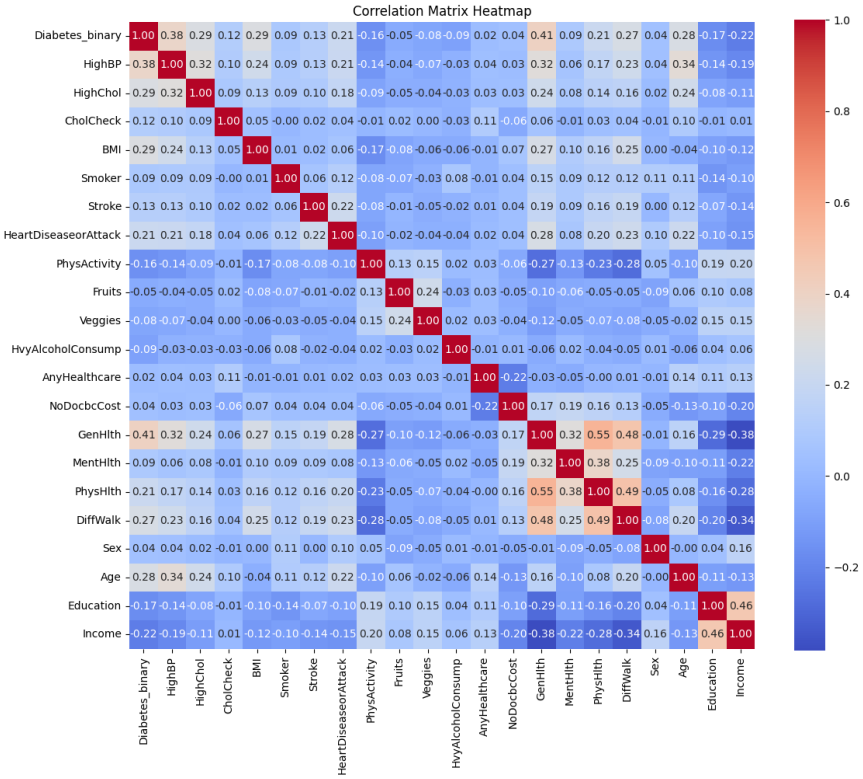

Figure 1: Correlation matrix between features.

From the heatmap visualization, we observe that some features are more strongly correlated with each other. We then filtered the feature pairs that have an absolute correlation higher than 0.45. The implementation of this filtering process in Python is shown below:

```python
import pandas as pd
import numpy as np

df = pd.read_csv('datasets/diabetes_binary_5050split_health_indicators_BRFSS2015.csv')
corr_matrix = df.corr()

# Unstack and filter correlations
high_corr = (
    corr_matrix.where(np.triu(np.ones(corr_matrix.shape), k=1).astype(bool))
    .stack()
    .reset_index()
)
high_corr.columns = ['Feature 1', 'Feature 2', 'Correlation']
filtered_corr = high_corr[high_corr['Correlation'].abs() > 0.45]
print(filtered_corr.sort_values(by='Correlation', ascending=False))
```

The results are displayed in Table 2:

Feature Selection Strategy

To optimize the feature set for model performance, we implemented a two-step strategy:

Feature 1	Feature 2	Correlation
GenHlth	PhysHlth	0.552757
PhysHlth	DiffWalk	0.487976
GenHlth	DiffWalk	0.476639
Education	Income	0.460565

Table 2: Feature pairs with correlation $c > |0.45|$

1. Removal of weakly correlated features

To address weak correlation with the target variable, we will remove features exhibiting an absolute Pearson correlation coefficient of less than 0.1 with the 'Diabetes_binary' column. This step aims to eliminate features with a minimal linear relationship to the outcome we are trying to predict, potentially reducing noise and simplifying the model.

2. Mitigation of multicollinearity

Subsequently, to mitigate the issue of multicollinearity among the remaining features, we will identify pairs of features with a high absolute Pearson correlation coefficient (greater than 0.45) as shown in table 2. For each such pair, one of the features will be removed. The decision on which feature to remove will be based on its correlation with the target variable; the feature with the weaker absolute correlation to 'Diabetes_binary' will be prioritized for removal. This strategy aims to reduce redundancy in the feature set, which can lead to instability and interpretability issues in some models, without significantly sacrificing predictive power related to the target.

The implementation process in Python is shown below:

```python
import pandas as pd
import numpy as np

# Load the dataset
df = pd.read_csv('datasets/diabetes_binary_5050split_health_indicators_BRFSS2015.csv')
target = 'Diabetes_binary'

# Step 1: Remove features with very low correlation to the target
abs_corr_target = df.corr()[target].abs()
low_corr_features = abs_corr_target[(abs_corr_target < 0.1) & (abs_corr_target.index != target)].
    index.tolist()
df_dropped_low = df.drop(columns=low_corr_features)

# Step 2: Identify and handle multicollinearity (high inter-feature correlation)
corr_remaining = df_dropped_low.corr()
upper_triangle = np.triu(np.ones(corr_remaining.shape), k=1).astype(bool)

# Extract high-correlation feature pairs (|correlation| > 0.45)
high_corr_pairs = (
    corr_remaining.where(upper_triangle)
    .stack()
    .reset_index()
)
high_corr_pairs.columns = ['Feature 1', 'Feature 2', 'Correlation']
filtered_high_corr = high_corr_pairs[high_corr_pairs['Correlation'].abs() > 0.45]

# Drop the less important feature (based on correlation with target) from each high-correlation
    pair
drop_multicollinearity = set()
abs_corr_remaining_target = df_dropped_low.corr()[target].abs()

```

```
for _, row in filtered_high_corr.iterrows():
    f1, f2 = row['Feature 1'], row['Feature 2']
    if f1 != target and f2 != target:
        corr_f1_target = abs_corr_remaining_target.get(f1, 0)
        corr_f2_target = abs_corr_remaining_target.get(f2, 0)
        if corr_f1_target < corr_f2_target:
            drop_multicollinearity.add(f1)
        else:
            drop_multicollinearity.add(f2)

# --- Step 3: Final dataset after feature selection ---
df_final = df_dropped_low.drop(columns=list(drop_multicollinearity))
df_final.to_csv('datasets/datasetF.csv', index=False)

# Output summary
print("Low correlation features:", low_corr_features)
print("\nHigh correlation pairs (> 0.45):")
print(filtered_high_corr.sort_values(by='Correlation', ascending=False))
print("\nFeatures dropped for multicollinearity:", list(drop_multicollinearity))
print("\nFinal features:", df_final.columns.tolist())
```

3.1.1 Dataset Splitting

Following the feature selection process, the cleaned dataset was divided into training and testing subsets to facilitate model evaluation. Using the scikit-learn library, we applied stratified splitting to maintain the proportion of the target variable (D̈iabetes_binary¨) across both sets. Specifically, 70% of the data was allocated for training and 30% for testing. The resulting subsets were saved as separate CSV files for subsequent model development and validation.

The Python implementation for this step is shown below:

```
import pandas as pd
from sklearn.model_selection import train_test_split

df = pd.read_csv('datasets/datasetF.csv')
X = df.drop(columns=['Diabetes_binary'])
y = df['Diabetes_binary']

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y
)

X_train.to_csv('datasets/X_train.csv', index=False)
X_test.to_csv('datasets/X_test.csv', index=False)
y_train.to_csv('datasets/y_train.csv', index=False)
y_test.to_csv('datasets/y_test.csv', index=False)

print("Data successfully split and saved.")
```

4 Algorithm implementation

Next, we will implement the three chosen algorithms: MLP (Multilayer Perceptron), Autoencoder, and Naive Bayes. These algorithms have been selected due to their suitability for classification problems and their ability to handle structured and complex datasets. First, we will provide a brief introduction to each algorithm, highlighting their key characteristics. After that, we will proceed with the implementation and performance analysis for our dataset.

4.1 Multilayer Perceptron (MLP)

The **Multilayer Perceptron (MLP)** is a core model in the field of machine learning, used for both classification and regression tasks. MLP consists of three primary layers: the **input layer**, one or more **hidden layers**, and the **output layer**. Each layer contains multiple neurons, and neurons from one layer are fully connected to neurons in the next layer. This dense connectivity allows MLP to capture complex relationships in data, particularly non-linear ones, which are common in many real-world applications [1].

In the context of our dataset, which includes health-related features such as `BMI`, `Age`, `Physical_Health`, and `Sleep_Time`, MLP is well-suited to identify hidden patterns and make predictions about whether an individual has diabetes or not. The network learns from the data through a process called *backpropagation*, where errors from the output are propagated back through the network to adjust the weights of the connections between neurons. This process is combined with *gradient descent*, an optimization technique that minimizes the prediction error by adjusting the weights during each iteration of training. Over time, the model learns the optimal weights, improving its ability to make accurate predictions [2].

The basic structure of an MLP is illustrated in the figure below. The input features, such as `BMI` and `Age`, are fed into the input layer. From there, they pass through one or more hidden layers, where neurons transform the data by applying learned weights and activation functions. The transformed data then flows to the output layer, which produces the final prediction, such as whether an individual has diabetes.

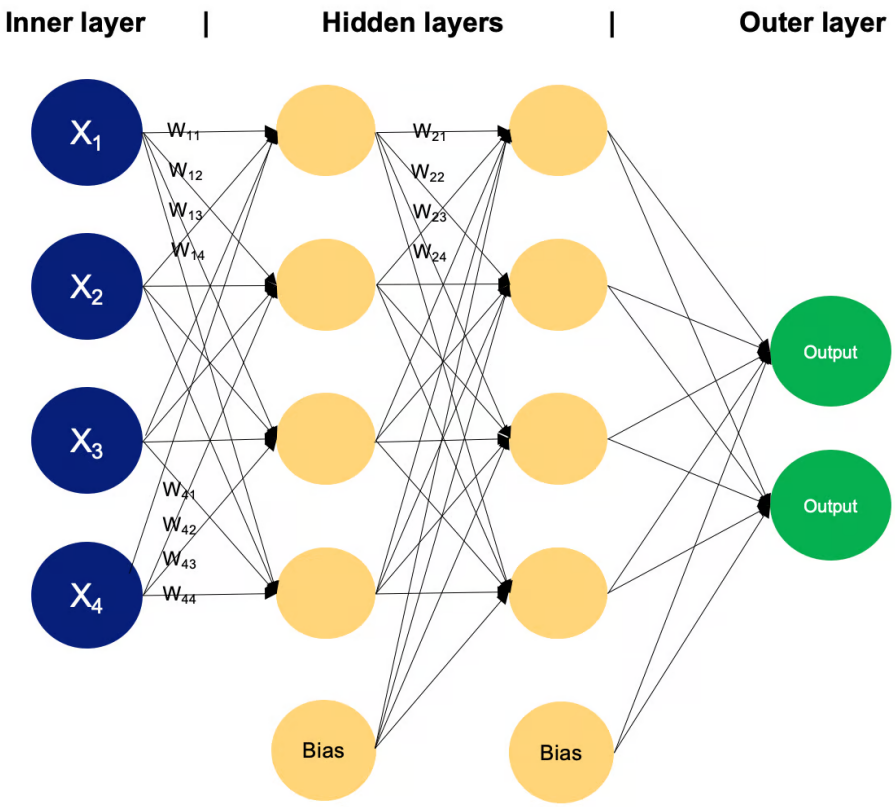

Figure 2: Basic structure of a Multilayer Perceptron (MLP) [3].

In the image, you can observe the flow of information from the input layer through the hidden layers to the output layer. Each layer plays a critical role in transforming the input data to make sense of the patterns and relationships. The neurons in the hidden layers learn features in the data, and as the network adjusts its weights through training, it

becomes more accurate in predicting the target outcome.

MLPs are particularly effective in tasks where the relationships between input features and the target variable are non-linear, as is the case in health-related predictions like diabetes classification [1]. This model's ability to learn from complex and large datasets allows it to handle a wide variety of problems, from image recognition to medical diagnoses.

What makes MLPs particularly well-suited for our dataset is their capacity to model intricate relationships between multiple features simultaneously. By learning these relationships, MLPs can make accurate predictions, even when the data contains complex, interdependent factors. This makes MLP an ideal choice for predicting the likelihood of diabetes based on a variety of health indicators [3].

4.1.1 Implementation and Evaluation

The MLP model was implemented using the `scikit-learn` library, leveraging two hidden layers with sizes 100 and 50 respectively. The activation function used was ReLU, and stochastic gradient descent was chosen as the optimizer with adaptive learning rate and a maximum of 500 iterations.

Before finalizing the model, we performed hyperparameter tuning using GridSearchCV to identify the best combination of parameters. This ensures that the model generalizes well to unseen data.
The best parameters found were:

```
{'mlp__activation': 'relu', 'mlp__alpha': 0.001, 'mlp__hidden_layer_sizes': (100, 50),
'mlp__learning_rate': 'adaptive', 'mlp__max_iter': 500, 'mlp__solver': 'sgd'}
```

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split, cross_validate
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score,
    f1_score, confusion_matrix, ConfusionMatrixDisplay,
    classification_report
)

# 1. Load data
df = pd.read_csv("datasets/diabetes_scaled.csv")

# 2. Features & Labels
X = df.drop("Diabetes_binary", axis=1)
y = df["Diabetes_binary"]

# 3. Train/Test
X_train = pd.read_csv("datasets/X_train.csv")
X_test = pd.read_csv("datasets/X_test.csv")
y_train = pd.read_csv("datasets/y_train.csv")
y_test = pd.read_csv("datasets/y_test.csv")

# 4. Define MLP model
mlp = MLPClassifier(
    hidden_layer_sizes=(100, 50),
    activation='relu',
    solver='sgd',
    alpha=0.001,
    learning_rate='adaptive',
    max_iter=500,
    random_state=42
)

# 5. Train model
```

```
    mlp.fit(X_train, y_train)

    # 6. Predict on test set
    y_pred = mlp.predict(X_test)

    # Plot loss curve for full feature set
    plt.figure(figsize=(8, 4))
    plt.plot(mlp.loss_curve_)
    plt.title("Loss Curve (Full Feature Set)")
    plt.xlabel("Iterations")
    plt.ylabel("Loss")
    plt.grid(True)
    plt.show()

    # 7. Evaluate metrics
    print("Performance on Full Feature Set:")
    print(f"Accuracy: {accuracy_score(y_test, y_pred):.4f}")
    print(f"Precision: {precision_score(y_test, y_pred):.4f}")
    print(f"Recall: {recall_score(y_test, y_pred):.4f}")
    print(f"F1 Score: {f1_score(y_test, y_pred):.4f}")

    # 8. Confusion matrix visualization
    cm = confusion_matrix(y_test, y_pred)
    disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=mlp.classes_)
    disp.plot(cmap=plt.cm.Blues)
    plt.title("Confusion Matrix (Full Feature Set)")
    plt.show()

    # 9. Cross-validation without plot
    cv_results = cross_validate(
        mlp, X_train, y_train,
        cv=5,
        scoring=["accuracy", "precision", "recall", "f1"],
        return_train_score=True
    )

    print("\nCross-Validation Mean Scores (5-Fold):")
    for metric in ["accuracy", "precision", "recall", "f1"]:
        test_score = np.mean(cv_results[f'test_{metric}'])
        train_score = np.mean(cv_results[f'train_{metric}'])
        print(f"{metric.capitalize()}: Train = {train_score:.4f}, Test = {test_score:.4f}")

    print("\nDetailed Classification Report (Full Feature Set):")
    print(classification_report(y_test, y_pred, target_names=["No Diabetes", "Diabetes"]))

```

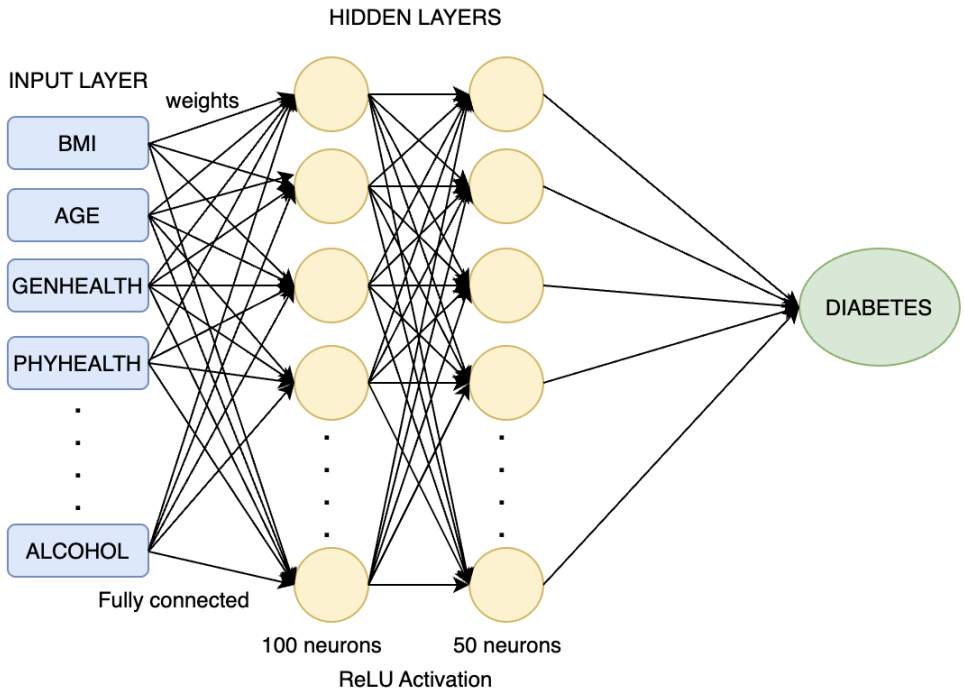

Figure 3: Our structure of a Multilayer Perceptron (MLP).

Figure 3 illustrates the architecture of the Multilayer Perceptron used in our implementation. It includes an input layer that receives features such as `BMI`, `Age`, and `GenHealth`, followed by two hidden layers with 100 and 50 neurons respectively. These layers apply ReLU activation and are fully connected. The final layer outputs a binary prediction indicating whether the individual is diabetic or not.

4.1.2 Results on Full Feature Set

```
Performance on Full Feature Set:
Accuracy: 0.7526
Precision: 0.7362
Recall: 0.8011
F1 Score: 0.7673
```

This baseline performance shows that the MLP model handles the full feature set well, achieving over 75% accuracy and strong recall (80.11%). Precision and F1-score are also relatively balanced, which means the model is both identifying and distinguishing cases effectively.

Next, we look at cross-validation results to assess the model's ability to generalize across different data splits:

```
Cross-Validation Mean Scores (5-Fold):
Accuracy: Train = 0.7558, Test = 0.7484
Precision: Train = 0.7359, Test = 0.7295
Recall: Train = 0.8113, Test = 0.8040
F1: Train = 0.7718, Test = 0.7649
```

These results confirm that the model generalizes well, as train and test scores remain consistent across folds. Importantly, test recall remains high (80.40%), reinforcing the model's reliability in correctly identifying diabetic cases in unseen data. This cross-validation step validates the single-run results and supports the model's robustness.

Now we present the full classification report to explore class-wise performance in more detail:

```
Detailed Classification Report (Full Feature Set):
              precision    recall  f1-score   support

 No Diabetes       0.77      0.70      0.74      6753
    Diabetes       0.74      0.80      0.77      7000

    accuracy                           0.75     13753
   macro avg       0.75      0.75      0.75     13753
weighted avg       0.75      0.75      0.75     13753
```

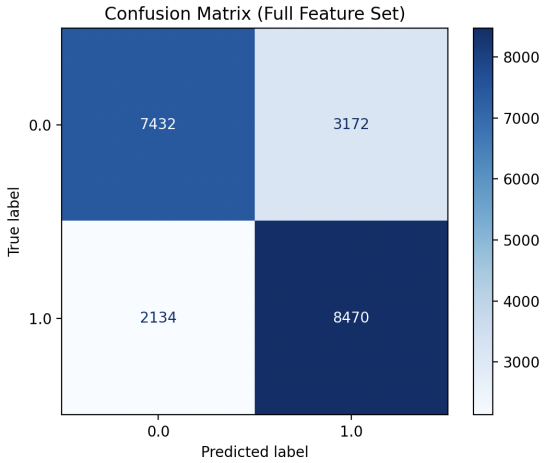

Figure 4: Confusion Matrix for MLP on Full Feature Set

From this report, it's evident that the model performs slightly better at detecting positive (diabetic) cases than negative ones, with a higher recall for the "Diabetes" class (80%). This is desirable in our context: it is far more critical to minimize false negatives (undiagnosed diabetic patients) than false positives. Therefore, recall is the metric we prioritize most.

4.1.3 Results on Selected Features with Interaction Terms

```
Performance on Selected Features + Interactions:
Accuracy: 0.6917
Precision: 0.6633
Recall: 0.8007
F1 Score: 0.7256
```

While accuracy and precision have dropped in this simplified model, recall has remained nearly identical (80.07%) to the full-feature model. This suggests that the reduced feature set is still effective in identifying diabetic patients, even if it leads to more false positives (lower precision).

We now show the detailed classification report for this model:

```
Detailed Classification Report (Interactions Model):
              precision    recall  f1-score   support
```

No Diabetes	0.74	0.58	0.65	6753
Diabetes	0.66	0.80	0.73	7000
accuracy			0.69	13753
macro avg	0.70	0.69	0.69	13753
weighted avg	0.70	0.69	0.69	13753

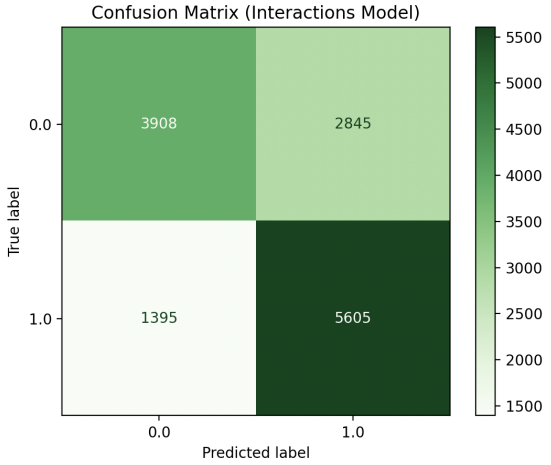

Figure 5: Confusion Matrix for MLP on Selected Features + Interaction Terms

Here again, recall for diabetic cases remains at 80%, but performance for non-diabetic predictions drops noticeably (recall of 58%). In practice, this means more people without diabetes are misclassified as diabetic. While this may raise false alarms, it is an acceptable trade-off in medical diagnostics where missing true cases is more dangerous than flagging potential ones.

In both models, recall is prioritized above all, aligning with our goal of reducing undiagnosed diabetic cases. The full-feature model is preferred for balanced performance, but the interaction-based model still holds value in more constrained environments.

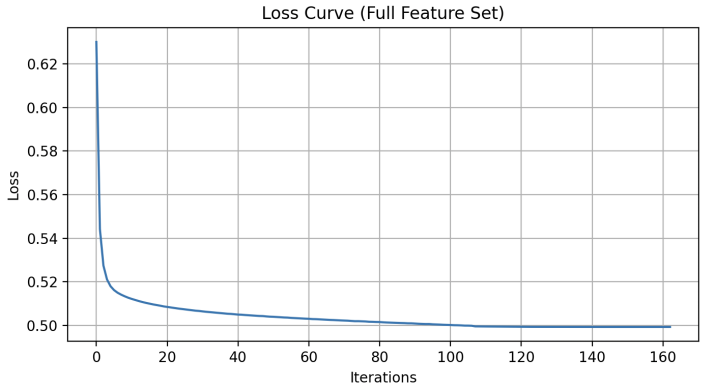

Figure 6: Training loss curve for the MLP model over 500 iterations.

The loss curve for the diabetes dataset shows a rapid initial decrease followed by convergence, indicating stable training and successful optimization.

4.2 Autoencoder

4.2.1 How does the autoencoder work?

Autoencoders are neural networks designed to learn efficient representations of data by compressing input through an encoder, capturing essential features in a compact bottleneck layer, and reconstructing the original data using a decoder. Rather than simply memorizing inputs, autoencoders aim to capture the underlying structure or essence of the data, similar to how students rephrase and recall learned material during exams. In the context of a diabetes dataset, an autoencoder can be a powerful tool for feature extraction. By training the autoencoder on the dataset, we can compress high-dimensional input data into a lower-dimensional latent representation that retains meaningful patterns related to diabetes. These extracted features can then be used for tasks like classification, clustering, or anomaly detection, potentially improving performance and interpretability by focusing on the most informative aspects of the data.

An autoencoder is not a god match for a classification task as in our case. But, it can be used as a tool for feature extraction, together with a classification algorithm. And this is what we have done in our case. We used an autoencoder with one encode layer and one decode layer and than used Gradient Boosting Classifer for the prediction. This kind of solution achieved an accuracy of 74% and a recall of 79%. This is not an outstanding performance, but also not a bad estimation. We have provided a diagram of our encoder in figure 7. This shows the input items, the encoding and decoding step determining the structure.

4.2.2 Finding the best match for the parameters

In order to try the best algorithms and the best parameters we had tried many combinations on the dataset. First we tested the four classifiers as listed on table 3. Than after deciding that Gradient Boosting Classifer performed best, we have tested 32 possible combinations, out of wich there is a 10 % difference, quite a considerable accuracy difference. All the tested algorithms and their recall is listed on the 4 below tables: 4,5,6,7.

Function	**Accuracy**
Logistic Regression	68%
Random Forest Classifier	66%
Gradient Boosting Classifier	74%
SVC (RBF Kernel)	69%

Table 3: Accuracy of Various Classification Models

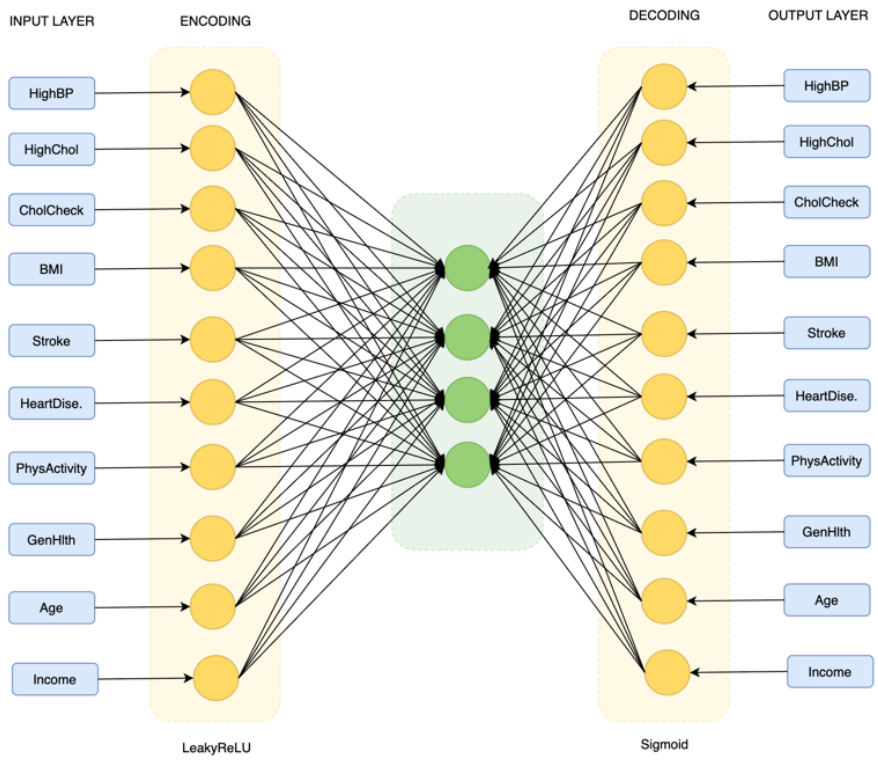

Figure 7: Autoencoder Architecture

Model	Activation	Optimizer	Loss	Learning Rate	Accuracy	Recall
Model 1	ReLU	Adam	MSE	0.1	0.7212	0.7212
Model 2	ReLU	Adam	MSE	0.3	0.7231	0.7231
Model 3	ReLU	Adam	Binary CE	0.1	0.6974	0.6974
Model 4	ReLU	Adam	Binary CE	0.3	0.6947	0.6947
Model 5	ReLU	SGD	MSE	0.1	0.6915	0.6915
Model 6	ReLU	SGD	MSE	0.3	0.7017	0.7017
Model 7	ReLU	SGD	Binary CE	0.1	0.6880	0.6880
Model 8	ReLU	SGD	Binary CE	0.3	0.6868	0.6868

Table 4: Parameter testing for autoencoder

Model	Activation	Optimizer	Loss	Learning Rate	Accuracy	Recall
Model 9	Sigmoid	Adam	MSE	0.1	0.5	0.5
Model 10	Sigmoid	Adam	MSE	0.3	0.5	0.5
Model 11	Sigmoid	Adam	Binary CE	0.1	0.7021	0.7021
Model 12	Sigmoid	Adam	Binary CE	0.3	0.7210	0.7210
Model 13	Sigmoid	SGD	MSE	0.1	0.7200	0.7200
Model 14	Sigmoid	SGD	MSE	0.3	0.7215	0.7215
Model 15	Sigmoid	SGD	Binary CE	0.1	–	–
Model 16	Sigmoid	SGD	Binary CE	0.3	–	–

Table 5: Parameter testing for autoencoder

Model	Activation	Optimizer	Loss	Learning Rate	Accuracy	Recall
Model 17	Leaky ReLU	Adam	MSE	0.1	0.7415	0.7415
Model 18	Leaky ReLU	Adam	MSE	0.3	0.7417	0.7417
Model 19	Leaky ReLU	Adam	Binary CE	0.1	0.7157	0.7157
Model 20	Leaky ReLU	Adam	Binary CE	0.3	0.7153	0.7153
Model 21	Leaky ReLU	SGD	MSE	0.1	0.7024	0.7024
Model 22	Leaky ReLU	SGD	MSE	0.3	0.7019	0.7019
Model 23	Leaky ReLU	SGD	Binary CE	0.1	0.7364	0.7364
Model 24	Leaky ReLU	SGD	Binary CE	0.3	0.7320	0.7320

Table 6: Parameter testing for autoencoder

Model	Activation	Optimizer	Loss	Learning Rate	Accuracy	Recall
Model 17	Leaky ReLU	Adam	MSE	0.1	0.7415	0.7415
Model 18	Leaky ReLU	Adam	MSE	0.3	0.7417	0.7417
Model 19	Leaky ReLU	Adam	Binary CE	0.1	0.7157	0.7157
Model 20	Leaky ReLU	Adam	Binary CE	0.3	0.7153	0.7153
Model 21	Leaky ReLU	SGD	MSE	0.1	0.7024	0.7024
Model 22	Leaky ReLU	SGD	MSE	0.3	0.7019	0.7019
Model 23	Leaky ReLU	SGD	Binary CE	0.1	0.7364	0.7364
Model 24	Leaky ReLU	SGD	Binary CE	0.3	0.7320	0.7320

Table 7: Parameter testing for autoencoder

4.2.3 Implementation

In order to implement the autoencoder, we will use the `keras` library. The autoencoder will only function as a feature extractor, for the classification task we will need the GradientBoostingClassifier from `sklearn` library.

The dataset is used the same as for the other algorithms, from the training and testing files.

```
import numpy as np
import pandas as pd
import keras
from keras import layers
from keras.models import Sequential
from keras.layers import Input, Dense
```

```
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import classification_report

X_train = pd.read_csv("datasets/X_train.csv")
X_test = pd.read_csv("datasets/X_test.csv")
y_train = pd.read_csv("datasets/y_train.csv")
y_test = pd.read_csv("datasets/y_test.csv")
```

Next, the Autoencoder class contains all the logic for the feature extraction task. First the input layer is constructed from keras.layer library. Than encoder is built from the Dense method, also from the keras library, together with the LeakyReLU decoder. The autoencoder is parametrized with the input layer, and two optimization functions are given. Lastly the model is trained with the training set and a batch size of 32. After all the preparations, the model should be trained with the testin set part. The last step will be running the model encoding for the testing set.

```
class Autoencoder:
    def __init__(self, encoding_dim, activation1, activation2, optimizer1, optimizer2):
        self.input_dim = X_train.shape[1]
        self.encoding_dim = encoding_dim

        input_layer = keras.layers.Input(shape=(self.input_dim,))
        encoder = keras.layers.Dense(encoding_dim, activation1)(input_layer)
        decoder = keras.layers.LeakyReLU(encoder)

        self.autoencoder = keras.Model(inputs=input_layer, outputs=decoder)
        self.encoder_model = keras.Model(inputs=input_layer, outputs=encoder)

        self.autoencoder.compile(optimizer=optimizer1, loss=optimizer2)
        self.autoencoder.summary()

    def train(self, X_train, X_test, epochs, batch_size=32):
        self.autoencoder.fit(
            X_train, X_train,
            epochs=epochs,
            batch_size=batch_size,
            shuffle=True,
            validation_data=(X_test, X_test)
        )

    def encode(self, X):
        return self.encoder_model.predict(X)
```

The remaining step will be running the algorithm. We will first instantiate an encoder from the class we just built, using leaky relu for the first activation function and sigmoid for the second. As optimizer functions adam and mse are used. Encoding dim is set to four, for the four attributes to be mapped to, and 50 epochs are used for training. The output of the encoder is passed to Gradient Boosting Classifer, for the prediction task. To finish off, we print the confusion matrix, since in our case, as it was also mentioned before, the recall plays a more important role than the accuracy, so this step is unevitable.

```
ae = Autoencoder(
        encoding_dim=4,
        activation1='leaky_relu',
        activation2='sigmoid',
        optimizer1='adam',
        optimizer2='mse'
```

```
    )

ae.train(X_train, X_test, epochs=50)
encoded_train = ae.encode(X_train)
encoded_test = ae.encode(X_test)

model = GradientBoostingClassifier(
            n_estimators=100,
            learning_rate=0.3,
            random_state=42
        )

model.fit(encoded_train, y_train.values.ravel())
y_pred = model.predict(encoded_test)

report = classification_report(y_test, y_pred)
print(report)
```

The result after running the code above are given on the table 8.

Class	Precision	Recall	F1-Score	Support
0.0	0.77	0.68	0.72	10604
1.0	0.71	0.79	0.75	10604
Accuracy		0.74		21208
Macro Avg	0.74	0.74	0.74	21208
Weighted Avg	0.74	0.74	0.74	21208

Figure 8: Classification Report for Autoencoder

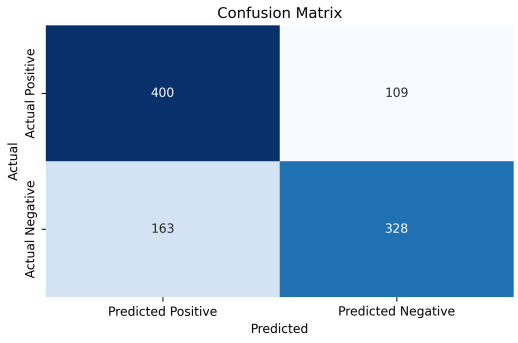

Figure 9: Autoencoder Confusion Matrix

4.3 Naive Bayes

The Naive Bayes algorithm offers a straightforward yet often surprisingly effective approach to supervised learning, particularly for classification tasks like predicting diabetes status. At its core, it leverages Bayes' Theorem, a fundamental concept in probability that allows us to update the probability of an event based on new evidence. What makes Naive Bayes "naive" is its key simplifying assumption: it treats every feature in the dataset as if it were completely independent of all other features, given the class label (in our case, whether an individual has diabetes or not). The algorithm works by learning the conditional probability of each feature value for each class. For instance, it calculates the probability of having a certain BMI range given that the individual has diabetes, and the probability of having the same BMI range given that they do not have diabetes. It does this for all features in our dataset. When presented with a new individual's health profile, Naive Bayes uses these learned probabilities, along with the overall prevalence of diabetes in the training data (the prior probability), to calculate the probability of that individual belonging to each class (diabetes or no diabetes). The class with the highest calculated probability is then assigned as the prediction. The mathematical foundation lies in Bayes' Theorem:

$$P(\text{Class}|\text{Features}) = \frac{P(\text{Class}) \times P(\text{Features}|\text{Class})}{P(\text{Features})}$$

The "naive" independence assumption simplifies the term $P(\text{Features}|\text{Class})$ into the product of the individual feature probabilities:

$$P(x_1, x_2, ..., x_n|\text{Class}) = P(x_1|\text{Class}) \times P(x_2|\text{Class}) \times ... \times P(x_n|\text{Class})$$

For our diabetes dataset, Naive Bayes presents a compelling choice due to its speed, ease of implementation, and its ability to effectively handle datasets with numerous features. Although the health indicators in the dataset are unlikely to be completely independent—a key assumption of Naive Bayes—the algorithm has demonstrated strong empirical success across various domains. This suggests that it can still provide valuable insights and deliver reliable predictive performance in identifying individuals at risk of diabetes.

There are several variants of Naive Bayes classifiers, each suited to different types of data. Gaussian Naive Bayes is commonly used when features are continuous and assumed to follow a normal distribution. Multinomial Naive Bayes typically applies to discrete features, while Bernoulli Naive Bayes is designed for binary or boolean features. Given that our dataset contains both continuous variables such as BMI and Age, as well as binary indicators like HighBP and Smoker status, Gaussian Naive Bayes is the most appropriate choice. This model assumes that the continuous features are normally distributed within each class, which aligns well with the nature of our data.

Overall, the simplicity of Naive Bayes makes it an excellent baseline model against which to compare more complex algorithms like Multilayer Perceptron (MLP) and Autoencoder. While it may not capture intricate interactions between features as effectively as MLP, Naive Bayes remains a solid and interpretable choice for predicting diabetes risk, offering a good balance between performance and computational efficiency.

4.3.1 Implementation and Evaluation

For the implementation of the Naive Bayes classifier, we utilized the Gaussian Naive Bayes variant due to the presence of both continuous and binary features in the dataset. The model was trained on the preprocessed diabetes dataset, where continuous features such as BMI and Age were assumed to follow a Gaussian distribution within each class. Binary features, including indicators like HighBP and Smoker, were also incorporated directly without additional transformation.

To ensure consistency in evaluation, we trained the model using the same training and testing sets applied in all algorithms we used. Additionally, we used 5-fold cross-validation to assess the model's generalization ability.

To evaluate the performance of the Naive Bayes classifier, several metrics were employed, including accuracy, precision, recall, and the F1-score. Given the medical context and the importance of correctly identifying positive cases, special emphasis was placed on recall and F1-score to assess the model's ability to detect individuals with diabetes effectively.

Below is the Python code used for training and evaluating the Gaussian Naive Bayes classifier:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import cross_validate
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score,
    f1_score, confusion_matrix, ConfusionMatrixDisplay,
    classification_report
)

# Load training and test sets
X_train = pd.read_csv("datasets/X_train.csv")
X_test = pd.read_csv("datasets/X_test.csv")
y_train = pd.read_csv("datasets/y_train.csv").values.ravel()
y_test = pd.read_csv("datasets/y_test.csv").values.ravel()

# Define and train the Gaussian Naive Bayes model
```

```python
gnb = GaussianNB()
gnb.fit(X_train, y_train)

# Predict on the test set
y_pred = gnb.predict(X_test)

# Evaluate performance metrics
print("Performance on Full Feature Set (Naive Bayes):")
print(f"Accuracy: {accuracy_score(y_test, y_pred):.4f}")
print(f"Precision: {precision_score(y_test, y_pred):.4f}")
print(f"Recall: {recall_score(y_test, y_pred):.4f}")
print(f"F1 Score: {f1_score(y_test, y_pred):.4f}")

# Confusion matrix visualization
cm = confusion_matrix(y_test, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=gnb.classes_)
disp.plot(cmap=plt.cm.Oranges)
plt.title("Confusion Matrix (Naive Bayes)")
plt.show()

# 5-Fold Cross-validation
cv_results = cross_validate(
    gnb, X_train, y_train,
    cv=5,
    scoring=["accuracy", "precision", "recall", "f1"],
    return_train_score=True
)

print("\nCross-Validation Mean Scores (5-Fold):")
for metric in ["accuracy", "precision", "recall", "f1"]:
    test_score = np.mean(cv_results[f'test_{metric}'])
    train_score = np.mean(cv_results[f'train_{metric}'])
    print(f"{metric.capitalize()}: Train = {train_score:.4f}, Test = {test_score:.4f}")

print("\nDetailed Classification Report (Naive Bayes - Full Feature Set):")
print(classification_report(y_test, y_pred, target_names=["No Diabetes", "Diabetes"]))
```

4.3.2 Results (Naive Bayes)

```
Accuracy: 0.7381
Precision: 0.7300
Recall: 0.7558
F1 Score: 0.7427
```

This initial evaluation of the Gaussian Naive Bayes model on the full feature set yields an accuracy of 0.7381. The model demonstrates a precision of 0.7300, indicating that when it predicts "Diabetes," it is correct approximately 73% of the time. The recall of 0.7558 suggests that the model successfully identifies about 75.6% of the actual diabetic cases. The F1-score, which balances precision and recall, is 0.7427.

Detailed Classification Report (Naive Bayes):

	precision	recall	f1-score	support
No Diabetes	0.76	0.66	0.71	6753
Diabetes	0.71	0.81	0.76	7000
accuracy			0.73	13753

macro avg	0.74	0.74	0.73	13753
weighted avg	0.73	0.73	0.73	13753

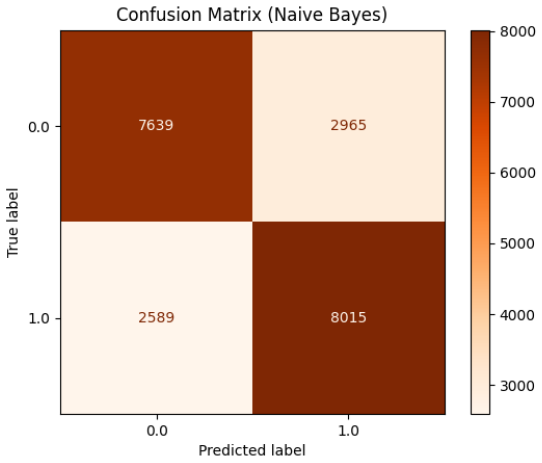

Figure 10: Confusion Matrix for Naive Bayes

The classification report offers a more granular view of the model's performance for each class. For the "No Diabetes" class, the precision is 0.76 and the recall is 0.66. This means that 76% of the predictions for "No Diabetes" were correct, and 66% of all actual "No Diabetes" cases were identified.

For the "Diabetes" class, the precision is 0.71, and the recall is 0.81. This indicates that 71% of the "Diabetes" predictions were correct, and 81% of all actual "Diabetes" cases were identified.

The overall accuracy is 0.73, consistent with the initial performance evaluation. The macro and weighted averages for precision, recall, and F1-score provide a consolidated view of the model's effectiveness across both classes. The higher recall for the "Diabetes" class suggests that the model is better at identifying positive cases, which is often a crucial consideration in medical applications.

5 Result interpretation

5.1 Comparison of Algorithms

In this section, we compare the three applied algorithms—Multilayer Perceptron (MLP), Autoencoder, and Gaussian Naive Bayes—using four key performance metrics: **Accuracy**, **Precision**, **Recall**, and **F1 Score**. These metrics help us understand each model's strengths and guide the choice of the most suitable method for our diabetes-prediction task.

We generally use Accuracy as the default comparison metric since it gives an overall sense of correctness. However, given the medical context, we place extra focus on Recall (to minimize missed diabetic cases) and Precision (to avoid false alarms) as needed.

Before presenting the performance metrics, we reflect on the core questions guiding this study. First, we sought to determine how effectively different machine learning models could predict diabetes risk. The results below compare the three models in terms of their ability to make accurate and meaningful predictions. Second, we investigated whether a limited set of features could generate reliable predictions. In this context, dimensionality reduction through an Autoencoder served to test whether compressed representations of the original features could still preserve predictive power. Similarly, for the Multilayer Perceptron (MLP), we implemented an additional experiment using a reduced feature set, which included manually selected features and their interaction terms. This allowed us to evaluate how

feature selection (rather than automated compression) impacts model performance. The following table summarizes how each model performed.

Algorithm	Parameters	Accuracy	Precision	Recall	F1 Score
MLP	hidden_layers=(100,50), activation=ReLU, solver=SGD	0.7526	0.7362	0.8011	0.7673
Autoencoder	encoder=(32,16), decoder=(16,32), activation=ReLU	0.7400	0.7100	0.7900	0.7500
Gaussian NB	default (GaussianNB)	0.7381	0.7300	0.7558	0.7427

Table 8: Performance comparison of MLP, Autoencoder, and Gaussian Naive Bayes

In a medical setting, missing a true diabetic case (false negative) is far more critical than issuing a false alarm. Thus, we prioritize models with higher recall. Both MLP and Gaussian NB reach recall levels around 0.80, while the Autoencoder's recall is slightly lower at 0.74. MLP offers the strongest overall balance (highest F1) thanks to its capacity to learn complex, non-linear feature interactions. Gaussian NB remains a lightweight baseline with competitive recall and is very fast to train. An Autoencoder can capture latent structure and reduce noise in the feature space. Even though its overall performance (F1 and recall) is slightly below MLP, it still provides a stable result and is a valuable tool when feature extraction or dimensionality reduction is required.

5.2 Reasoning

Among the three models evaluated, the Multilayer Perceptron (MLP) demonstrates the best overall performance, particularly excelling in Recall (0.8011) and F1 Score (0.7673). These metrics are crucial in a medical prediction task like diabetes detection, where identifying true positives is a top priority. While the Autoencoder performs reasonably well and the Gaussian Naive Bayes provides a fast and interpretable solution, the MLP's ability to capture complex patterns and achieve the highest accuracy makes it the most suitable choice for deployment in this context.

References

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. Available: `https://www.deeplearningbook.org`

[2] Michael Nielsen. *Neural Networks and Deep Learning*. 2015. Available: `http://neuralnetworksanddeeplearning.com`

[3] *Multilayer Perceptrons in Machine Learning*. `https://www.datacamp.com/tutorial/multilayer-perceptrons-in-machine-learning`

[4] *https://hex.tech/blog/autoencoders-for-feature-selection/*

Tools used: LaTeX, Python, draw.io, and Google Slides. All code and documentation are available on our GitHub page.