Kevin Monisit, kgm74
Donavon Holgado, dmh313

System for Evaluating Diabetic Risk

Project Definition:

In this project we are trying to explore bodily features(eg.BMI High Cholesterol) and lifestyle features (eg. if a person consumes vegetables regularly or regularly works out) to better understand their relation to diabetes and prediabetes. We will be breaking the project into the following:

Preprocessing
Exploration of the Data
Creating Predictive Models
Create a web page for Healthcare professionals

This project relates to many of the topics covered in this class including, data cleaning and processing, binary classification, predictive models, and database and data management through SQL.

Novelty and Importance:

Novelty:

There have been other projects that have used this dataset and performed data exploration tasks. What will make our project novel is the small web page that we create in HTML, CSS, and JavaScript. We will have a form that a medical professional can use that will instantly give them the probability that the individual is diabetic/prediabetic.

Importance:

This will allow medical professionals to easily input a patient's data into a form and they can instantly easily see how likely they are to be diabetic or prediabetic in an easy to use user-interface. Furthermore, being able to accurately predict important health risks, such as diabetes, as soon as possible using a patient's medical records would be important in mitigating future health issues.

Personal Motivations:

Because we know family members who are prediabetic (including Kevin, who is in our group), we think it will be very interesting to input their data and see how accurate the model is. By knowing which variables correlate with being diabetic/prediabetic via data exploration we can better understand their health and what needs to be focused on in order to lead a healthier life.

Progress and Contribution:

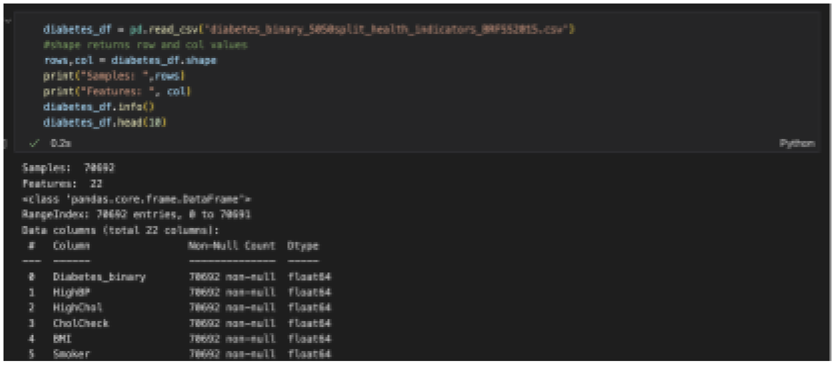
Dataset:

For this project we used a tabular dataset gathered and funded by the Center for Disease Control and Prevention(CDC). This dataset was gathered through The Behavioral Risk Factor Surveillance System (BRFSS), which is a telephone survey collected annually. From this dataset we looked at 70,692 survey responses, that consisted of a 50-50 split of respondents with no diabetes and with either prediabetes or diabetes. In this dataset we had one target variable that was Diabetes_binary which had 2 responses either 0 for no diabetes and 1 for prediabetes or diabetes. In this dataset we also had 21 features including things like BMI, High blood pressure, and smoker that could help gain insight into the relationship between lifestyle and diabetes. This dataset is available for free online at https://archive.ics.uci.edu/dataset/891/cdc+diabetes+health+indicators and from this we were able to download the csv file to be used in our analysis and modeling.

Models/techniques/algorithms used or developed:

Machine Learning/Data Exploration:

When performing data analysis and exploration we used a few different techniques in order to organize the information and produce helpful visuals. Firstly, we read the csv file into a pandas dataframe. This allows for easy manipulation and viewing. After it's in the dataframe we can garner preliminary information such as how many samples and features there are. Additionally we can look at what these features entail and the data types they hold.



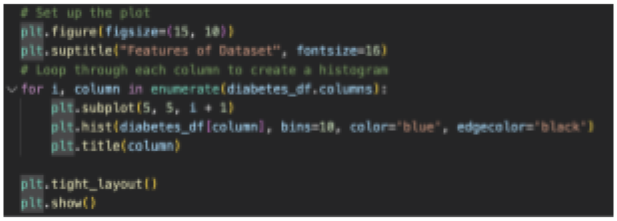
```
diabetes_df = pd.read_csv('diabetes_binary_5050split_health_indicators_BRFSS2015.csv')
#shape returns row and col values
rows,col = diabetes_df.shape
print("Samples: ",rows)
print("Features: ", col)
diabetes_df.info()
diabetes_df.head(10)
```

✓ 0.2s Python

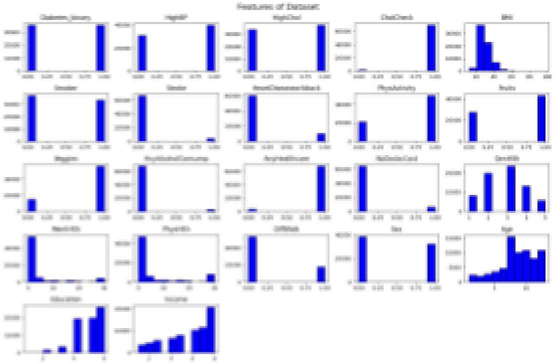
```
Samples:  70692
Features:  22
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 70692 entries, 0 to 70691
Data columns (total 22 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   Diabetes_binary       70692 non-null  float64
 1   HighBP                70692 non-null  float64
 2   HighChol              70692 non-null  float64
 3   CholCheck             70692 non-null  float64
 4   BMI                   70692 non-null  float64
 5   Smoker                70692 non-null  float64
```

Loading into pandas dataframe and getting preliminary info

Next, we can use matplotlib, to get even more information on these features and we did so by producing a plot with a sublot for each feature showing the histogram for each. This gives us insight into distribution and what features may need scaling once we get to building the model.

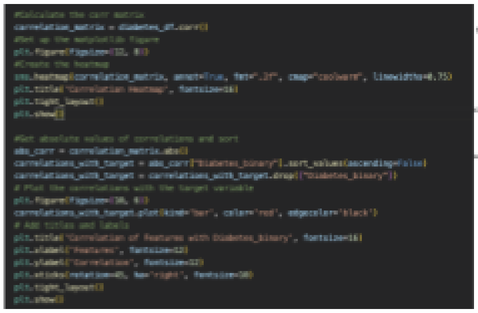


```
# Set up the plot
plt.figure(figsize=(15, 10))
plt.suptitle("Features of Dataset", fontsize=16)
# Loop through each column to create a histogram
for i, column in enumerate(diabetes_df.columns):
    plt.subplot(5, 5, i + 1)
    plt.hist(diabetes_df[column], bins=10, color='blue', edgecolor='black')
    plt.title(column)

plt.tight_layout()
plt.show()
```

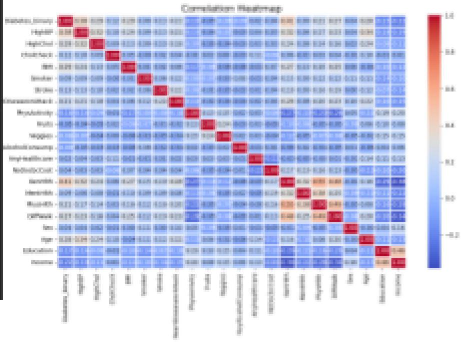


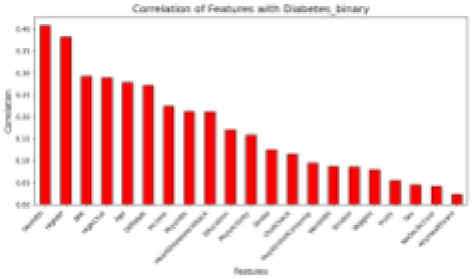
Creating Plot to look at distribution of features

During this analysis we also used seaborn to generate heatmaps. For example when looking at the correlation matrix between features we can use seaborn's heatmap to get a better understanding about how the features correlate to each other as well as the target variable. Again looking at this matrix we organized the information to look at which features are most correlated to the target variable Diabetes_binary and displayed them as a barchart using matplotlib.



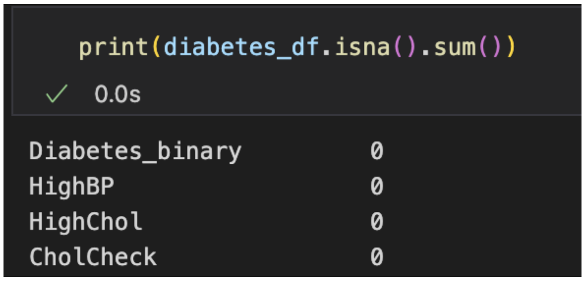
```
#Calculate the corr matrix
correlation_matrix = diabetes_df.corr()
#Set up the matplotlib figure
plt.figure(figsize=(12, 8))
#Create the heatmap
sns.heatmap(correlation_matrix, annot=True, fmt=".2f", cmap="coolwarm", linewidths=0.75)
plt.title("Correlation Heatmap", fontsize=16)
plt.tight_layout()
plt.show()

#Get absolute values of correlations and sort
abs_corr = correlation_matrix.abs()
correlations_with_target = abs_corr["Diabetes_binary"].sort_values(ascending=False)
correlations_with_target = correlations_with_target.drop(["Diabetes_binary"])
# Plot the correlations with the target variable
plt.figure(figsize=(10, 8))
correlations_with_target.plot(kind='bar', color='red', edgecolor='black')
# Add titles and labels
plt.title("Correlation of Features with Diabetes_binary", fontsize=16)
plt.xlabel("Features", fontsize=12)
plt.ylabel("Correlation", fontsize=12)
plt.xticks(rotation=45, ha='right', fontsize=10)
plt.tight_layout()
plt.show()
```





Heatmap and plot generated

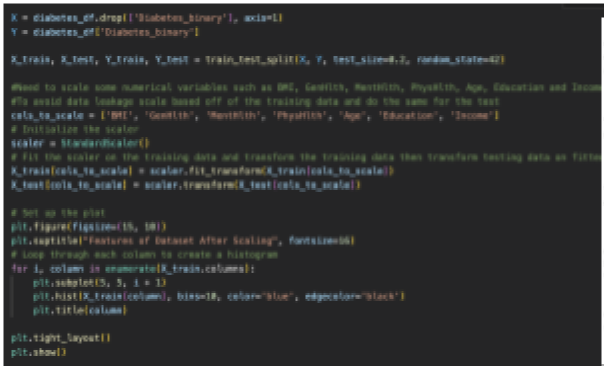
Getting into the machine learning aspect of this project we took advantage of sklearns many helpful features as well as pandas dataframes. So first we checked the data for missing values that would need to be filled, but found none.



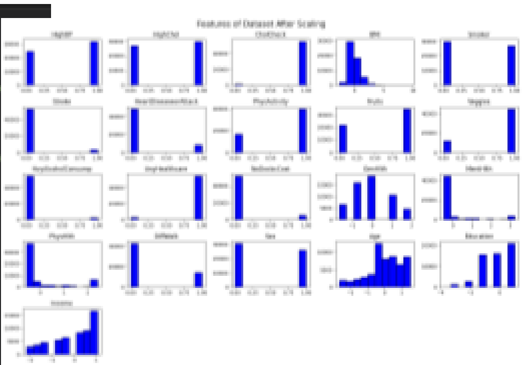
```
print(diabetes_df.isna().sum())
✓ 0.0s

Diabetes_binary        0
HighBP                 0
HighChol               0
CholCheck              0
```

Getting the amount of missing data

Next we have to split the data into training and testing, then scale the numerical data so that they work well with the various models we will train. Then we replot the features to ensure the scaling of each variable. We do these by using sklearn's "train_test_split" function as well as the "StandardScalar()". With the standard scalar make sure to fit with the training data so that we don't introduce data leakage.

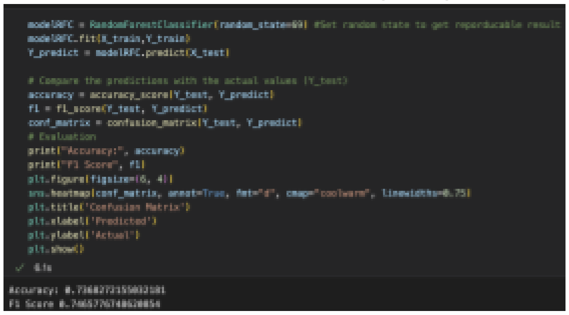


```
X = diabetes_df.drop(['Diabetes_binary'], axis=1)
Y = diabetes_df['Diabetes_binary']

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, random_state=42)

#Need to scale some numerical variables such as BMI, GenHlth, MentHlth, PhysHlth, Age, Education and Incom
#To avoid data leakage scale based off of the training data and do the same for the test
cols_to_scale = ['BMI', 'GenHlth', 'MentHlth', 'PhysHlth', 'Age', 'Education', 'Income']
# Initialize the scaler
scaler = StandardScaler()
# Fit the scaler on the training data and transform the training data then transform testing data as fitte
X_train[cols_to_scale] = scaler.fit_transform(X_train[cols_to_scale])
X_test[cols_to_scale] = scaler.transform(X_test[cols_to_scale])

# Set up the plot
plt.figure(figsize=(15, 10))
plt.suptitle("Features of Dataset After Scaling", fontsize=16)
# Loop through each column to create a histogram
for i, column in enumerate(X_train.columns):
    plt.subplot(5, 5, i + 1)
    plt.hist(X_train[column], bins=10, color='blue', edgecolor='black')
    plt.title(column)

plt.tight_layout()
plt.show()
```

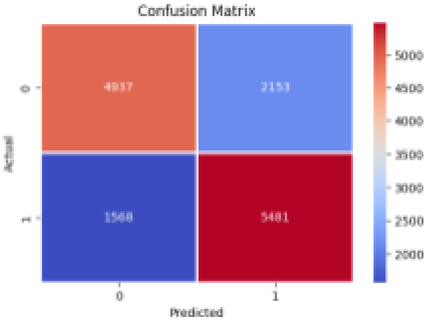


Splitting into training and testing data then Scaling features

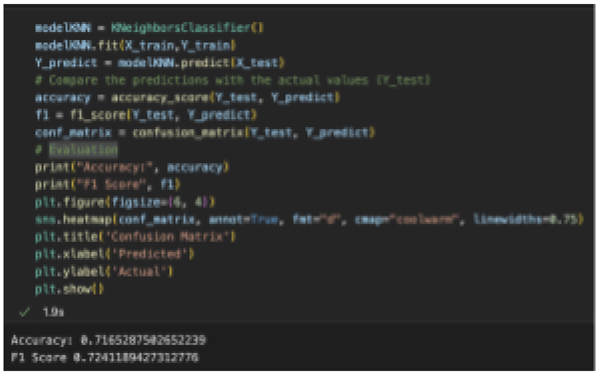
Next we continue to use sklearn to build various models including RandomFrorest, K Nearest Neighbor, Logistic Regression and Support vector Classifier. After the model is built and used on the testing data, we can evaluate the model by looking at the accuracy score, f-1 score and the confusion matrix. All of these are generated using sklearn functions. With the confusion matrix, we used seaborn to again generate a heatmap.



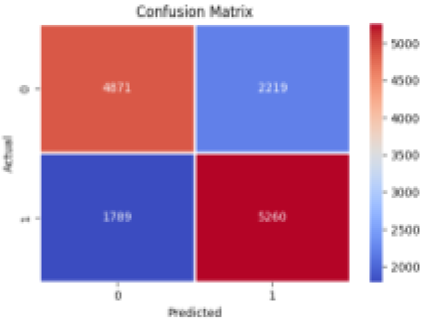
```
modelRFC = RandomForestClassifier(random_state=69) #Set random state to get reporducable result
modelRFC.fit(X_train,Y_train)
Y_predict = modelRFC.predict(X_test)

# Compare the predictions with the actual values (Y_test)
accuracy = accuracy_score(Y_test, Y_predict)
f1 = f1_score(Y_test, Y_predict)
conf_matrix = confusion_matrix(Y_test, Y_predict)
# Evaluation
print("Accuracy:", accuracy)
print("F1 Score", f1)
plt.figure(figsize=(6, 4))
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="coolwarm", linewidths=0.75)
plt.title('Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()
 ✓ 0.1s

Accuracy: 0.7368273155032181
F1 Score 0.7465776748628854
```



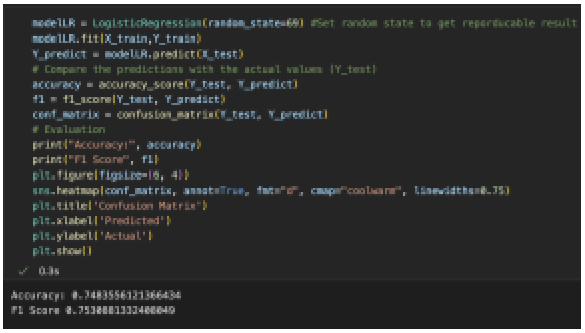
Random Forest



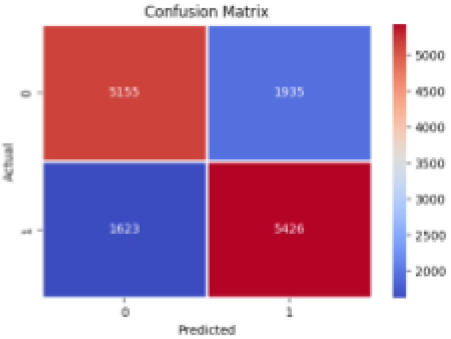
```
modelKNN = KNeighborsClassifier()
modelKNN.fit(X_train,Y_train)
Y_predict = modelKNN.predict(X_test)
# Compare the predictions with the actual values (Y_test)
accuracy = accuracy_score(Y_test, Y_predict)
f1 = f1_score(Y_test, Y_predict)
conf_matrix = confusion_matrix(Y_test, Y_predict)
# Evaluation
print("Accuracy:", accuracy)
print("F1 Score", f1)
plt.figure(figsize=(6, 4))
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="coolwarm", linewidths=0.75)
plt.title('Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()
 ✓ 1.9s

Accuracy: 0.7165287502652239
F1 Score 0.7241189427312776
```



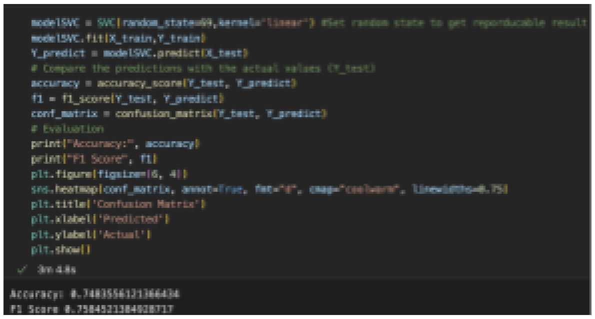
KNN



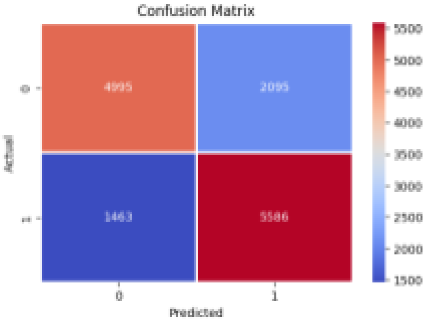
```
modelLR = LogisticRegression(random_state=69) #Set random state to get reporducable result
modelLR.fit(X_train,Y_train)
Y_predict = modelLR.predict(X_test)
# Compare the predictions with the actual values (Y_test)
accuracy = accuracy_score(Y_test, Y_predict)
f1 = f1_score(Y_test, Y_predict)
conf_matrix = confusion_matrix(Y_test, Y_predict)
# Evaluation
print("Accuracy:", accuracy)
print("F1 Score", f1)
plt.figure(figsize=(6, 4))
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="coolwarm", linewidths=0.75)
plt.title('Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()
 ✓ 0.3s

Accuracy: 0.7483556121366434
F1 Score 0.7530881332400049
```



Logistic Regression



```
modelSVC = SVC(random_state=69,kernel="linear") #Set random state to get reporducable result
modelSVC.fit(X_train,Y_train)
Y_predict = modelSVC.predict(X_test)
# Compare the predictions with the actual values (Y_test)
accuracy = accuracy_score(Y_test, Y_predict)
f1 = f1_score(Y_test, Y_predict)
conf_matrix = confusion_matrix(Y_test, Y_predict)
# Evaluation
print("Accuracy:", accuracy)
print("F1 Score", f1)
plt.figure(figsize=(6, 4))
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="coolwarm", linewidths=0.75)
plt.title('Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()
 ✓ 3m 4.8s

Accuracy: 0.7483556121366434
F1 Score 0.7584521384928717
```



Support Vector Classifier

Create a web page for Healthcare professionals

In order to make a webpage for healthcare professionals, we needed two components: the frontend and the backend. We wanted to implement the following features:

1. Searching up a patient by ID
2. Adding a patient using a patient form
3. Predicting whether or not is likely to have diabetes using the patient form

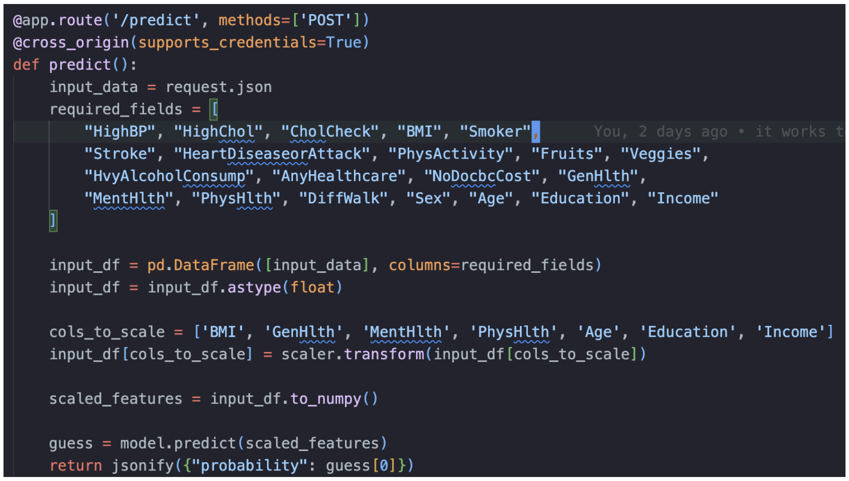
Setting up the backend:

In setting up the backend, we wanted to keep it simple and use Pyhton and Flask to create the endpoints so that the frontend can call upon to execute functions. Because we are looking up patients by ID and adding patients persistently, we need a place to store this data and have it be available to the Flask server. To do this, we need a database. The database was created in the *create_db.py* file which takes in the *diabetes.csv* file and stores every entry in a sqlite database called tutorial.db in a table called DiabetesData.

Prediction Model: the /predict endpoint

In the Flask server itself, we loaded up the trained model object from the Jupyter notebook that we exported. Furthermore, we also exported the scaler as well which was also trained on the dataset. Because this was in memory of the server, this allowed us to execute predictions based on the form data that is given to a specific endpoint.

Then, we created a predict endpoint that is also a POST request that allows the frontend to give the form data to the endpoint so that the endpoint can consume it.



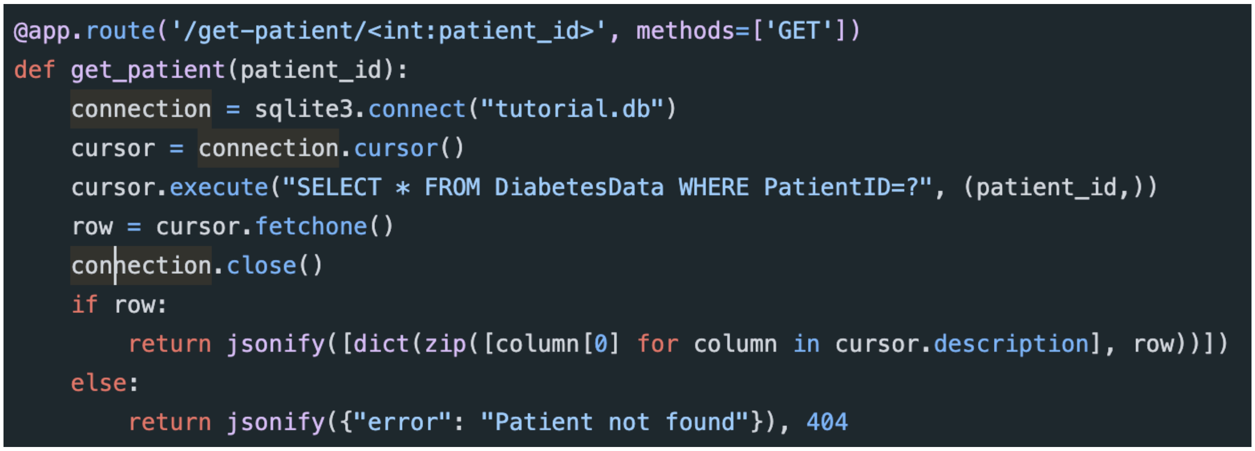
```
@app.route('/predict', methods=['POST'])
@cross_origin(supports_credentials=True)
def predict():
    input_data = request.json
    required_fields = [
        "HighBP", "HighChol", "CholCheck", "BMI", "Smoker",      You, 2 days ago • it works t
        "Stroke", "HeartDiseaseorAttack", "PhysActivity", "Fruits", "Veggies",
        "HvyAlcoholConsump", "AnyHealthcare", "NoDocbcCost", "GenHlth",
        "MentHlth", "PhysHlth", "DiffWalk", "Sex", "Age", "Education", "Income"
    ]

    input_df = pd.DataFrame([input_data], columns=required_fields)
    input_df = input_df.astype(float)

    cols_to_scale = ['BMI', 'GenHlth', 'MentHlth', 'PhysHlth', 'Age', 'Education', 'Income']
    input_df[cols_to_scale] = scaler.transform(input_df[cols_to_scale])

    scaled_features = input_df.to_numpy()

    guess = model.predict(scaled_features)
    return jsonify({"probability": guess[0]})
```

We used a module from the standard Python library called pickle that allows us to export and import models. We thought that if we could export the model, we could run it on a server that exists outside the Jupyter notebook itself.

Adding a Patient: the /add-patient endpoint

To create the endpoint to add a patient, we need to take in the patient form data from the frontend and use a POST request to consume the data. Then, we use an INSERT statement to add the data that is sent to the endpoint in order to persist the data. However, before we do that we need to make sure that the data is sanitized and everything that we need is there.

Getting a Patient: the /get-patient/<int:patient_id>



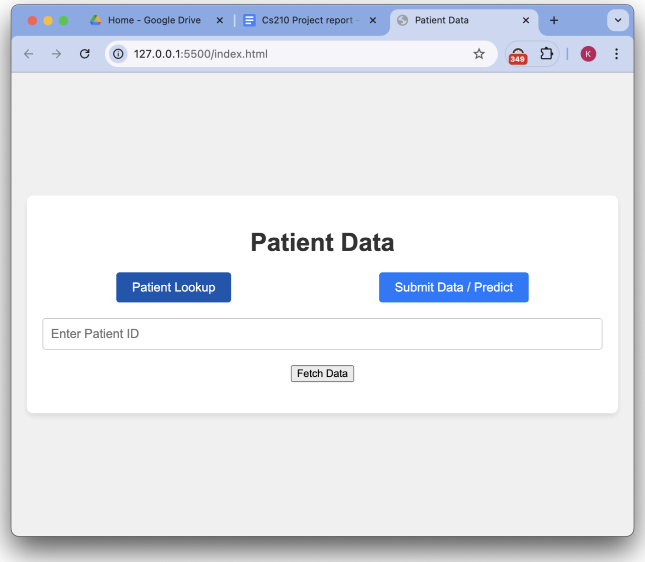
```
@app.route('/get-patient/<int:patient_id>', methods=['GET'])
def get_patient(patient_id):
    connection = sqlite3.connect("tutorial.db")
    cursor = connection.cursor()
    cursor.execute("SELECT * FROM DiabetesData WHERE PatientID=?", (patient_id,))
    row = cursor.fetchone()
    connection.close()
    if row:
        return jsonify([dict(zip([column[0] for column in cursor.description], row))])
    else:
        return jsonify({"error": "Patient not found"}), 404
```

To get the patient by patient ID, we can specify on the frontend what patient we want by the URL. Then we consume the argument that is sent to the server, and then make a SQL query to the database that we created as seen above in the code snippet. When we get the data, we can then extract the data and put it in a format that is readable by the frontend.

Making the frontend

To make the frontend, we used plain HTML, CSS, and JavaScript and opted to not use a library like React. We implemented two tabs: patient lookup and submitting data and predicting.

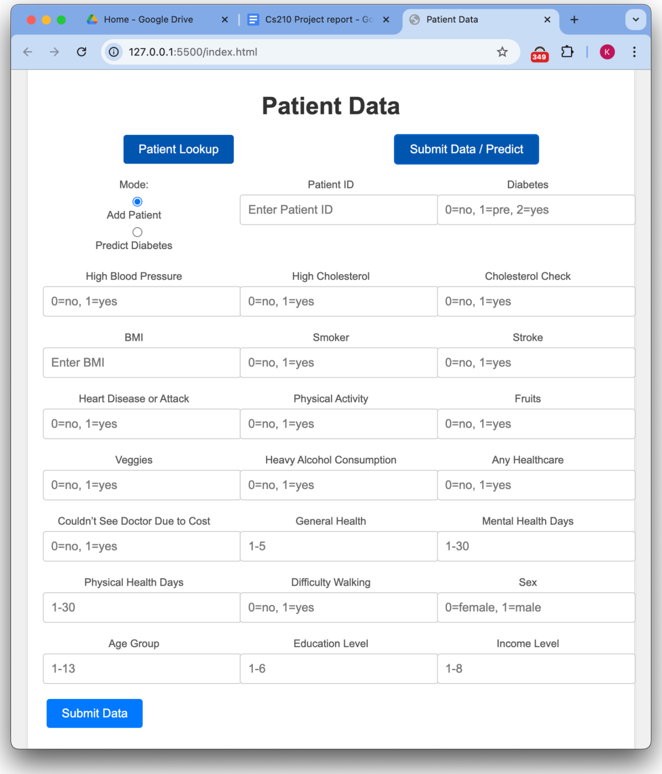
Patient Lookup



For patient lookup, we allow the user to insert an ID into a textbook and press a button called "Fetch Data". Using the number ID, we call /get-patient/<patient_id> which calls an endpoint in the database. Then, we display the contents for the user.

There is a div within the HTML file that we change the innerHTML of so that the innerHTML will contain all the different data—each on new lines.

Adding a new patient or predicting whether or not the patient is likely to be diabetic



Patient Data

Patient Lookup | Submit Data / Predict

Mode: ● Add Patient ○ Predict Diabetes	Patient ID Enter Patient ID	Diabetes 0=no, 1=pre, 2=yes
High Blood Pressure 0=no, 1=yes	High Cholesterol 0=no, 1=yes	Cholesterol Check 0=no, 1=yes
BMI Enter BMI	Smoker 0=no, 1=yes	Stroke 0=no, 1=yes
Heart Disease or Attack 0=no, 1=yes	Physical Activity 0=no, 1=yes	Fruits 0=no, 1=yes
Veggies 0=no, 1=yes	Heavy Alcohol Consumption 0=no, 1=yes	Any Healthcare 0=no, 1=yes
Couldn't See Doctor Due to Cost 0=no, 1=yes	General Health 1-5	Mental Health Days 1-30
Physical Health Days 1-30	Difficulty Walking 0=no, 1=yes	Sex 0=female, 1=male
Age Group 1-13	Education Level 1-6	Income Level 1-8

Submit Data

In this view, we have a form that contains all the different features of a patient. There are two different modes: Add Patient and Predict Diabetes.

In the Add Patient mode, it contains every possible feature.

In the Predict Diabetes, it contains every feature except “Patient ID” and “Diabetes.” Diabetes is the feature that states whether or not the patient is diabetic or not. However, if we are predicting, we should not be stating whether or not the patient is diabetic nor do we really need the patient ID since that has no relevance to whether or not the patient is diabetic. What we want are the variables that do, indeed, matter.

Depending on the current mode of the form, the button on the bottom will change to “Submit Data” or “Predict”. Submit data will hit the endpoint on the flask server to insert all the contents into the database. “Predict” will return to the user whether or not the patient is likely to be diabetic by displaying a message on the bottom.