Build a basic Diabetes detection system, which takes basic variables as input and returns if a person is Diabetic or Non-Diabetic.

Created by: Amitava Chatterjee

https://github.com/cloudsony999/diabetics-ML

Medical Science is one of the sectors which uses Machine Learning very frequently at it's core. From diagnosis and prognosis to drug development, ML is infiltrating this sector to a great extent. In fact recently, Machine Learning algorithm was able to accurately detect patients with cancer at risk of short-term mortality (***refer***).

Our Business Problem belongs to Medical Science domain i.e., predicting if a patient is diabetic or not. According to CDC, more than 34 million people in the United States itself have diabetes, and 1 in 5 of them don't know they have it (***refer***). This fact in itself is more than sufficient to build a production level system which can help actual patients who might be having Diabetes and not knowing it. So, let's clearly state our objective.

We've stated our problem which we need to keep in mind for our entire project lifecycle. Now the main challenge is to get the data. In real life scenario, to get the data for our business problem is the real challenge. This alone might consume a lot of time in your project. Sometimes there is actually no data for you to start with and sometimes it is so messy that you'll be dedicating 80% of project time to clean your data (Feature Engineering).

Fortunately, the ideal dataset required for our problem is available on **Kaggle.** Dataset name is "**Pima Indians Diabetes Database**" collected from the National Institute of Diabetes and Digestive and Kidney Disease. This dataset is small with **9** features and **768** observations which is enough to show how can we solve this problem.

It is very important to know about each and every feature in your data. Consult the domain specialist if required to know about all the features. This will make Feature Engineering step easier and less time consuming. Here's a brief description of all the features which I'm referring from data source itself for you to follow along.

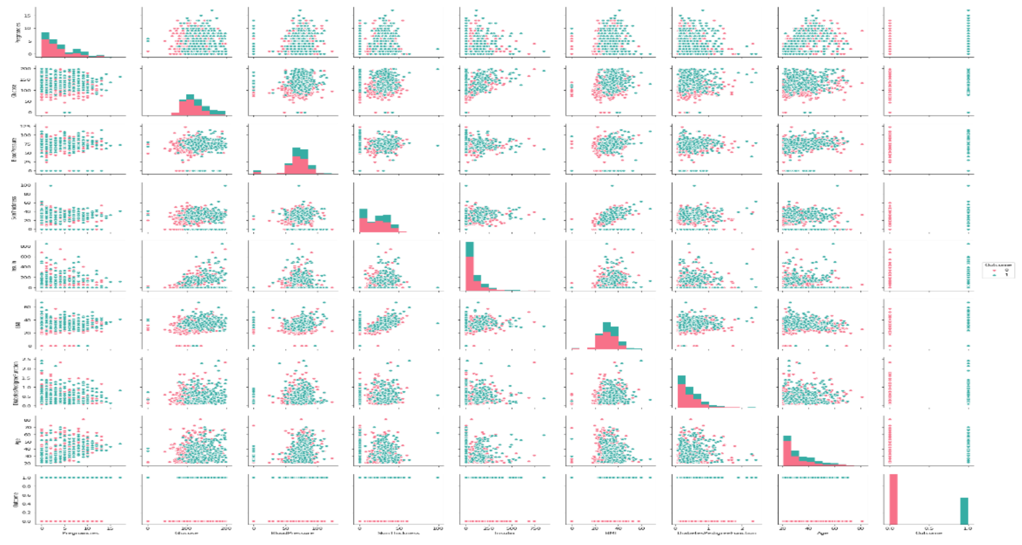
1: **Pregnancies**: Number of times pregnant
2: **Glucose**: Plasma glucose concentration 2 hours in an oral glucose tolerance test.
3: **BloodPressure**: Diastolic blood pressure (mm Hg)
4: **SkinThickness**: Triceps skin fold thickness (mm)

5: **Insulin**: 2-Hour serum insulin (mu U/ml)
6: **BMI**: Body mass index (weight in kg/ (height in m) ²)
7: **DiabetesPedigreeFunction**: Diabetes pedigree function
8: **Age**: Age (years)
9: **Outcome**: Class variable (0 or 1) 268 of 768 are 1, the others are 0

All the variables are either self-known or available in a simple Blood Test and "Outcome" is what we need to predict. Clearly dataset is simple and our objective is also well defined.

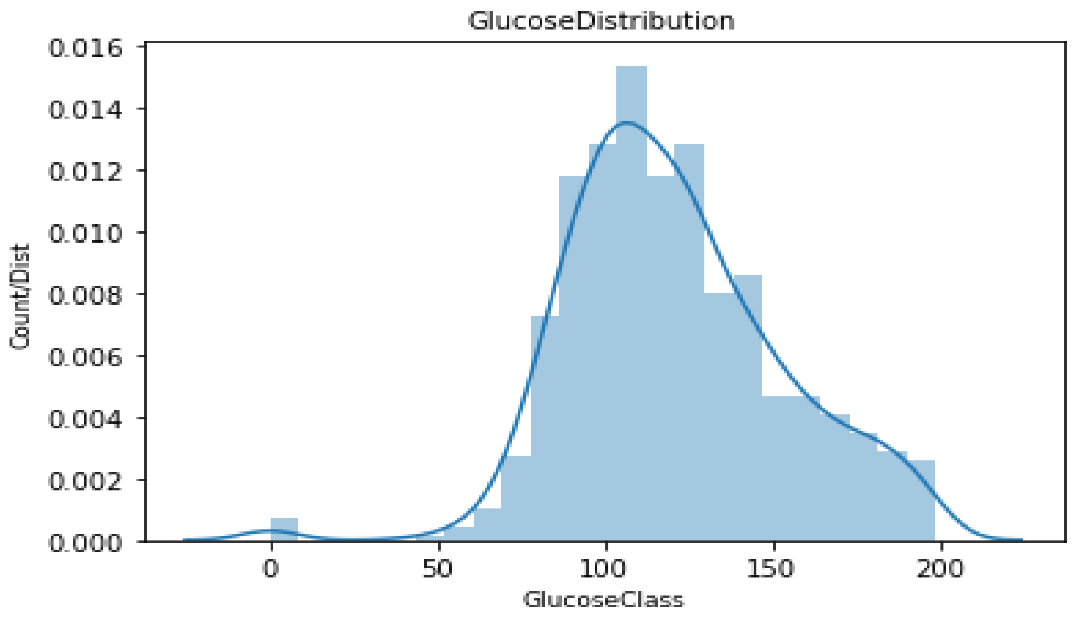
Visualizing the Data: -

Whenever it comes to knowing your data more, there cannot be a better way other than representing your data graphically. Mapping between different variables show us relationship between data which cannot be seen as a whole.

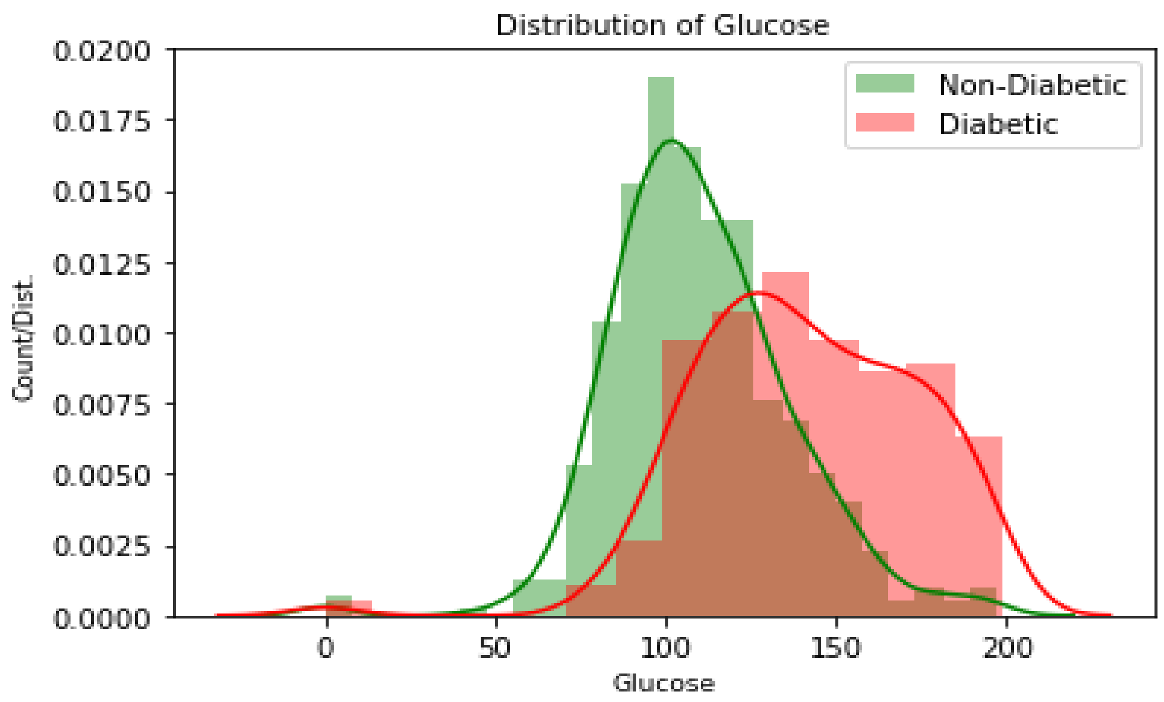


Above is the scatter plot of all variables with each other and is a great way to see relationship and a general layout. With this viz, we can clearly observe that there are lot of outliers in our data which needs to be taken care of.
After getting a glimpse with scatter plot, it's time to do **Univariate** and **Bivariate** analysis of the data. Univariate analysis as name suggests considers a single variable and creates graphical structure from it. Similarly, Bivariate analysis involves 2 variables and tries to find relationship between them.
For each variable, I observed the distribution of the variable and its relationship with our dependent variable (Outcome). Code for the 2 curves was really simple.

```
# Function to observe variable's distribution

def plot2(frame,var):
    plt2=sns.distplot(frame[frame.Outcome==False][var.name],color='green',label='Non-Diabetic')
    sns.distplot(frame[frame.Outcome==True][var.name],color='red',label='Diabetic')
    plt2.set_title('Distribution of '+var.name,fontdict={'fontsize':10})
    plt2.set_xlabel(var.name,fontdict={'fontsize':9})
    plt2.set_ylabel('Count/Dist.',fontdict={'fontsize':9})
    plt2.axes.legend(loc=0)

# Plot to compare distribution of a variable with diabetic or non-diabetic

def plot2(frame,var):
    plt2=sns.distplot(frame[frame.Outcome==False][var.name],color='green',label='Non-Diabetic')
    sns.distplot(frame[frame.Outcome==True][var.name],color='red',label='Diabetic')
    plt2.set_title('Distribution of '+var.name,fontdict={'fontsize':10})
    plt2.set_xlabel(var.name,fontdict={'fontsize':9})
    plt2.set_ylabel('Count/Dist.',fontdict={'fontsize':9})
    plt2.axes.legend(loc=0)
```



This plot if seen carefully throws a lot of information towards you. It clearly depicts that most of the Glucose levels lie between 80–120 range. Also, observe that few values are at 0. Can a person actually have 0 Glucose level in his body?
This leads to the conclusion that in our data, missing values are depicted by 0 and needs to be taken care of during our feature engineering. This behavior was observed for almost all the variables in our small dataset.



This curve shows the distribution of Glucose, color coded with Outcome variable. Overlapping between Diabetic and Non-Diabetic patient can be observed around 100–120 range. Also, higher glucose levels suggest higher probability for a person to be diabetic which is certainly a crucial observation. Another observation that can be made is based on Glucose levels, data is not linearly separable.

After repeating similar viz for all the variables, I ended my data visualization with a correlation plot, which is always a good way to see the relation between our variables. Code for correlation plot again is simple and all thanks to seaborn.

plt.figure(dpi = 125,figsize= (5,4))

mask = np.triu(df.corr())

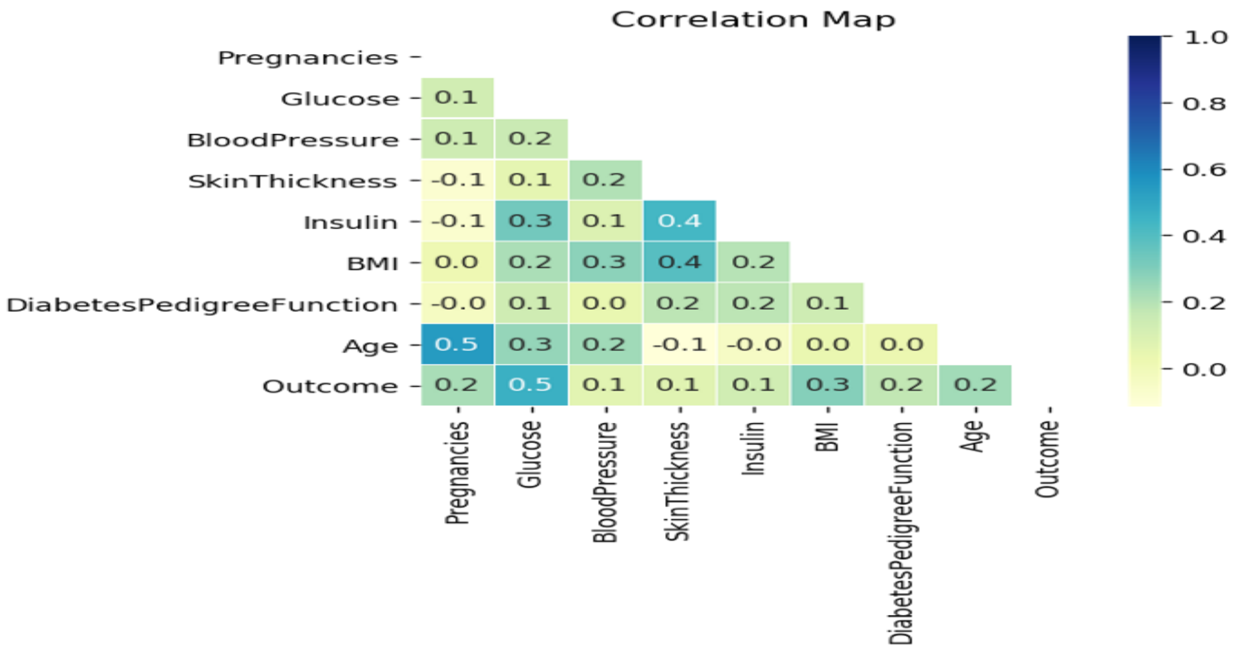
#np.triu returns lower triangle for our heatmap as we do not need upper map

sns.heatmap(df.corr(),mask = mask, fmt = ".1f",annot=True,lw=0.1,cmap = 'YlGnBu')

plt.title('Correlation Map')

plt.show()

Above code produces a beautiful correlation plot between our all 9 variables represented as a heat map. If we observe below plot carefully, we can see that Glucose and BMI are strongly correlated with our Outcome variable.



	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
Pregnancies									
Glucose	0.1								
BloodPressure	0.1	0.2							
SkinThickness	-0.1	0.1	0.2						
Insulin	-0.1	0.3	0.1	0.4					
BMI	0.0	0.2	0.3	0.4	0.2				
DiabetesPedigreeFunction	-0.0	0.1	0.0	0.2	0.2	0.1			
Age	0.5	0.3	0.2	-0.1	-0.0	0.0	0.0		
Outcome	0.2	0.5	0.1	0.1	0.1	0.3	0.2	0.2	

We will observe the changes in our correlation plot once we are done with our feature engineering step. For now, we have good knowledge about our data and it's a good time to move to Feature Engineering.

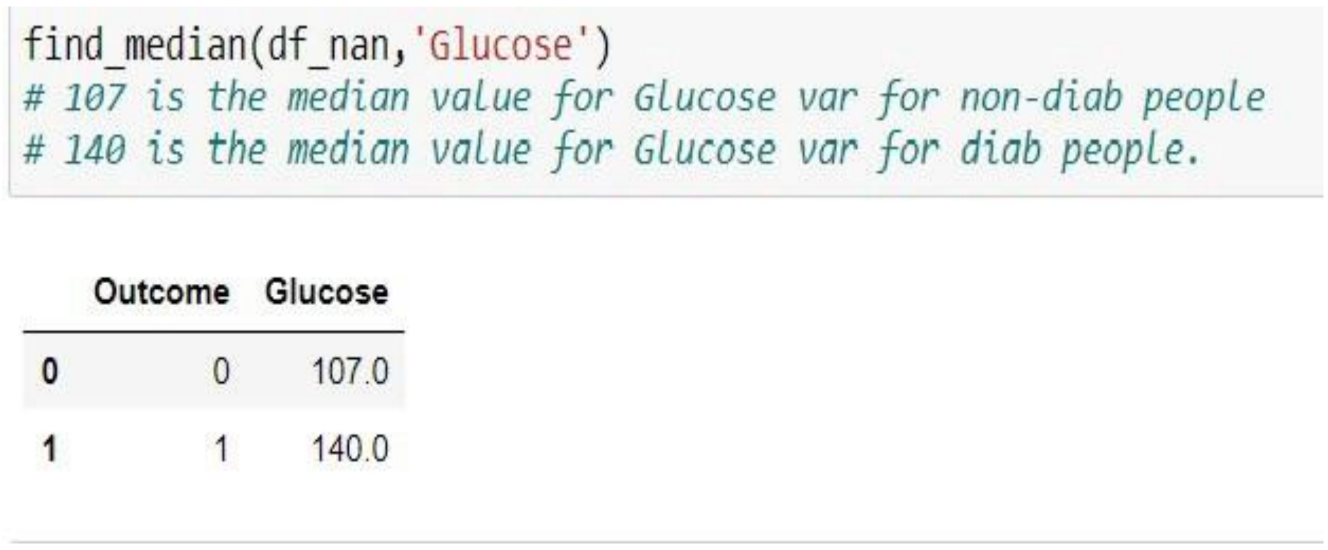
Feature Engineering

When it comes to a Machine Learning project, Feature Engineering is what's really going to make the difference for your model's performance. A person who doesn't even have any ML experience can implement models directly on data but feature engineering is where your true Engineering skills should implement.
In our case, since dataset was simple only couple of steps were required for feature engineering.

1: Handling the Null Values:

There are multiple strategies to handle Null values. There is simple mean, median or mode replacement methods that you can use which is standard approach. Amongst these, median

replacement is a better approach as 1 outlier can significantly impact mean value whereas median is more robust in these situations.
You can also opt for prediction-based replacement where you build a model just to predict missing values of a variable. This is advisable when your dataset is large and missing values are relatively very small.
In our scenario, median replacement seemed to be an ideal case. Almost all variables had few missing values and median value was used.
Now, I want to emphasize on median replacement strategy based on outcome. For this, let’s see an example.



```
find_median(df_nan,'Glucose')
# 107 is the median value for Glucose var for non-diab people
# 140 is the median value for Glucose var for diab people.
```

	Outcome	Glucose
0	0	107.0
1	1	140.0

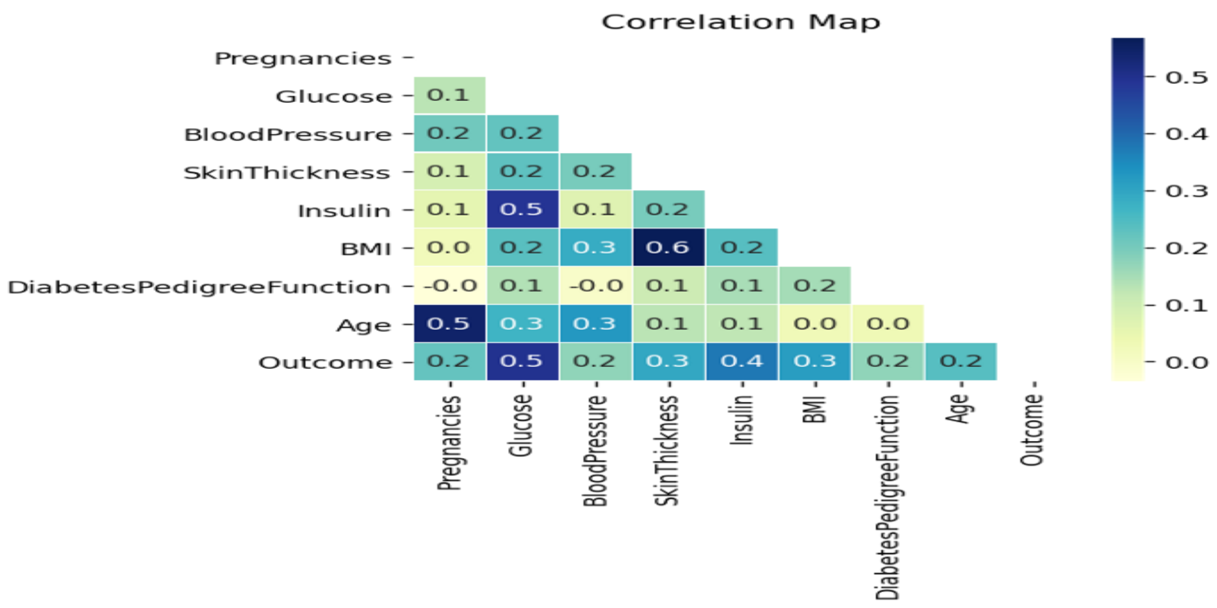
Glucose Median Values for Diabetic and Non-Diabetic patients

Firstly, 0 values were replaced by NA (too many 0s could impact median as well). Secondly, if you look above image, there’s a difference in median values for diabetic and non-diabetic patients. As we saw in Viz section also, Glucose levels were higher for diabetic patients and less for non-diabetic patients.
So, we are replacing median based on Outcome variable which seems to be more logical.

```
# Replacing all 0 values with Null values
def replace_zero(df):
    df_nan=df.copy(deep=True)
    cols = ["Glucose","BloodPressure","SkinThickness","Insulin","BMI"]
    df_nan[cols] = df_nan[cols].replace({0:np.nan})
    return df_nan
df_nan=replace_zero(df)

# Finding median
def find_median(frame,var):
    temp = frame[frame[var].notnull()]
    temp = frame[[var,'Outcome']].groupby('Outcome')[[var]].median().reset_index()
    return temp

# Replacing Null values
def replace_null(frame,var):
    median_df=find_median(frame,var)
    var_0=median_df[var].iloc[0]
    var_1=median_df[var].iloc[1]
    frame.loc[(frame['Outcome'] == 0) & (frame[var].isnull()), var] = var_0
    frame.loc[(frame['Outcome'] == 1) & (frame[var].isnull()), var] = var_1
    return frame[var].isnull().sum()

print(str(replace_null(df_nan,'Glucose'))+ ' Nulls for Glucose')
print(str(replace_null(df_nan,'SkinThickness'))+ ' Nulls for SkinThickness')
print(str(replace_null(df_nan,'Insulin'))+ ' Nulls for Insulin')
print(str(replace_null(df_nan,'BMI'))+ ' Nulls for BMI')
print(str(replace_null(df_nan,'BloodPressure'))+ ' Nulls for BloodPressure')
# Calling function for each variable
```

2: Standardization

Data Standardization is simply bringing all your variables to a common ground. If your distance is in Kilometers, height in centimeters or any other unit, standardization simply convert your data in such a way that mean is centered around 0 and maximum standard deviation around 1.
This is really important for your model building process non-standardized data can lead to uncanny results. I won't go into coding details as this can be achieved by simple Scikit-Lean's Standard Scalar module.
After all this median replacement and standardization, we should again observe our correlation plot and observe some noticeable changes.



Insulin and Glucose are now even more strongly correlated with out outcome variable!
We will use functions created here in our ML pipeline which will be discussed later. With these 2 steps carried on our dataset, we are ready to jump to everyone's favorite Data Modeling part.

Training Basic and Advance Models

Before moving onto the model building part, it is essential to make provisions for testing your model. For this Scikit-learns train_test_split is more than sufficient. I always opt for 80–20 train-test split with 10-fold Cross-Validation from train data (thumb-rule) for model's evaluations.
One final thing left is to choose model's evaluation metric. There are numerous metrics out there for Binary Classification. Confusion Metrics, F1 Score, AUC-ROC, Accuracy etc. are well known. For sake of simplicity, I chose Accuracy as everyone is comfortable with that. Now we can finally build our baseline model.

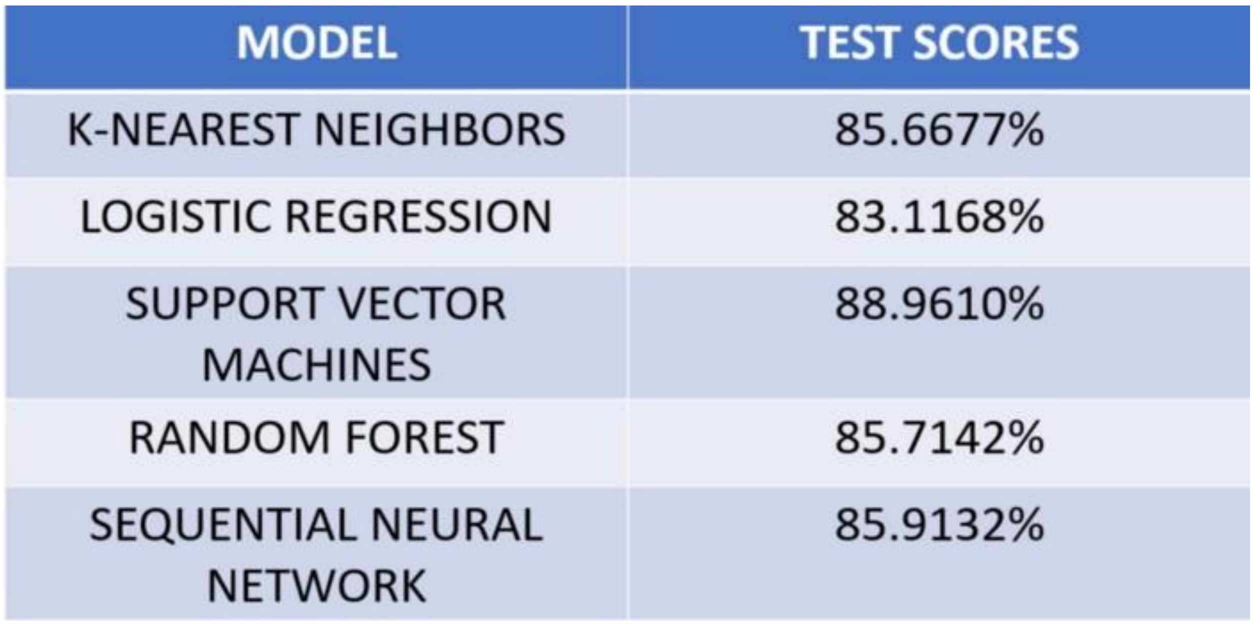
As I said earlier training the baseline models is a piece of cake and can be done very easily with Scikit-Learn. So, there's not much of a discussion required for baseline models.

A sequential neural network was also implemented with just 4 dense layer, ReLU Activation, Adam Optimizer and 1000 Epochs. This model scored well on train data (About 91%) but could manage only ~86% on test data. Here's code for our Neural Network model which I think is worth sharable.

```
def build_model():
    model = tf.keras.Sequential([
    tf.keras.layers.Dense(8, activation='relu', input_shape=[len(X_train.keys())]),
    tf.keras.layers.Dense(4, activation='relu'),
    tf.keras.layers.Dense(2, activation='relu'),
    tf.keras.layers.Dense(1,activation='sigmoid')
  ])


    optimizer = tf.keras.optimizers.Adam(learning_rate=0.01, beta_1=0.9, beta_2=0.999,
epsilon=1e-07)


    model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['accuracy'])
    return model


neural_model = build_model()


EPOCHS = 1000
neural_pred = neural_model.fit(X_train, Y_train,epochs=EPOCHS, validation_split=0.1,
verbose=2)
# Let's measure final performance
hist = pd.DataFrame(neural_pred.history)
hist['epoch'] = neural_pred.epoch
hist.tail()
# 91% accuracy on train
```

neural_test=neural_model.predict(X_test)

Below table shows all the model's performance on test data and model which was finalized for deployment was SVM as it had highest accuracy for test data.



MODEL	TEST SCORES
K-NEAREST NEIGHBORS	85.6677%
LOGISTIC REGRESSION	83.1168%
SUPPORT VECTOR MACHINES	88.9610%
RANDOM FOREST	85.7142%
SEQUENTIAL NEURAL NETWORK	85.9132%

Before moving on to the next step, SVM was dumped as a pickle file with the help of pickle module. This model would later on be used for real-time predictions.

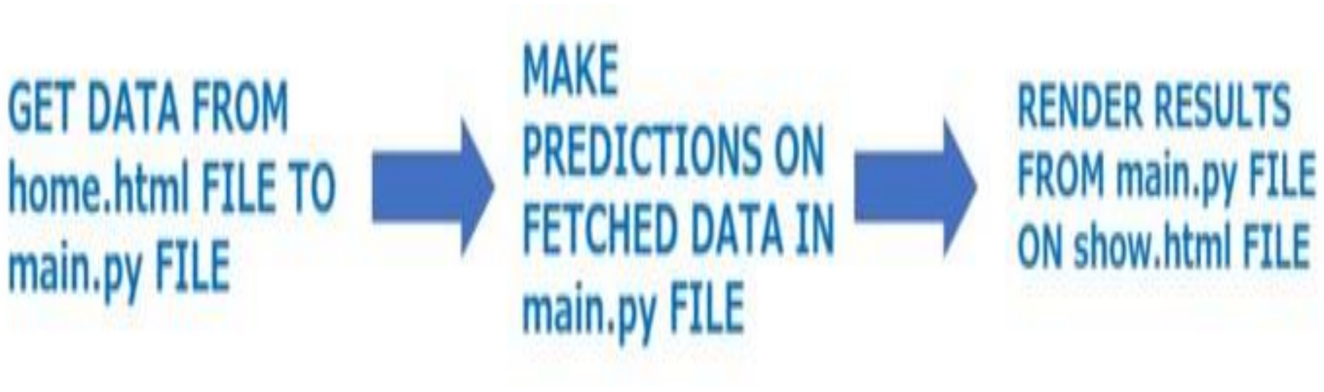
Creating HTML+CSS Layout

In order to get user input and then render your predictions as output, you need a web-page structure and some aesthetics for better user experience. We all know that HTML will provide base structure and CSS will give aesthetics to it.
Before moving onto the actual code, you have to maintain a specific file structure if you are using Flask. There should be a "**main.py**" file which is responsible for creating routes via functions to different files in your "**template folder**". You need to place all the files which main.py will route to in template folder as this is default setting of Flask.
Now we begin writing our code by creating 2 HTML files in template folder i.e. "**home.html**" and "**show.html**".
home.html is the home page of our application. This page is a basic HTML form where user will enter details of his reports which is also required for our model. Internal CSS was used for this page as aesthetics were kept simple. We need to capture responses entered by the user in our form whenever user hits the Submit button.

show.html shows the results of the model and the values he has entered in a very user-friendly manner. Here also internal CSS was implemented as simple aesthetics were more than sufficient. The best part about Flask is how simple it is to render results on web-page. show.html file is the best example for that. Have a glimpse at the code.

```
<h1>DIABETES PREDICTION</h1>
<h3>Pregnancy Value : {{ preg }}</h3>
<h3>Blood Pressure Value : {{ bp }}</h3>
<h3>Glucose Levels : {{ gluc }}</h3>
<h3>Skin Thickness : {{ st }}</h3>
<h3>Insulin Levels : {{ ins }}</h3>
<h3>BMI : {{ bmi }}</h3>
<h3>DiabetesPedigreeFunction : {{ dbf }}</h3>
<h3>Age : {{ age }}</h3>


<div class="prediction">
   <h2>Diabetes Prediction : {{ res }}</h2>
```

We will communicate with show.html file from our main.py file with just "{{ }}"**. Whatever data you need to send after your analysis on your web page just uses brackets and you are done; your results will now render. Let me summarize this section for you all for simplicity.**


GET DATA FROM home.html FILE TO main.py FILE
MAKE PREDICTIONS ON FETCHED DATA IN main.py FILE
RENDER RESULTS FROM main.py FILE ON show.html FILE

Remember, we still need to create our main.py file, which is most important file in this project and is also our final task.

Deploying Model with Flask

In this section, we'll be creating main.py file, which is heart of the project and links to our other files.
We need to start by importing Flask, render_template, request and all the other modules that are required to compute results. Then we load our saved model from "**svm_model.pkl**" file which will later return predictions as probability.
Now we need to fetch data from home.html file which is done by "**get_data()**" function. After this we use our feature engineering functions in our pipeline to transform values that user will enter.
With these transformed features, we make predictions with our model and convert probability scores to Diabetic or Non-Diabetic by keeping 0.5 as threshold value. Finally, all the values entered by user and predictions of our model are sent to show.html. Code for this is as follows

```
features = [float() for x in request.form.values()]
   final_features = [np.array(features)]


 prediction = svm_model.predict(feature_transform)
   if prediction==0:
      result="You Are Non-Diabetic"
   else:
      result="You Are Diabetic"


 return render_template('show.html', preg=Pregnancies, bp=BloodPressure,
                    gluc=Glucose, st=SkinThickness, ins=Insulin, bmi=BMI,
                    dbf=DiabetesPedigreeFunction, age=Age, res=result)
```

Conclusion

This was a very basic end to end ML project. Django could've also been used instead of Flask and same process could've been carried out. Flask is a micro-framework and very easy to grasp.

Focus was more on implementation part rather than aesthetics of the applications. There’s a scope of improvement in HTML and CSS.
Also, hyperparamter tuning was not incorporated during model building as it is time consuming and is something that you can play around with for improved results.