1) DRY (Don't repeat yourself)

Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

According to me it is one of the simplest coding principles. Its only rule is that code should not be duplicated. Instead of duplicating lines, find an algorithm that uses iteration. DRY code is easily maintainable. You can take this principle even further with model/data abstraction. The cons of the DRY principle are that you can end up with too many abstractions, external dependency creations, and complex code. DRY can also cause complications if you try to change a bigger chunk of your codebase. This is why you should avoid DRYing your code too early. It's always better to have a few repeated code sections than wrong abstractions.

For example, here in my code, there is no duplication of code. Based on sonarcloud results you can verify that there is no duplication of codes.

2) Curly's Law - Do One Thing

Curly's Law is about choosing a single, clearly defined goal for any particular bit of code: Do One Thing. Curly's Law stats that a entity (class, function, variable) should mean one thing, and one thing only. It should not mean one thing in one circumstance and carry a different value from a different domain some other time.

It should not mean two things at once. It should mean One Thing and should mean it all of the time. for example, here in my project each function works individually for a specific task. split functions, splits the data into training (X_train and y_train) and test data (X_test, y_test). Train functions, trains the training data and predicts a score for training data.

```
class ModelBuilding:

    def __init__(self, df, path=None, X_train=None, X_test=None,
    y_train=None, y_test=None):
        self.data = df
        self.X_train = X_train
        self.X_test = X_test
        self.y_train = y_train
        self.y_test = y_test
        self.y_pred = None
        Plot(df, path=path)
        self.split()
        self.train()
        self.predict()


    def split(self):
        X = self.data[['Pregnancies', 'Glucose', 'BloodPressure',
       'SkinThickness', 'Insulin',
       'BMI', 'DiabetesPedigreeFunction', 'Age']]
        y = self.data['Outcome']
        self.X_train, self.X_test, self.y_train, self.y_test = train_test_split(X, y,
        test_size=0.3)

    def train(self):
        self.rcb = RandomForestClassifier()
        self.rcb.fit(self.X_train, self.y_train)
        score = self.rcb.score(self.X_train, self.y_train)
        print("Training score: ", score)
```

3) KISS (Keep it simple, stupid)

Most systems work best if they are kept simple, rather than made complicated. The KISS principle states that most systems work best if they are kept simple rather than made complicated. Simplicity should be a key goal in design, and unnecessary complexity should be avoided.

Here in my code there is no complex model used like Neural Networks , XGBoost. A Simple Random Forest Classifier is used in my case, which is implemented in a simple way which provides a better prediction based on user inputs.

```
    #Store a dictionary into a variable
    user_data = {'pregnancies' : pregnancies,
                 'glucose' : glucose,
                 'blood_pressure' : blood_pressure,
                 'skin_thickness' : skin_thickness,
                 'insulin' : insulin,
                 'BMI' : BMI,
                 'DPF' : DPF,
                 'age' : age,
                 }

    #Transform the data into a data frame
    features = pd.DataFrame(user_data, index = [0])
    return features

#Store the user_input into a variable
user_input = get_user_input()

#Set a subheader and display the users input
st.subheader('User input:')
st.write(user_input)

#Create and train the model
RandomForestClassifier = RandomForestClassifier()
RandomForestClassifier.fit(X_train, Y_train)

#Show the models metrics
st.subheader('Model Test Accurary Score:')
st.write(str(accuracy_score(Y_test, RandomForestClassifier.predict(X_test)) * 100) + '%')

#Store the models prediction in a variable
prediction = RandomForestClassifier.predict(user_input)

#Set a subheader and display the classification
st.subheader('Classification:')
st.write(prediction)
```

4) SoC (Separation of concerns)

SoC is a design principle for separating a computer program into distinct sections such that each section addresses a separate concern. A concern is a set of information that affects the code of a computer program. A good example of SoC is MVC (Model - View - Controller). According to this principle if we decide to go with this approach, we should be careful about not to split our application into too many modules. We should only create a new module only when it makes sense to do so. More modules equal more problems. Here for my case, my program is separated into distinct functions such that each functions addresses a separate concern.

```
class ModelBuilding:

    def __init__(self, df, path=None, X_train=None, X_test=None,
    y_train=None, y_test=None):
        self.data = df
        self.X_train = X_train
        self.X_test = X_test
        self.y_train = y_train
        self.y_test = y_test
        self.y_pred = None
        Plot(df, path=path)
        self.split()
        self.train()
        self.predict()


    def split(self):
        X = self.data[['Pregnancies', 'Glucose', 'BloodPressure',
       'SkinThickness', 'Insulin',
       'BMI', 'DiabetesPedigreeFunction', 'Age']]
        y = self.data['Outcome']
        self.X_train, self.X_test, self.y_train, self.y_test = train_test_split(X, y,
        test_size=0.3)

    def train(self):
        self.rcb = RandomForestClassifier()
        self.rcb.fit(self.X_train, self.y_train)
        score = self.rcb.score(self.X_train, self.y_train)
        print("Training score: ", score)
```

5) You Aren't Gonna Need It (YAGNI)

You Aren't Gonna Need It (YAGNI) is an Extreme Programming (XP) practice which states: "Always implement things when you actually need them, never when you just foresee that you need them."
Even if you're totally, totally, totally sure that you'll need a feature, later on, don't implement it now. Usually, it'll turn out either:

- you don't need it after all, or
- what you actually need is quite different from what you foresaw needing earlier.

This doesn't mean you should avoid building flexibility into your code. It means you shouldn't overengineer something based on what you think you might need later on.
There are two main reasons to practice YAGNI:

- You save time because you avoid writing code that you turn out not to need.
- Your code is better because you avoid polluting it with 'guesses' that turn out to be more or less wrong but stick around anyway.

I implemented this principle and didn’t add any extra feature that is not needed for the model. I didn’t include any hyperparameters, multiples machine learning model. Many programmers implement a lot of extra features, multiple machine learning model and keep these codes commented for later use. But at the end these extra features are not needed for the model and it is not used at all. This increases the program complexity and length.

6) Coupling

Coupling is the degree of interdependence between software modules. It specifies how much information one class has about another class. If information is transferred from one class to another directly, we can say that coupling is tight here, however, if there is an interface between them, then we can say that it is loosely coupled. We should make our structure as loosely as possible.

If two machine learning components are coupled when at least one of them uses the other. The less these ML components know about each other, the looser they are coupled. An ML component that is only loosely coupled to its environment can be more easily changed, or replaced than a strongly coupled component. Here below in my project code all functions of a machine learning model is implemented separately and have loose coupling , they are independent. All functions, perform a distinct task and doesn't know about other functions , they all act independently .

```
class ModelBuilding:

    def __init__(self, df, path=None, X_train=None, X_test=None,
    y_train=None, y_test=None):
        self.data = df
        self.X_train = X_train
        self.X_test = X_test
        self.y_train = y_train
        self.y_test = y_test
        self.y_pred = None
        Plot(df, path=path)
        self.split()
        self.train()
        self.predict()


    def split(self):
        X = self.data[['Pregnancies', 'Glucose', 'BloodPressure',
       'SkinThickness', 'Insulin',
       'BMI', 'DiabetesPedigreeFunction', 'Age']]
        y = self.data['Outcome']
        self.X_train, self.X_test, self.y_train, self.y_test = train_test_split(X, y,
        test_size=0.3)

    def train(self):
        self.rcb = RandomForestClassifier()
        self.rcb.fit(self.X_train, self.y_train)
        score = self.rcb.score(self.X_train, self.y_train)
        print("Training score: ", score)
```

7) Cohesion

Cohesion is the extent to which two or more parts of a system are related and how they work together to create something more valuable than the individual parts. “The whole is greater than the sum of the parts.”. We definitely want more cohesive structures.

Here, a strong cohesive structure has a clear responsibility, it is being focused on a single task. Single responsibility in solid principles is actually about how cohesive we are, and how high or low the level of cohesion is. Each part of the function addresses a separate concern, a set of information that affects the code of my program. Here in my code, functions of class Modelbuilding are for the same common purpose, that is all together work for building the model in a sequential way by splitting, training the data therefore the cohesion of the classes is high.

```
class ModelBuilding:

    def __init__(self, df, path=None, X_train=None, X_test=None,
    y_train=None, y_test=None):
        self.data = df
        self.X_train = X_train
        self.X_test = X_test
        self.y_train = y_train
        self.y_test = y_test
        self.y_pred = None
        Plot(df, path=path)
        self.split()
        self.train()
        self.predict()


    def split(self):
        X = self.data[['Pregnancies', 'Glucose', 'BloodPressure',
       'SkinThickness', 'Insulin',
       'BMI', 'DiabetesPedigreeFunction', 'Age']]
        y = self.data['Outcome']
        self.X_train, self.X_test, self.y_train, self.y_test = train_test_split(X, y,
        test_size=0.3)

    def train(self):
        self.rcb = RandomForestClassifier()
        self.rcb.fit(self.X_train, self.y_train)
        score = self.rcb.score(self.X_train, self.y_train)
        print("Training score: ", score)
```

8) Mind-sized Components

Break your system down into components that are of a size you can grasp within your mind so that you can predict consequences of changes easily (dependencies, control flow, …). Here the code, is divided into small functions, that performs distinct task.

```
class Plot:

    def __init__(self, data, path):
        self.data = data
        self.path = path
        self.hist_observation()
        self.heatmap_observation()

    def hist_observation(self):
        fig, ax = plt.subplots()
        self.data[['Insulin', 'Glucose', 'BloodPressure',]].hist(bins=30, figsize=(1500,1000), ax=ax)
        fig.savefig("{}/histogram.png".format(self.path))

    def heatmap_observation(self):
        fig, ax = plt.subplots()
        d = self.data[['Pregnancies', 'Glucose', 'BloodPressure',
       'SkinThickness', 'Insulin',
       'BMI', 'DiabetesPedigreeFunction', 'Age', 'Outcome']]
        dataplot = sns.heatmap(d.corr(), annot=True, cmap="YlGnBu")
        fig.savefig("{}/heatmap.png".format(self.path))
```

9) Naming Conventions

Good names are especially important in program and often hard to find. A good name for an object should answer these questions: Why it exists? What does it do? How it is used? I tried to use self-explanatory names in all my code.

10) Principle of Least Surprise

This principle comes from Interface design for users. In this context it means, that the user should be surprised as little as possible by the interface/application. When applying it to code it is related to Naming Conventions. It states, that functions/ methods names should make absolutely clear what they are doing, also in terms of side-effects.

11) TDD: F.I.R.S.T Principles

Testing is important in Software. And unit tests are code as well. These are important rules for clean tests: It states that tests should be: Fast, Isolated/Independent, Repeatable, Self-validating, Timely. My tests fulfill all these criteria.