Clasificare:

1. Încărcă datele în X, y, aplică standardizarea input-ului și separă setul de date în 2 seturi de antrenare și testare cu o rație de 70% - 30%.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn import datasets

# Încărcarea Datelor
iris = datasets.load_iris()
X = iris.data
y = iris.target

# Standardizare
scaler = StandardScaler()
X = scaler.fit_transform(X)

# Separarea datelor
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

2. def **get_score_rf**(X, y, n_estimators):
   - Antrenează clasificatorul cerut și returnează modelul creat.

```
def get_score_rf(X, y, n_estimators):
  """
  Antrenarea clasificatorului
  INPUT
  -----
  X: features
  y: labels
  n_estimators: numarul de arbori in random forest
  OUTPUT
  ------
  rf_clf: clasificatorul antrenat
  """
  rf_clf = RandomForestClassifier(n_estimators, random_state=42)
  rf_clf.fit(x_train, y_train)
  return rf_clf
```

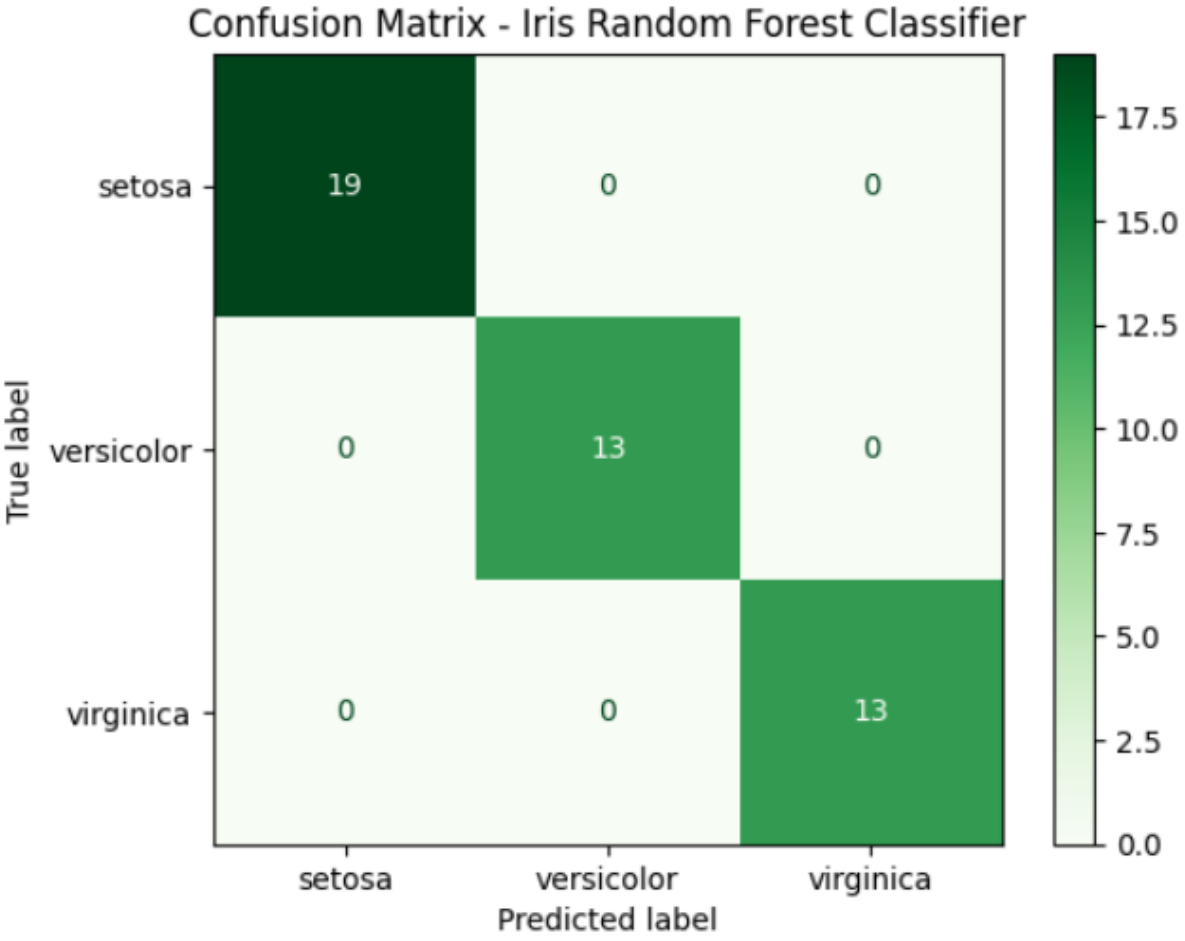
3. Accuracy:

```
y_pred = get_score_rf(x_train, y_train, 100).predict(x_test)
print("Iris Accuracy", accuracy_score(y_test, y_pred))
```

4.

```
Iris Accuracy 1.0
```

5. Matricea de Confuzie:



Se poate observa faptul că clasificatorul a prezis toate input-urile ca fiind Pozitives, de aici și accuracy = 1.

Regresie:

1. Încărcă datele în X, y, aplică standardizarea input-ului și separă setul de date în 2 seturi de antrenare și testare cu o rație de 80% - 20%.

```
from sklearn.ensemble import RandomForestRegressor

# Încărcarea datelor
diabetes = datasets.load_diabetes()
X = diabetes.data
y = diabetes.target

# Standardizare
scaler = StandardScaler()
X = scaler.fit_transform(X)

# Separarea datelor
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

2. Antrenarea regresorului folosind RandomForestRegressor din librăria sklearn.

```
def get_score_rf(x_train, y_train, n_estimators):
  """
  Antrenarea regresorului
  INPUT
  -----
  X: features
  y: labels
  n_estimators: numarul de arbori in random forest
  OUTPUT
  ------
  rf_clf: regresorul antrenat
  """
  rf_reg = RandomForestRegressor(n_estimators, random_state=42)
  rf_reg.fit(x_train, y_train)
  return rf_reg
```

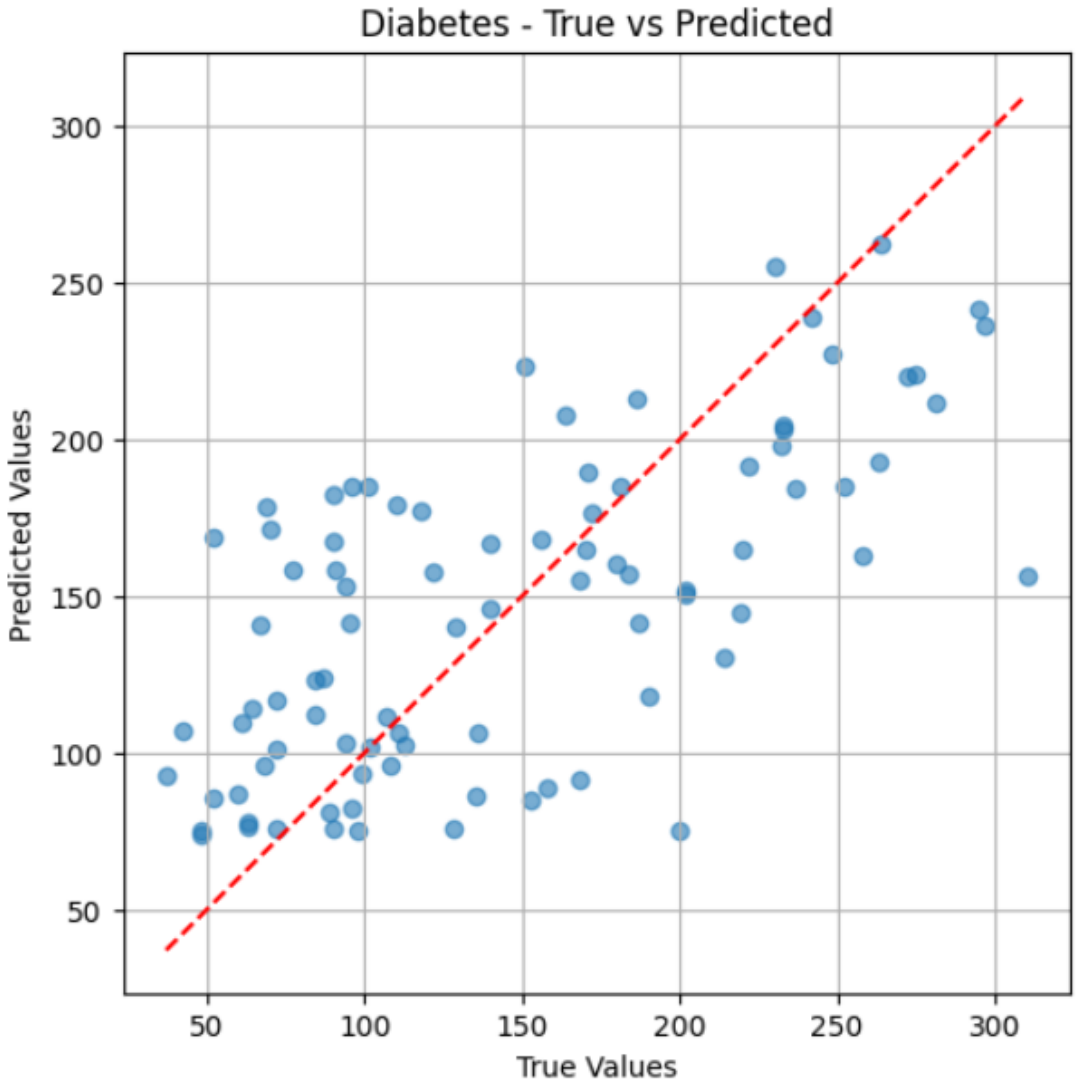
3. Aplică r2_score pentru a calcula performanța regresorului.

```
# Predicție și Evaluare
y_pred = get_score_rf(x_train, y_train, 100).predict(x_test)
diabetes_r2 = r2_score(y_test, y_pred)
print(f"Diabetes Regression R2 Score: {diabetes_r2:.4f}")
```

4.
```
Diabetes Regression R2 Score: 0.4407
```

5. def **scatter_plot_random_forest_diabetes**():

- Aplică un plot pentru a putea observa offset-ul între valorea adevărată a input-ului și valoarea prezisă de regresor.



Rețea Neuronală:

S-a propus setul de date MNIST conținând imagini cu cifre scrise de mână și presupunem implementarea unei metode Deep Learning folosin PyTorch și o rețea neuronală cu 2 straturi ascunse pentru a prezice cifra din imagine.

1.
```python
# Configurare
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])
```

2.
```python
# Încărcarea datelor
train_dataset = datasets.MNIST(root='./data', train=True, transform=transform, download=True)
test_dataset = datasets.MNIST(root='./data', train=False, transform=transform, download=True)

# Împărțirea datelor
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=1000)

class MNISTNet(nn.Module):
  """
  Rețea Neuronală cu două straturi ascunse
  """
  def __init__(self):
      super(MNISTNet, self).__init__()
      self.net = nn.Sequential(
          nn.Flatten(),
          nn.Linear(28 * 28, 128),
          nn.ReLU(),
          nn.Linear(128, 64),
          nn.ReLU(),
          nn.Linear(64, 10)
      )

  def forward(self, x):
      return self.net(x)
```

3. Definirea tabelului de valori pentru diversi parametri.

```python
# Lista cu optimizatori și rate de învățare
optimizers_dict = {
    'Adam': optim.Adam,
    'SGD': optim.SGD,
    'GD': lambda params, lr: optim.SGD(params, lr=lr)  # GD = SGD fără momentum
}
learning_rates = [0.01, 0.001, 0.0001]

# Tabel Markdown
markdown_table = "| Optimizer | Learning Rate | Accuracy | Time (s) |\n"
markdown_table += "|-----------|---------------|----------|----------|\n"
```

4.
```python
# Antrenare și evaluare pentru fiecare combinație de optimizer si learning_rate
for opt_name, opt_func in optimizers_dict.items():
    for lr in learning_rates:
        model = MNISTNet().to(device)
        criterion = nn.CrossEntropyLoss()
        optimizer = opt_func(model.parameters(), lr=lr)

        start = time.time()
        for epoch in range(5):
            model.train()
            for data, target in train_loader:
                data, target = data.to(device), target.to(device)
                optimizer.zero_grad()
                output = model(data)
                loss = criterion(output, target)
                loss.backward()
                optimizer.step()
        training_time = time.time() - start

        # Evaluare
        model.eval()
        correct, total = 0, 0
        with torch.no_grad():
            for data, target in test_loader:
                data, target = data.to(device), target.to(device)
                output = model(data)
                preds = output.argmax(dim=1)
                correct += (preds == target).sum().item()
                total += target.size(0)

        accuracy = correct / total
        markdown_table += f"| {opt_name} | {lr} | {accuracy:.4f} | {training_time:.2f} |\n"
markdown_table
```

5.

Optimizer	Learning Rate	Accuracy	Time (s)
Adam	0.01	0.9669	66.17
Adam	0.001	0.9729	51.23
Adam	0.0001	0.9381	50.17
SGD	0.01	0.9118	42.86
SGD	0.001	0.6092	44.21
SGD	0.0001	0.2065	44.15
GD	0.01	0.9136	43.96
GD	0.001	0.6448	44.29
GD	0.0001	0.1199	45.23

Se poate observa faptul că cele mai bune rezultate au fost obținute pentru optimizatorul ADAM și learning rate-ul de 0.001. Pentru multe dintre modele (SGD si lr = 0.001) nici macar nu reusea o convergență către valoarea dorită, dat fiind că lr-ul era prea mic.