**Aim**

To predict the yield strength of steel using machine learning and optimize the predictions using Particle Swarm Optimization (PSO).

**Objective**

1. To build a regression model to predict the yield strength of steel.
2. To evaluate the model's performance using metrics like RMSE and R-squared.
3. To optimize the feature parameters for better predictions using PSO.

**Summary**

The project involves predicting the yield strength of steel using a Random Forest Regressor and optimizing the prediction using Particle Swarm Optimization (PSO). The process includes data preparation, model training, evaluation, and optimization of the prediction process. The PSO technique is implemented both manually and using the PySwarms library to find the optimal set of features that maximize the yield strength.

**Tools and Libraries Used**

- Python
- Pandas
- Scikit-learn
- Matplotlib
- Numpy
- PySwarms

**Procedure**

1. **Data Preparation:**
    - Load the steel strength dataset into a pandas DataFrame.
    - Define feature set X by dropping irrelevant columns and target variable y as the yield strength.
2. **Train-Test Split:**
    - Split the dataset into training and testing sets using an 80-20 ratio.
3. **Model Training:**
    - Train a Random Forest Regressor on the training data.
    - Predict the yield strength on the test data.
    - Calculate and print the Root Mean Squared Error (RMSE).

4. **Model Evaluation:**
    - Plot actual vs. predicted yield strength values.
    - Calculate and print the R-squared value to evaluate model fit.
5. **Feature Importance:**
    - Determine and plot the importance of each feature using the trained Random Forest model.
6. **PSO Optimization:**
    - Manually implement PSO to optimize the feature parameters.
    - Use the PySwarms library to further optimize the feature parameters and compare results.

CODE:

```
#Read the csv file and capture data into a pandas dataframe

import pandas as pd


df = pd.read_csv("/content/drive/MyDrive/Steel stength/steel_strength.csv")

#Define the X values by dropping all irrelevant columns

X = df.drop(columns=["formula", "elongation", "tensile strength", "yield strength"])

#Define y

y = df['yield strength']

#Split data into train and test sets.

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X,y,

test_size=0.2, random_state = 42)

#Import the ML algorithm for regression - here, we will import the Random Forest Regressor from scikit-learn.

from sklearn.ensemble import RandomForestRegressor

model = RandomForestRegressor(n_estimators=100, max_depth=100)

model.fit(X_train, y_train)

#Predict using the trained model on our test data.

y_pred = model.predict(X_test)
```

```
from sklearn.metrics import mean_squared_error
import numpy as np

rmse = np.sqrt(mean_squared_error(y_test, y_pred))

print("RMSE: ", rmse)
#root mean squared error

#Plot actual vs. predicted to see if they fall into approximately a straight line.
import matplotlib.pyplot as plt

plt.scatter(y_test, y_pred)
plt.title("Actual vs. Predicted")
plt.xlabel("Actual")
plt.ylabel("Predicted")

#check how good the fit is between actual and predicted by fitting to a linear regression and calculating R squared.
from sklearn.linear_model import LinearRegression

#initiate linear regression model
lin_model = LinearRegression()
#fit regression model
lin_model.fit((y_test.values.reshape(-1, 1)), (y_pred))
#calculate R-squared of regression model
r_squared = lin_model.score((y_test.values.reshape(-1, 1)), (y_pred))
print("R squared value is: ", r_squared)
#check the feature ranking to assess the importance of each dfeature, as reported by Random Forest.
# Get feature names from training data
```

```
features = X_train.columns
# Extract importances from model
importances = model.feature_importances_
# Create a series with feature names and importances
feat_imp = pd.Series(importances, index=features)
# Plot 10 most important features
feat_imp.sort_values().plot(kind="barh")
plt.xlabel("Importance")
plt.ylabel("Feature")
plt.title("Feature Importance");
#define a new model object, which we will train on the full data set.
model_full= RandomForestRegressor(n_estimators=100, max_depth=100,
random_state =42)
model_full.fit(X, y)
#Using PSO.

#Let us use our own PSO code first. Then, we will do the same optimization using the pyswarms library.
!pip install pyswarms
# Define the objective function
def obj_fun(X):
   X = [X]
   results = model_full.predict(X)
   return -results

# Define the boundaries
boundaries = (np.array([[df['c'].min(), df['c'].max()],
                        [df['mn'].min(), df['mn'].max()],
                        [df['si'].min(), df['si'].max()],
                        [df['cr'].min(), df['cr'].max()],
```

```
                [df['ni'].min(), df['ni'].max()],
                [df['mo'].min(), df['mo'].max()],
                [df['v'].min(), df['v'].max()],
                [df['n'].min(), df['n'].max()],
                [df['nb'].min(), df['nb'].max()],
                [df['co'].min(), df['co'].max()],
                [df['w'].min(), df['w'].max()],
                [df['al'].min(), df['al'].max()],
                [df['ti'].min(), df['ti'].max()]]))

boundaries = np.array([[df['c'].min(), df['mn'].min(), df['si'].min(), df['cr'].min(), df['ni'].min(), df['mo'].min(), df['v'].min(), df['n'].min(), df['nb'].min(), df['co'].min(), df['w'].min(), df['al'].min(), df['ti'].min()], [df['c'].max(), df['mn'].max(), df['si'].max(), df['cr'].max(), df['ni'].max(), df['mo'].max(), df['v'].max(), df['n'].max(), df['nb'].max(), df['co'].max(), df['w'].max(), df['al'].max(), df['ti'].max()]])

# Define the parameters of the optimization
n_parameters = 13
n_particles = 100
max_iterations = 30
w = 0.5
c1 = 0.8
c2 = 0.9

# Initialize the particles and velocities
particles = np.random.uniform(low=boundaries[0], high=boundaries[1], size=(n_particles, n_parameters))
velocities = np.zeros((n_particles, n_parameters))

# Initialize the best positions and best costs
best_positions = particles.copy()
```

```
best_costs = np.array([obj_fun(p) for p in particles])

# Initialize the global best position and global best cost
global_best_position = particles[0].copy()
global_best_cost = best_costs[0]

# Perform the optimization
for i in range(max_iterations):
    # Update the velocities
    r1, r2 = np.random.rand(n_particles, n_parameters), np.random.rand(n_particles, n_parameters)
    cognitive_component = c1 * r1 * (best_positions - particles)
    social_component = c2 * r2 * (global_best_position - particles)
    velocities = w * velocities + cognitive_component + social_component

    # Update the particles
    particles += velocities

    # Enforce the bounds of the search space
    particles = np.clip(particles, boundaries[0], boundaries[1])

    # Evaluate the objective function
    costs = np.array([obj_fun(p) for p in particles])

    # Update the best positions and best costs
    is_best = costs < best_costs
    #best_positions[is_best] = particles[is_best]
    best_positions[is_best[:, 0]] = particles[is_best[:, 0]]
```

```
    best_costs[is_best] = costs[is_best]

    # Update the global best position and global best cost
    global_best_index = np.argmin(best_costs)
    global_best_position = best_positions[global_best_index].copy()
    global_best_cost = best_costs[global_best_index]

    # Print the progress
    #print(f'Iteration {i+1}: Best Cost = {global_best_cost:.6f}')
    print(f'Iteration {i+1}: Best Cost = {global_best_cost.item():.6f}')
print('Best Position:', global_best_position)
print('Best Cost:', global_best_cost)
import pyswarms as ps

# Define the objective function
def obj_fun(X):
    #X = [X]
    #results = model_full.predict(X)
    results = model_full.predict(X.reshape(1, -1))

    return -results

# Define the boundaries
boundaries = np.array([[df['c'].min(), df['mn'].max()],
                       [df['mn'].min(), df['mn'].max()],
                       [df['si'].min(), df['si'].max()],
                       [df['cr'].min(), df['cr'].max()],
                       [df['ni'].min(), df['ni'].max()],
                       [df['mo'].min(), df['mo'].max()],
```

```
                [df['v'].min(), df['v'].max()],
                [df['n'].min(), df['n'].max()],
                [df['nb'].min(), df['nb'].max()],
                [df['co'].min(), df['co'].max()],
                [df['w'].min(), df['w'].max()],
                [df['al'].min(), df['al'].max()],
                [df['ti'].min(), df['ti'].max()]])


boundaries = tuple(map(tuple, np.transpose(boundaries)))
# Define the PSO optimizer
options = {'c1': 0.8, 'c2': 0.9, 'w': 0.5}

### N O T E ###
#IN our case, the RandomForest regressor works with 1D data, so our objective
# function forces us to use only 1 particle when working with this library.
optimizer = ps.single.GlobalBestPSO(n_particles=1, dimensions=13, options=options,
bounds=boundaries)


# Run the PSO optimizer
cost, pos = optimizer.optimize(obj_fun, iters=100)
# Print the results
print("Optimized cost:", -cost)
print("Optimized parameters:", pos)
```

**Highlights**

- **Model Evaluation:** The project includes visualizing the actual vs. predicted values and calculating the R-squared value for assessing model performance.
- **Feature Importance:** The importance of each feature is analyzed and visualized to understand their contribution to the model.
- **PSO Optimization:** Both manual and library-based PSO implementations are used to optimize the feature parameters, demonstrating the effectiveness of PSO in improving model predictions.

**Conclusion**

The project successfully demonstrates the prediction of steel yield strength using a Random Forest Regressor and the enhancement of predictions through Particle Swarm Optimization. The RMSE and R-squared metrics provide a clear understanding of the model's performance, and the feature importance analysis highlights the most significant factors influencing the yield strength. The use of PSO effectively optimizes the feature parameters, leading to improved prediction accuracy.