```
pip install deap
```

```
Collecting deap
  Downloading deap-1.4.1.tar.gz (1.1 MB)
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.1/1.1 MB 1.6 MB/s eta 0:00:00a
0:00:010m
ents to build wheel ... etadata (pyproject.toml) ... ent already satisfied:
numpy in
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-
packages (from deap) (1.24.2)
Building wheels for collected packages: deap
  Building wheel for deap (pyproject.toml) ... e=deap-1.4.1-cp311-cp311-
macosx_10_9_universal2.whl size=111593
sha256=878dc1883198ef51bc285139bd72eeda35e03f0138feb9592766a79f78244525
  Stored in directory:
/Users/omkarsomeshwarkondhalkar/Library/Caches/pip/wheels/f8/64/b8/65eacfbff3
024ae2e2beb22e691d5c8abb89fbd863b8049b5f
Successfully built deap
Installing collected packages: deap
Successfully installed deap-1.4.1
Note: you may need to restart the kernel to use updated packages.
```

```python
import numpy as np
import random
from deap import base, creator, tools, algorithms
from keras.models import Sequential
from keras.layers import Dense

def create_model(weights):
    model = Sequential()
    model.add(Dense(16, input_dim=8, activation='relu',
kernel_initializer='random_uniform', bias_initializer='zeros'))
    model.add(Dense(8, activation='relu',
kernel_initializer='random_uniform', bias_initializer='zeros'))
    model.add(Dense(1, activation='linear',
kernel_initializer='random_uniform', bias_initializer='zeros'))

    # Convert the relevant parts of the weights list to numpy arrays
    weights_array_1 = np.array(weights[:128])
    weights_array_2 = np.array(weights[128:224])
    weights_array_3 = np.array(weights[224:264])
    weights_array_4 = np.array(weights[264:304])
    weights_array_5 = np.array(weights[304:305])
    weights_array_6 = np.array(weights[305:306])

    # Reshape the arrays
    model.layers[0].set_weights([weights_array_1.reshape(8, 16),
weights_array_2])
    model.layers[1].set_weights([weights_array_3.reshape(16, 8),
weights_array_4])
```

```
    model.layers[2].set_weights([weights_array_5.reshape(8, 1),
weights_array_6])

    return model
# The rest of the code remains the same...



# Define the fitness function
def evaluate(individual):
    model = create_model(individual)
    # Evaluate the model using the spray drying dataset
    # Return the mean squared error of the predictions
    return (model.evaluate(X_train, y_train)[0],)

# Define the genetic algorithm parameters
population_size = 100
mutation_rate = 0.1
cxpb = 0.5
ngen = 50

# Initialize the genetic algorithm
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
creator.create("Individual", list, fitness=creator.FitnessMax)

toolbox = base.Toolbox()
toolbox.register("attr_float", random.uniform, -1, 1)
toolbox.register("individual", tools.initRepeat, creator.Individual,
toolbox.attr_float, n=32*8+32*16+16*1)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
toolbox.register("evaluate", evaluate)
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutGaussian, mu=0, sigma=0.1,
indpb=mutation_rate)
toolbox.register("select", tools.selTournament, tournsize=3)

# Optimize the population size and mutation rate
for population_size in range(50, 200, 50):
    for mutation_rate in np.arange(0.05, 0.3, 0.05):
        print(f"Population size: {population_size}, Mutation rate:
{mutation_rate}")
        stats = tools.Statistics(lambda ind: ind.fitness.values)
        stats.register("avg", np.mean)
        stats.register("min", np.min)
        stats.register("max", np.max)
        pop = toolbox.population(n=population_size)
        hof = tools.HallOfFame(1)
        algorithms.eaSimple(pop, toolbox, cxpb=cxpb, mutpb=mutation_rate,
ngen=ngen, stats=stats, halloffame=hof, verbose=True)
```

```
        best_individual = hof[0]
        best_fitness = best_individual.fitness.values[0]
        print(f"Best fitness: {best_fitness}")
```

```
Population size: 50, Mutation rate: 0.05
```