Python
Machine Learning
Hyperparameters

Integrated Master's in Informatics Engineering

Learning and Extraction of Knowledge

2018/2019

Synthetic Intelligence Lab

Filipe Gonçalves

César Analide

Hyperparameter Optimization

- Machine learning models are parameterized so that their behavior can be tuned for a given problem

- Finding the best combination of parameters is a final step in the process of applied machine learning before presenting results

- This search process is called "Hyperparameter optimization"

Hyperparameter Optimization Strategies

- Different search strategies exist to find the best parameters (must be guided by a performance metric, e.g. accuracy, recall, f1-score, etc.):
  - **Grid search** – also called parameter sweep, it exhaustively searches through a <u>manually specified subset</u> of the hyperparameter space of a learning algorithm
    - Suffers when evaluating the number of hyperparameters grows exponentially
    - No guarantee that the search will produce the perfect solution
  - **Random Search** - random combinations of the hyperparameters are used to find the best solution for the built model
    - Yields in the most part better results then Grid Search
    - However, selected parameters are completely random - no intelligence is used to sample these combinations
  - **Evolutionary optimization** - uses evolutionary algorithms to search the space of hyperparameters for a given algorithm
    - Black-box strategy – complex interpretation of the results
  - Among others (e.g. **Bayesian optimization, Gradient-based optimization, ...**)

Grid Search - Ridge Regression Algorithm

```python
# Exhaustive Grid Search for Algorithm Tuning
import numpy as np
from sklearn import datasets
from sklearn.linear_model import Ridge
from sklearn.model_selection import GridSearchCV

# load the diabetes datasets
dataset = datasets.load_diabetes()

# prepare a range of alpha values to test
alphas = np.array([1,0.1,0.01,0.001,0.0001,0])

# create and fit a ridge regression model, testing each alpha
model = Ridge()
grid = GridSearchCV(estimator=model, param_grid=dict(alpha=alphas))
grid.fit(dataset.data, dataset.target)
print(grid)

# summarize the results of the grid search
print(grid.best_score_)
print(grid.best_estimator_.alpha)
```

Random Search - Ridge Regression Algorithm

```python
# Randomized Search for Algorithm Tuning
import numpy as np
from scipy.stats import uniform as sp_rand
from sklearn import datasets
from sklearn.linear_model import Ridge
from sklearn.model_selection import RandomizedSearchCV

# load the diabetes datasets
dataset = datasets.load_diabetes()

# prepare a uniform distribution to sample for the alpha parameter
param_grid = {'alpha': sp_rand()}

# create and fit a ridge regression model, testing random alpha values
model = Ridge()
rsearch = RandomizedSearchCV(estimator=model, param_distributions=param_grid, n_iter=100)
rsearch.fit(dataset.data, dataset.target)
print(rsearch)

# summarize the results of the random parameter search
print(rsearch.best_score_)
print(rsearch.best_estimator_.alpha)
```

Grid & Random Search - Support Vector Machine

```
import pandas as pd
import numpy as np
from sklearn import preprocessing
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV
from sklearn.svm import SVC as svc
from sklearn.metrics import make_scorer, roc_auc_score
from scipy import stats

# DATA PREPARATION
df = pd.read_csv("credit_count.txt")
y = df[df.CARDHLDR == 1].DEFAULT.values
x = preprocessing.scale(df[df.CARDHLDR == 1].ix[:, 2:12], axis = 0)

# DEFINE MODEL AND PERFORMANCE MEASURE
mdl = svc(probability = True, random_state = 1)
auc = make_scorer(roc_auc_score)

# GRID SEARCH FOR 20 COMBINATIONS OF PARAMETERS
grid_list = {"C": np.arange(2, 10, 2),
             "gamma": np.arange(0.1, 1, 0.2)}

grid_search = GridSearchCV(mdl, param_grid = grid_list, n_jobs = 4, cv = 3, scoring = auc)
grid_search.fit(x, y)
grid_search.cv_results_

# RANDOM SEARCH FOR 20 COMBINATIONS OF PARAMETERS
rand_list = {"C": stats.uniform(2, 10),
             "gamma": stats.uniform(0.1, 1)}

rand_search = RandomizedSearchCV(mdl, param_distributions = rand_list, n_iter = 20, n_jobs = 4, cv = 3, random_state = 2017, scoring = auc)
rand_search.fit(x, y)
rand_search.cv_results_
```

Evolutionary Optimization Implementation

1. Generate the initial population of individuals randomly (first generation)
2. Evaluate the fitness of each individual in that population (time limit, performance achieved, etc.)
3. Repeat the following regenerational steps until termination:
   - Select the best-fit individuals for reproduction. (Parents)
   - Breed new individuals through crossover and mutation operations to give birth to offspring.
   - Evaluate the individual fitness of new individuals.
   - Replace least-fit population with new individuals.

Genetic Algorithm Module

```python
import numpy

def cal_pop_fitness(equation_inputs, pop):
    # Calculating the fitness value of each solution in the current population.
    # The fitness function caulcuates the sum of products between each input and its corresponding weight.
    fitness = numpy.sum(pop*equation_inputs, axis=1)
    return fitness

def select_mating_pool(pop, fitness, num_parents):
    # Selecting the best individuals in the current generation as parents for producing the offspring of the next generation.
    parents = numpy.empty((num_parents, pop.shape[1]))
    for parent_num in range(num_parents):
        max_fitness_idx = numpy.where(fitness == numpy.max(fitness))
        max_fitness_idx = max_fitness_idx[0][0]
        parents[parent_num, :] = pop[max_fitness_idx, :]
        fitness[max_fitness_idx] = -99999999999
    return parents

def crossover(parents, offspring_size):
    offspring = numpy.empty(offspring_size)
    # The point at which crossover takes place between two parents. Usually it is at the center.
    crossover_point = numpy.uint8(offspring_size[1]/2)

    for k in range(offspring_size[0]):
        # Index of the first parent to mate.
        parent1_idx = k%parents.shape[0]
        # Index of the second parent to mate.
        parent2_idx = (k+1)%parents.shape[0]
        # The new offspring will have its first half of its genes taken from the first parent.
        offspring[k, 0:crossover_point] = parents[parent1_idx, 0:crossover_point]
        # The new offspring will have its second half of its genes taken from the second parent.
        offspring[k, crossover_point:] = parents[parent2_idx, crossover_point:]
    return offspring

def mutation(offspring_crossover):
    # Mutation changes a single gene in each offspring randomly.
    for idx in range(offspring_crossover.shape[0]):
        # The random value to be added to the gene.
        random_value = numpy.random.uniform(-1.0, 1.0, 1)
        offspring_crossover[idx, 4] = offspring_crossover[idx, 4] + random_value
return offspring_crossover
```

Evolutionary Optimization – Part I

```
import numpy
import GA

"""
The y=target is to maximize this equation:
    y = w1.x1 + w2.x2 + w3.x3 + w4.x4 + w5.x5 + w6.x6
    where (x1,x2,x3,x4,x5,x6)=(4,-2,3.5,5,-11,-4.7)
    What are the best values for the 6 weights w1 to w6?
    We are going to use the genetic algorithm for the best possible values after a number of generations.
"""
# Inputs of the equation.
equation_inputs = [4,-2,3.5,5,-11,-4.7]

# Number of the weights we are looking to optimize.
num_weights = 6
"""
Genetic algorithm parameters:
    Mating pool size
    Population size
"""
sol_per_pop = 8
num_parents_mating = 4

# Defining the population size.
pop_size = (sol_per_pop,num_weights) # The population will have sol_per_pop chromosome where each chromosome has num_weights genes.
#Creating the initial population.
new_population = numpy.random.uniform(low=-4.0, high=4.0, size=pop_size)
print(new_population)
```

Evolutionary Optimization – Part II

```
num_generations = 5
for generation in range(num_generations):
    print("Generation : ", generation)
    # Measing the fitness of each chromosome in the population.
    fitness = GA.cal_pop_fitness(equation_inputs, new_population)

    # Selecting the best parents in the population for mating.
    parents = GA.select_mating_pool(new_population, fitness,
                                    num_parents_mating)

    # Generating next generation using crossover.
    offspring_crossover = GA.crossover(parents,
                                      offspring_size=(pop_size[0]-parents.shape[0], num_weights))

    # Adding some variations to the offsrping using mutation.
    offspring_mutation = GA.mutation(offspring_crossover)

    # Creating the new population based on the parents and offspring.
    new_population[0:parents.shape[0], :] = parents
    new_population[parents.shape[0]:, :] = offspring_mutation

    # The best result in the current iteration.
    print("Best result : ", numpy.max(numpy.sum(new_population*equation_inputs, axis=1)))

# Getting the best solution after iterating finishing all generations.
#At first, the fitness is calculated for each solution in the final generation.
fitness = GA.cal_pop_fitness(equation_inputs, new_population)
# Then return the index of that solution corresponding to the best fitness.
best_match_idx = numpy.where(fitness == numpy.max(fitness))

print("Best solution : ", new_population[best_match_idx, :])
print("Best solution fitness : ", fitness[best_match_idx])
```

Evolutionary Optimization

- Good methods for any problem, where we have no idea how to optimize some function
- However, cannot guarantee performance optimality!
- The quality of the results depends highly on:
  - The initial population
  - The genetic operators (crossover, selection, mutation) and whether they are well-suited for the problem you're solving
  - The probabilities of crossover and mutation

Python
Machine Learning
Hyperparameters

Integrated Master's in Informatics Engineering

Learning and Extraction of Knowledge

2018/2019

<u>Synthetic Intelligence Lab</u>

Filipe Gonçalves

César Analide