mlrose Documentation

Release 1.2.0

Genevieve Hayes

Mar 30, 2019

Contents

mlrose is a Python package for applying some of the most common randomized optimization and search algorithms to a range of different optimization problems, over both discrete- and continuous-valued parameter spaces.

The source code was written by Genevieve Hayes and is available on GitHub.

CHAPTER 1

User Guide

1.1 Overview

mlrose is a Python package for applying some of the most common randomized optimization and search algorithms to a range of different optimization problems, over both discrete- and continuous-valued parameter spaces.

1.1.1 Project Background

mlrose was initially developed to support students of Georgia Tech's OMSCS/OMSA offering of CS 7641: Machine Learning.

It includes implementations of all randomized optimization algorithms taught in this course, as well as functionality to apply these algorithms to integer-string optimization problems, such as N-Queens and the Knapsack problem; continuous-valued optimization problems, such as the neural network weight problem; and tour optimization problems, such as the Travelling Salesperson problem. It also has the flexibility to solve user-defined optimization problems.

At the time of development, there did not exist a single Python package that collected all of this functionality together in the one location.

1.1.2 Main Features

Randomized Optimization Algorithms

- Implementations of: hill climbing, randomized hill climbing, simulated annealing, genetic algorithm and (discrete) MIMIC;
- Solve both maximization and minimization problems;
- Define the algorithm's initial state or start from a random state;
- Define your own simulated annealing decay schedule or use one of three pre-defined, customizable decay schedules: geometric decay, arithmetic decay or exponential decay.

Problem Types

- Solve discrete-value (bit-string and integer-string), continuous-value and tour optimization (travelling salesperson) problems;
- Define your own fitness function for optimization or use a pre-defined function.
- Pre-defined fitness functions exist for solving the: One Max, Flip Flop, Four Peaks, Six Peaks, Continuous Peaks, Knapsack, Travelling Salesperson, N-Queens and Max-K Color optimization problems.

Machine Learning Weight Optimization

- Optimize the weights of neural networks, linear regression models and logistic regression models using randomized hill climbing, simulated annealing, the genetic algorithm or gradient descent;
- Supports classification and regression neural networks.

1.1.3 Installation

mlrose was written in Python 3 and requires NumPy, SciPy and Scikit-Learn (sklearn).

The latest released version is available at the Python package index and can be installed using pip:

```
pip install mlrose
```

1.1.4 Licensing, Authors, Acknowledgements

mlrose was written by Genevieve Hayes and is distributed under the 3-Clause BSD license. The source code is maintained in a GitHub repository.

You can cite mlrose in research publications and reports as follows:

- Hayes, G. (2019). *mlrose: Machine Learning, Randomized Optimization and SEarch package for Python*. https://github.com/gkhayes/mlrose. Accessed: *day month year*.

BibTeX entry:

```
@misc{Hayes19,
 author = {Hayes, G},
 title  = {{mlrose: Machine Learning, Randomized Optimization and SEarch package for␣
↪Python}},
 year   = 2019,
 howpublished = {\url{https://github.com/gkhayes/mlrose}},
 note   = {Accessed: day month year}
}
```

1.2 Tutorial - Getting Started

mlrose provides functionality for implementing some of the most popular randomization and search algorithms, and applying them to a range of different optimization problem domains.

In this tutorial, we will discuss what is meant by an optimization problem and step through an example of how mlrose can be used to solve them. It is assumed that you have already installed mlrose on your computer. If not, you can do so using the instructions provided *here*.

1.2.1 What is an Optimization Problem?

An optimization problem is defined by Russell and Norvig (2010) as a problem in which "the aim is to find the best state according to an objective function."

What is meant by a "state" depends on the context of the problem. Some examples of states are:

- the weights used in a machine learning model, such as a neural network;
- the placement of chess pieces on a chess board;
- the order that cities are visited in a tour of all cities on a map of a country;
- the colors selected to color the countries in a map of the world.

What is important, for our purposes, is that the state can be represented numerically, ideally as a one-dimensional array (or vector) of values.

What is meant by "best" is defined by a mathematical formula or function (known as an objective function, fitness function, cost function or loss function), which we want to either maximize or minimize. The function accepts a state array as an input and returns a "fitness" value as an output.

The output fitness values allow us to compare the inputted state to other states we might be considering. In this context, the elements of the state array can be thought of as the variables (or parameters) of the function.

Therefore, an optimization problem can be simply thought of as a mathematical function that we would like to maximize/minimize by selecting the optimal values for each of its parameters.

Example

The five-dimensional One-Max optimization problem involves finding the value of state vector $x = [x_0, x_1, x_2, x_3, x_4]$ which maximizes $Fitness(x) = x_0 + x_1 + x_2 + x_3 + x_4$.

If each of the elements of x can only take the values 0 or 1, then the solution to this problem is $x = [1, 1, 1, 1, 1]$. When x is set equal to this optimal value, $Fitness(x) = 5$, the maximum value it can take.

1.2.2 Why use Randomized Optimization?

For the One-Max example given above, even if the solution was not immediately obvious, it would be possible to calculate the fitness value for all possible state vectors, x, and then select the best of those vectors. However, for more complicated problems, this cannot always be done within a reasonable period of time. Randomized optimization overcomes this issue.

Randomized optimization algorithms typically start at an initial "best" state vector (or population of multiple state vectors) and then randomly generate a new state vector (often a neighbor of the current "best" state). If the new state is better than the current "best" state, then the new vector becomes the new "best" state vector.

This process is repeated until it is no longer possible to find a better state vector than the current "best" state, or if a better state vector cannot be found within a pre-specified number of attempts.

There is no guarantee a randomized optimization algorithm will find the optimal solution to a given optimization problem (for example, it is possible that the algorithm may find a local maximum of the fitness function, instead of the global maximum). However, if a sufficiently large number of attempts are made to find a better state at each step of the algorithm, then the algorithm will return a "good" solution to the problem.

There is a trade-off between the time spent searching for the optimal solution to an optimization problem and the quality of the solution ultimately found.

1.2.3 Solving Optimization Problems with mlrose

Solving an optimization problem using mlrose involves three simple steps:

1. Define a fitness function object.
2. Define an optimization problem object.
3. Select and run a randomized optimization algorithm.

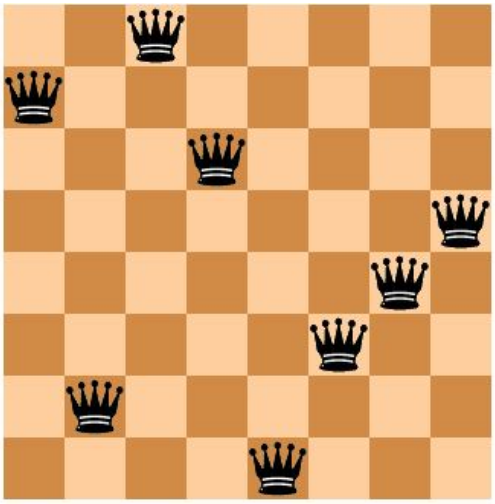
To illustrate each of these steps, in the next few sections we will work through the example of the 8-Queens optimization problem, described below:

Example: 8-Queens

In chess, the queen is the most powerful piece on the board. It can attack any piece in the same row, column or diagonal. In the 8-Queens problem, you are given a chessboard with eight queens (and no other pieces) and the aim is to place the queens on the board so that none of them can attack each other (Russell and Norvig (2010).

Clearly, in an optimal solution to this problem, there will be exactly one queen in each column. So, we only need to determine the row position of each queen, and we can define the state vector for this problem as $x = [x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7]$, where x_i denotes the row position of the queen in column i (for i = 0, 1, ..., 7).

The chessboard pictured below could, therefore, be described by the state vector $x = [6, 1, 7, 5, 0, 2, 3, 4]$, where the bottom left corner of the chessboard is assumed to be in column 0 and row 0.



This is not an optimal solution to the 8-Queens problem, since the three queens in columns 5, 6 and 7 are attacking each other diagonally, as are the queens in columns 2 and 6.

Before starting with this example, you will need to import the mlrose and Numpy Python packages.

```
import mlrose
import numpy as np
```

1.2.4 Define a Fitness Function Object

The first step in solving any optimization problem is to define the fitness function. This is the function we would ultimately like to maximize or minimize, and which can be used to evaluate the fitness of a given state vector, x.

In the context of the 8-Queens problem, our goal is to find a state vector for which no pairs of attacking queens exist. Therefore, we could define our fitness function as evaluating the number of pairs of attacking queens for a given state and try to minimize this function.

mlrose includes pre-defined fitness function classes for a range of common optimization problems, including the N-Queens family of problems (of which 8-Queens is a member). A list of the pre-defined fitness functions can be found *here*. The pre-defined `Queens()` class includes an implementation of the (8-)Queens fitness function described above.

We can initialize a fitness function object for this class, as follows:

```
fitness = mlrose.Queens()
```

Alternatively, we could look at the 8-Queens problem as one where the aim is to find a state vector for which all pairs of queens do *not* attack each other. In this context, we could define our fitness function as evaluating the number of pairs of *non-attacking* queens for a given state and try to maximize this function.

This definition of the 8-Queens fitness function is different from that used by mlrose's pre-defined `Queens()` class, so to use it, we will need to create a custom fitness function. This can be done by first defining a fitness function with a signature of the form `fitness_fn(state, **kwargs)`, and then using mlrose's `CustomFitness()` class to create a fitness function object, as follows:

```
# Define alternative N-Queens fitness function for maximization problem
def queens_max(state):

   # Initialize counter
       fitness_cnt = 0

          # For all pairs of queens
          for i in range(len(state) - 1):
                 for j in range(i + 1, len(state)):

                # Check for horizontal, diagonal-up and diagonal-down attacks
                if (state[j] != state[i]) \
                    and (state[j] != state[i] + (j - i)) \
                    and (state[j] != state[i] - (j - i)):

                   # If no attacks, then increment counter
                   fitness_cnt += 1

       return fitness_cnt

# Initialize custom fitness function object
fitness_cust = mlrose.CustomFitness(queens_max)
```

1.2.5 Define an Optimization Problem Object

Once we have created a fitness function object, we can use it as an input into an optimization problem object. In mlrose, optimization problem objects are used to contain all of the important information about the optimization problem we are trying to solve. mlrose provides classes for defining three types of optimization problem objects:

- `DiscreteOpt()`: This is used to describe discrete-state optimization problems. A discrete-state optimization problem is one where each element of the state vector can only take on a discrete set of values. In mlrose, these values are assumed to be integers in the range 0 to (max_val - 1), where max_val is defined at initialization.
- `ContinuousOpt()`: This is used to describe continuous-state optimization problems. Continuous-state optimization problems are similar to discrete-state optimization problems, except that each value in the state vector can take any value in the continuous range between min_val and max_val, as specified at initialization.
- `TSPOpt()`: This is used to describe travelling salesperson (or tour) optimization problems. Travelling salesperson optimization problems differ from the previous two problem types in that, we know the elements of the

optimal state vector are the integers 0 to (n - 1), where n is the length of the state vector, and our goal is to find the optimal ordering of those integers. We provide a worked example of this problem type *here*, so will not discuss it further for now.

The 8-Queens problem is an example of a discrete-state optimization problem, since each of the elements of the state vector must take on an integer value in the range 0 to 7.

To initialize a discrete-state optimization problem object, it is necessary to specify the problem length (i.e. the length of the state vector, which is 8 in this case); max_val, as defined above (also 8); the fitness function object created in the previous step; and whether the problem is a maximization or minimization problem.

For this example, we will use the first of the two fitness function objects defined above, so we want to solve a minimization problem.

```
problem = mlrose.DiscreteOpt(length = 8, fitness_fn = fitness, maximize = False, max_
↪val = 8)
```

However, had we chosen to use the second (custom) fitness function object, we would be dealing with a maximization problem, so, in the above code, we would have to set the `maximize` parameter to `True` instead of `False` (in addition to changing the value of the `fitness_fn` parameter).

1.2.6 Select and Run a Randomized Optimization Algorithm

Now that we have defined an optimization problem object, we are ready to solve our optimization problem. mlrose includes implementations of the (random-restart) hill climbing, randomized hill climbing (also known as stochastic hill climbing), simulated annealing, genetic algorithm and MIMIC (Mutual-Information-Maximizing Input Clustering) randomized optimization algorithms (references to each of these algorithms can be found *here*).

For discrete-state and travelling salesperson optimization problems, we can choose any of these algorithms. However, continuous-state problems are not supported in the case of MIMIC.

For our example, suppose we wish to use simulated annealing. To implement this algorithm, in addition to defining an optimization problem object, we must also define a schedule object (to specify how the simulated annealing temperature parameter changes over time); the number of attempts the algorithm should make to find a "better" state at each step (max_attempts); and the maximum number of iterations the algorithm should run for overall (max_iters). We can also specify the starting state for the algorithm, if desired (init_state).

To specify the schedule object, mlrose includes pre-defined decay schedule classes for geometric, arithmetic and expontential decay, as well as a class for defining your own decay schedule in a manner similar to the way in which we created a customized fitness function object. These classes are defined *here*.

Suppose we wish to use an exponential decay schedule (with default parameter settings); make at most 10 attempts to find a "better" state at each algorithm step; limit ourselves to at most 1000 iterations of the algorithm; and start at an initial state of $x = [0, 1, 2, 3, 4, 5, 6, 7]$. This can be done using the following code.

The algorithm returns the best state it can find, given the parameter values it has been provided, as well as the fitness value for that state.

```
# Define decay schedule
schedule = mlrose.ExpDecay()

# Define initial state
init_state = np.array([0, 1, 2, 3, 4, 5, 6, 7])

# Solve problem using simulated annealing
best_state, best_fitness = mlrose.simulated_annealing(problem, schedule = schedule,
                                                      max_attempts = 10, max_iters =␣
↪1000,
```

(continues on next page)

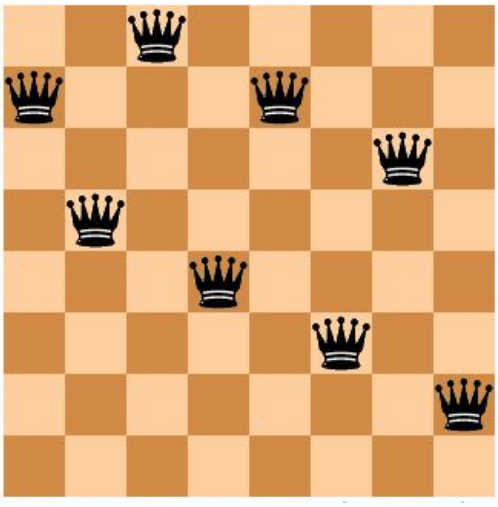
(continued from previous page)

```
                                                    init_state = init_state, random_
↪state = 1)

print(best_state)
[6 4 7 3 6 2 5 1]

print(best_fitness)
2.0
```

Running this code gives us a good solution to the 8-Queens problem, but not the optimal solution. The solution found by the algorithm, is pictured below:



The solution state has a fitness value of 2, indicating there are still two pairs of attacking queens on the chessboard (the queens in columns 0 and 3; and the two queens in row 6). Ideally, we would like our solution to have a fitness value of 0.

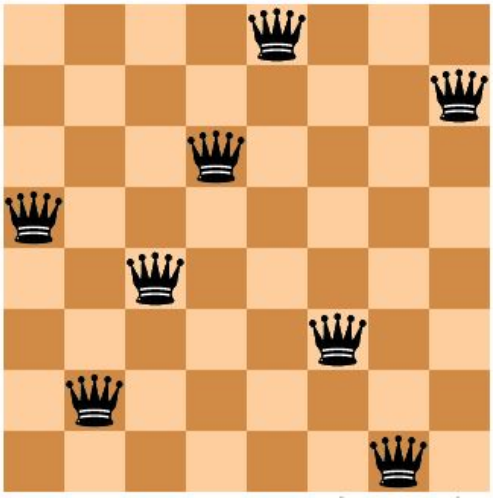
We can try to improve on our solution by tuning the parameters of our algorithm. Any of the algorithm's parameters can be tuned. However, in this case, let's focus on tuning the max_attempts parameter only, and increase it from 10 to 100.

```
# Solve problem using simulated annealing
best_state, best_fitness = mlrose.simulated_annealing(problem, schedule = schedule,
                                                      max_attempts = 100, max_iters =␣
↪1000,
                                                      init_state = init_state, random_
↪state = 1)

print(best_state)
[4 1 3 5 7 2 0 6]

print(best_fitness)
0.0
```

This time when we run our code, we get a solution with a fitness value of 0, indicating that none of the queens on the chessboard are attacking each other. This can be verified below:



1.2.7 Summary

In this tutorial we defined what is meant by an optimization problem and went through a simple example of how mlrose can be used to solve them. This is all you need to solve the majority of optimization problems. However, there is one type of problem we have only briefly touched upon so far: the travelling salesperson optimization problem. In the next tutorial we will go through an example of how mlrose can be used to solve this problem type.

1.2.8 References

Brownlee, J (2011). *Clever Algorithms: Nature-Inspired Programming Recipes*. http://www.cleveralgorithms.com.

De Bonet, J., C. Isbell, and P. Viola (1997). MIMIC: Finding Optima by Estimating Probability Densities. In *Advances in Neural Information Processing Systems* (NIPS) 9, pp. 424–430.

Russell, S. and P. Norvig (2010). *Artificial Intelligence: A Modern Approach*, 3rd edition. Prentice Hall, New Jersey, USA.

1.3 Tutorial - Travelling Saleperson Problems

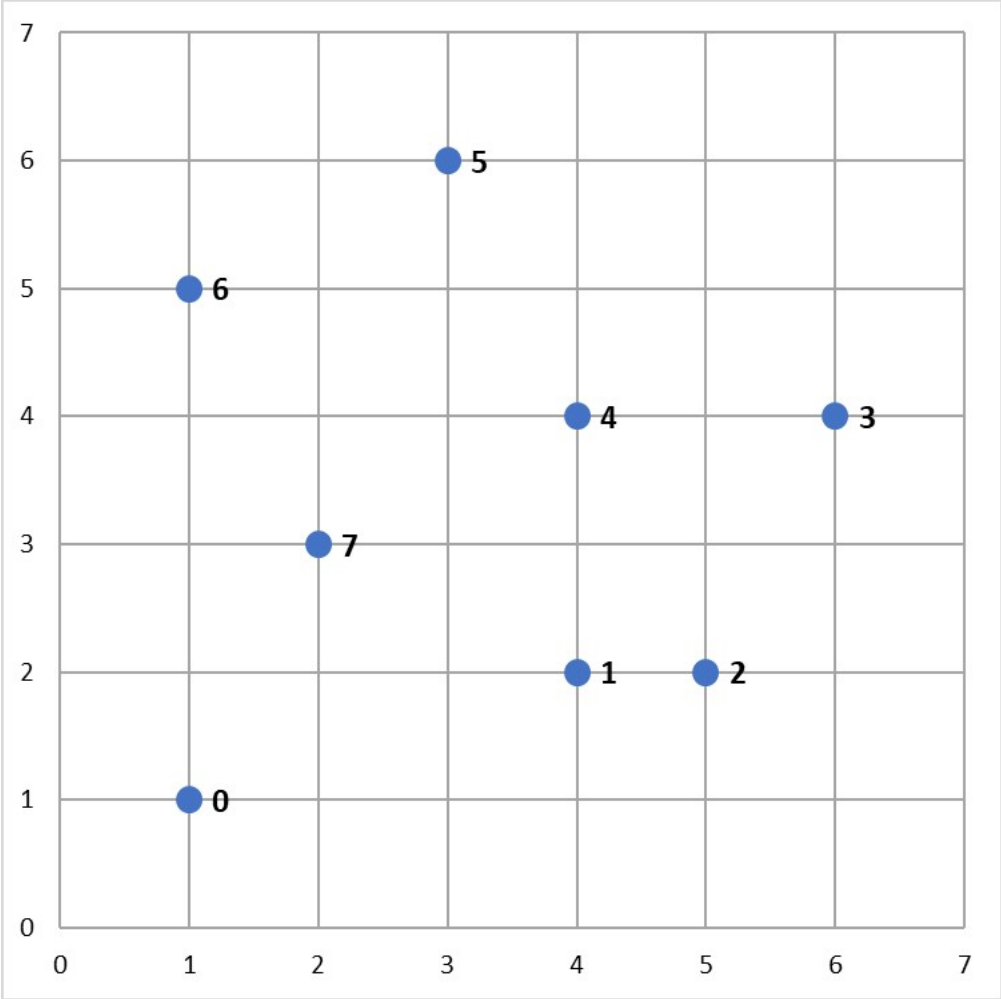
1.3.1 What is a Travelling Salesperson Problem?

The travelling salesperson problem (TSP) is a classic optimization problem where the goal is to determine the shortest tour of a collection of n "cities" (i.e. nodes), starting and ending in the same city and visiting all of the other cities exactly once. In such a situation, a solution can be represented by a vector of n integers, each in the range 0 to n-1, specifying the order in which the cities should be visited.

TSP is an NP-hard problem, meaning that, for larger values of n, it is not feasible to evaluate every possible problem solution within a reasonable period of time. Consequently, TSPs are well suited to solving using randomized optimization algorithms.

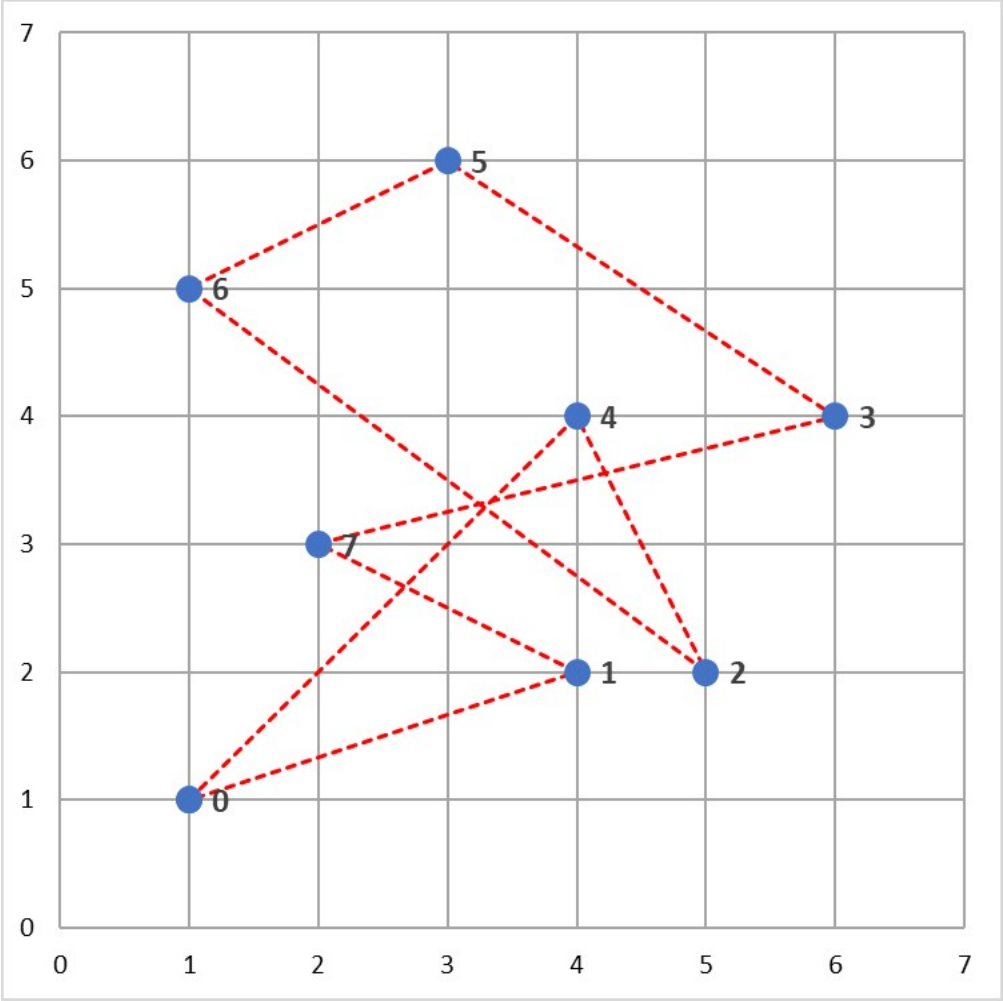
Example

Consider the following map containing 8 cities, numbered 0 to 7.



A salesperson would like to travel to each of these cities, starting and ending in the same city and visiting each of the other cities exactly once.

One possible tour of the cities is illustrated below, and could be represented by the solution vector $x = [0, 4, 2, 6, 5, 3, 7, 1]$ (assuming the tour starts and ends at City 0).



However, this is not the shortest tour of these cities. The aim of this problem is to find the *shortest* tour of the 8 cities.

1.3.2 Solving TSPs with mlrose

Given the solution to the TSP can be represented by a vector of integers in the range 0 to n-1, we could define a discrete-state optimization problem object and use one of mlrose's randomized optimization algorithms to solve it, as we did for the 8-Queens problem in the previous tutorial. However, by defining the problem this way, we would end up potentially considering invalid "solutions", which involve us visiting some cities more than once and some not at all.

An alternative is to define an optimization problem object that only allows us to consider valid tours of the n cities as potential solutions. This is a much more efficient approach to solving TSPs and can be implemented in mlrose using the `TSPOpt()` optimization problem class.

In this tutorial, we will use this alternative approach to solve the TSP example given above.

The steps required to solve this problem are the same as those used to solve any optimization problem in mlrose. Specifically:

1. Define a fitness function object.
2. Define an optimization problem object.
3. Select and run a randomized optimization algorithm.

Before starting with the example, you will need to import the mlrose and Numpy Python packages.

```
import mlrose
import numpy as np
```

1.3.3 Define a Fitness Function Object

For the TSP in the example, the goal is to find the shortest tour of the eight cities. As a result, the fitness function should calculate the total length of a given tour. This is the fitness definition used in mlrose's pre-defined `TravellingSales()` class.

The `TSPOpt()` optimization problem class assumes, by default, that the `TravellingSales()` class is used to define the fitness function for a TSP. As a result, if the `TravellingSales()` class is to be used to define the fitness function object, then this step can be skipped. However, it is also possible to manually define the fitness function object, if so desired.

To initialize a fitness function object for the `TravellingSales()` class, it is necessary to specify either the (x, y) coordinates of all the cities or the distances between each pair of cities for which travel is possible. If the former is specified, then it is assumed that travel between each pair of cities is possible.

If we choose to specify the coordinates, then these should be input as an ordered list of pairs (where pair i specifies the coordinates of city i), as follows:

```
# Create list of city coordinates
coords_list = [(1, 1), (4, 2), (5, 2), (6, 4), (4, 4), (3, 6), (1, 5), (2, 3)]

# Initialize fitness function object using coords_list
fitness_coords = mlrose.TravellingSales(coords = coords_list)
```

Alternatively, if we choose to specity the distances, then these should be input as a list of triples giving the distances, d, between all pairs of cities, u and v, for which travel is possible, with each triple in the form (u, v, d). The order in which the cities is specified does not matter (i.e., the distance between cities 1 and 2 is assumed to be the same as the distance between cities 2 and 1), and so each pair of cities need only be included in the list once.

Using the distance approach, the fitness function object can be initialize as follows:

```
# Create list of distances between pairs of cities
dist_list = [(0, 1, 3.1623), (0, 2, 4.1231), (0, 3, 5.8310), (0, 4, 4.2426), \
             (0, 5, 5.3852), (0, 6, 4.0000), (0, 7, 2.2361), (1, 2, 1.0000), \
             (1, 3, 2.8284), (1, 4, 2.0000), (1, 5, 4.1231), (1, 6, 4.2426), \
             (1, 7, 2.2361), (2, 3, 2.2361), (2, 4, 2.2361), (2, 5, 4.4721), \
             (2, 6, 5.0000), (2, 7, 3.1623), (3, 4, 2.0000), (3, 5, 3.6056), \
             (3, 6, 5.0990), (3, 7, 4.1231), (4, 5, 2.2361), (4, 6, 3.1623), \
             (4, 7, 2.2361), (5, 6, 2.2361), (5, 7, 3.1623), (6, 7, 2.2361)]

# Initialize fitness function object using dist_list
fitness_dists = mlrose.TravellingSales(distances = dist_list)
```

If both a list of coordinates and a list of distances are specified in initializing the fitness function object, then the distance list will be ignored.

1.3.4 Define an Optimization Problem Object

As mentioned previously, the most efficient approach to solving a TSP in mlrose is to define the optimization problem object using the `TSPOpt()` optimization problem class.

If a fitness function has already been manually defined, as demonstrated in the previous step, then the only additional information required to initialize a `TSPOpt()` object are the length of the problem (i.e. the number of cities to be visited on the tour) and whether our problem is a maximization or a minimization problem.

In our example, we want to solve a minimization problem of length 8. If we use the `fitness_coords` fitness function defined above, we can define an optimization problem object as follows:

```
# Define optimization problem object
problem_fit = mlrose.TSPOpt(length = 8, fitness_fn = fitness_coords, maximize=False)
```

Alternatively, if we had not previously defined a fitness function (and we wish to use the `TravellingSales()` class to define the fitness function), then this can be done as part of the optimization problem object initialization step by specifying either a list of coordinates or a list of distances, instead of a fitness function object, similar to what was done when manually initializing the fitness function object.

In the case of our example, if we choose to specify a list of coordinates, in place of a fitness function object, we can initialize our optimization problem object as:

```
# Create list of city coordinates
coords_list = [(1, 1), (4, 2), (5, 2), (6, 4), (4, 4), (3, 6), (1, 5), (2, 3)]

# Define optimization problem object
problem_no_fit = mlrose.TSPOpt(length = 8, coords = coords_list, maximize=False)
```

As with manually defining the fitness function object, if both a list of coordinates and a list of distances are specified in initializing the optimization problem object, then the distance list will be ignored. Furthermore, if a fitness function object is specified in addition to a list of coordinates and/or a list of distances, then the list of coordinates/distances will be ignored.

1.3.5 Select and Run a Randomized Optimization Algorithm

Once the optimization object is defined, all that is left to do is to select a randomized optimization algorithm and use it to solve our problem.

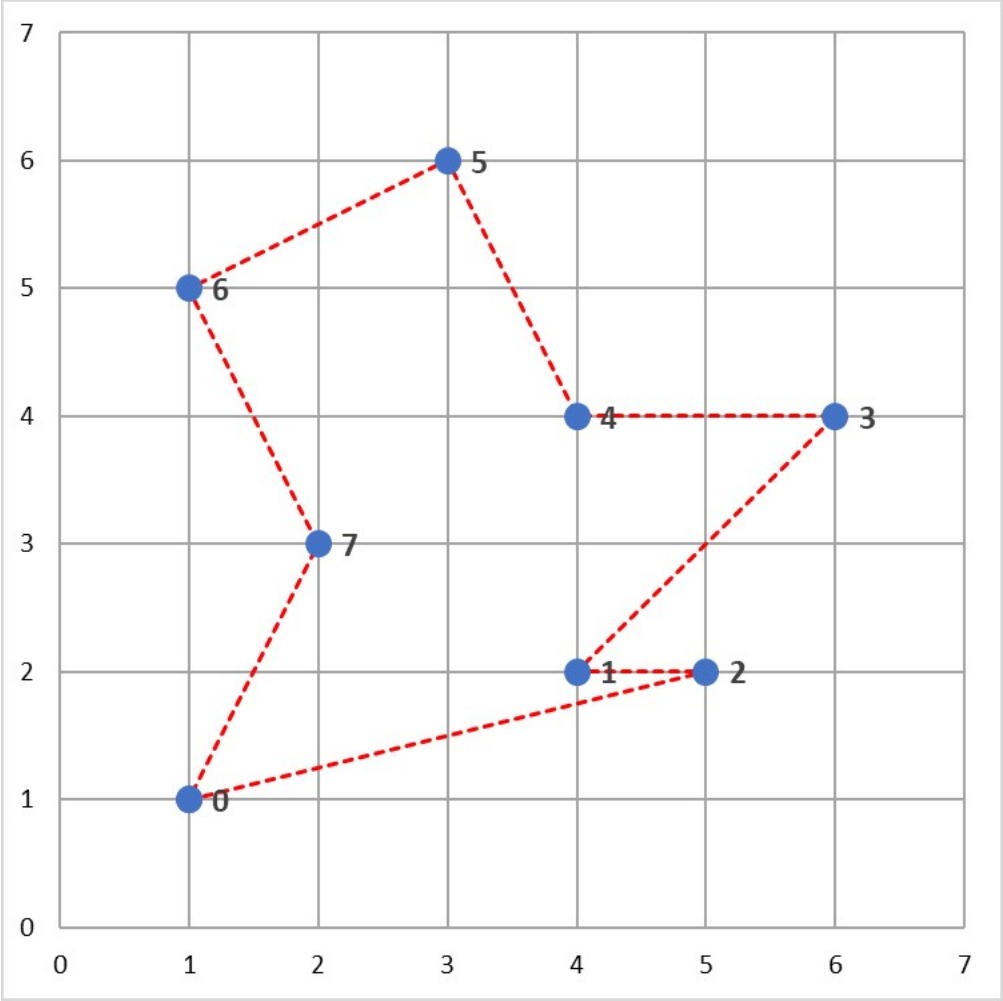
This time, suppose we wish to use the genetic algorithms with the default parameter settings of a population size (pop_size) of 200, a mutation probability (mutation_prob) of 0.1, a maximum of 10 attempts per step (max_attempts) and no limit on the maximum total number of iteration of the algorithm (max_iters). This returns the following solution:

```
# Solve problem using the genetic algorithm
best_state, best_fitness = mlrose.genetic_alg(problem_fit, random_state = 2)

print(best_state)
[1 3 4 5 6 7 0 2]

print(best_fitness)
18.8958046604
```

The solution tour found by the algorithm is pictured below and has a total length of 18.896 units.



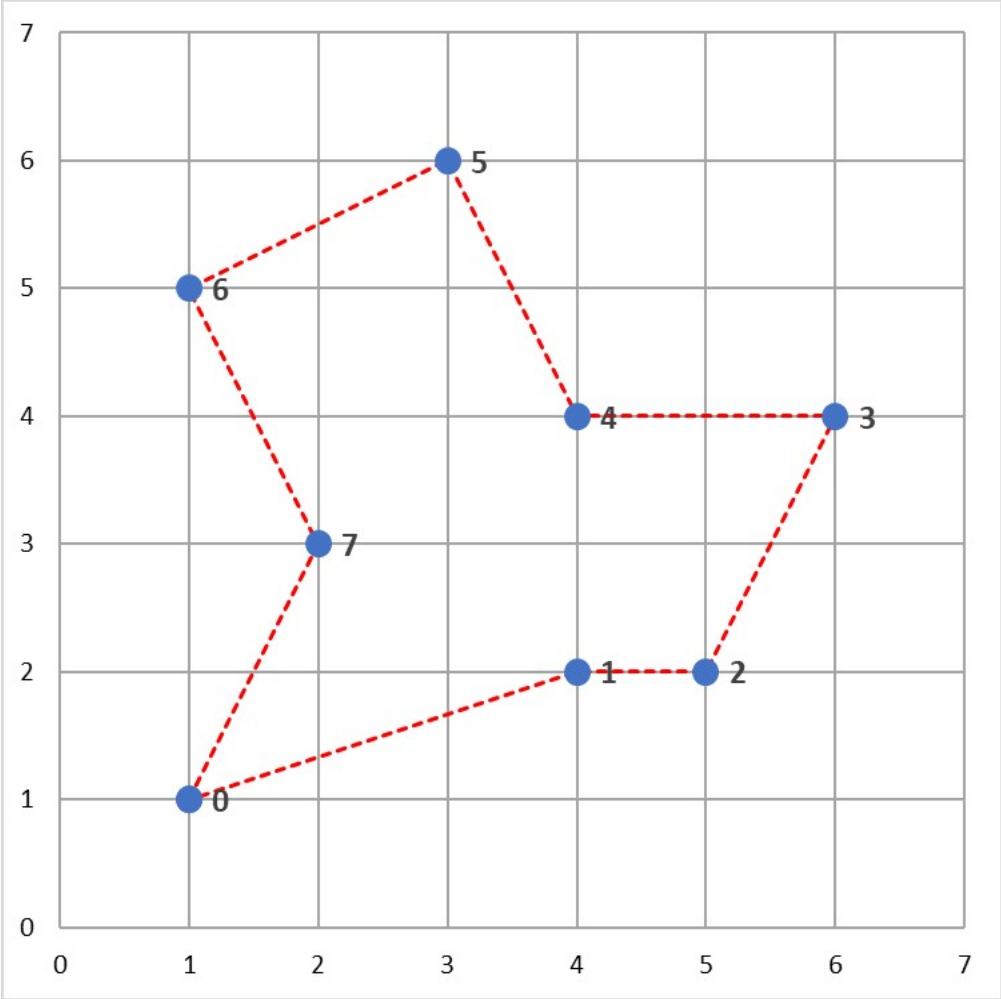
As in the 8-Queens example given in the previous tutorial, this solution can potentially be improved on by tuning the parameters of the optimization algorithm. For example, increasing the maximum number of attempts per step to 100 and increasing the mutation probability to 0.2, yields a tour with a total length of 17.343 units.

```
# Solve problem using the genetic algorithm
best_state, best_fitness = mlrose.genetic_alg(problem_fit, mutation_prob = 0.2,
                                              max_attempts = 100, random_state = 2)

print(best_state)
[7 6 5 4 3 2 1 0]

print(best_fitness)
17.3426175477
```

This solution is illustrated below and can be shown to be the optimal solution to this problem.



1.3.6 Summary

In this tutorial we introduced the travelling salesperson problem, and discussed how mlrose can be used to efficiently solve this problem. This is an example of how mlrose caters to solving one very specific type of optimization problem.

Another very specific type of optimization problem mlrose caters to solving is the machine learning weight optimization problem. That is, the problem of finding the optimal weights for machine learning models such as neural networks and regression models. We will discuss how mlrose can be used to solve this problem next, in our third and final tutorial.

1.4 Tutorial - Machine Learning Weight Optimization Problems

1.4.1 What is a Machine Learning Weight Optimization Problem?

For a number of different machine learning models, the process of fitting the model parameters involves finding the parameter values that minimize a pre-specified loss function for a given training dataset.

Examples of such models include neural networks, linear regression models and logistic regression models, and the optimal model weights for such models are typically found using methods such as gradient descent.

However, the problem of fitting the parameters (or weights) of a machine learning model can also be viewed as a continuous-state optimization problem, where the loss function takes the role of the fitness function, and the goal is to minimize this function.

By framing the problem this way, we can use any of the randomized optimization algorithms that are suited to continuous-state optimization problems to fit the model parameters. In this tutorial, we will work through an example of how this can be done with mlrose.

1.4.2 Solving Machine Learning Weight Optimization Problems with mlrose

mlrose contains built-in functionality for solving the weight optimization problem for three types of machine learning models: (standard) neural networks, linear regression models and logistic regression models. This is done using the `NeuralNetwork()`, `LinearRegression()` and `LogisticRegression()` classes respectively.

Each of these classes includes a `fit` method, which implements the three steps for solving an optimization problem defined in the previous tutorials, for a given training dataset.

However, when fitting a machine learning model, finding the optimal model weights is merely a means to an end. We want to find the optimal model weights so that we can use our fitted model to predict the labels of future observations as accurately as possible, not because we are actually interested in knowing the optimal weight values.

As a result, the abovementioned classes also include a `predict` method, which, if called after the `fit` method, will predict the labels for a given test dataset using the fitted model.

The steps involved in solving a machine learning weight optimization problem with mlrose are typically:

1. Initialize a machine learning weight optimization problem object.
2. Find the optimal model weights for a given training dataset by calling the `fit` method of the object initialized in step 1.
3. Predict the labels for a test dataset by calling the `predict` method of the object initialized in step 1.

To fit the model weights, the user can choose between using either randomized hill climbing, simulated annealing, the genetic algorithm or gradient descent. In mlrose, the gradient descent algorithm is only available for use in solving the machine learning weight optimization problem and has been included primarily for benchmarking purposes, since this is one of the most common algorithm used in fitting neural networks and regression models.

We will now work through an example to illustrate how mlrose can be used to fit a neural network and a regression model to a given dataset.

Example: the Iris Dataset

The Iris dataset is a famous multivariate classification dataset first presented in a 1936 research paper by statistician and biologist Ronald Fisher. It contains 150 observations of three classes (species) of iris flowers (50 observations of each class), with each observation providing the sepal length, sepal width, petal length and petal width (i.e. the feature values), as well as the class label (i.e. the target value), of each flower under consideration.

The Iris dataset is included with the Python sklearn package. The feature values and label of the first observation in the dataset are shown below, along with the maximum and minimum values of each of the features and the unique label values:

```
import numpy as np
from sklearn.datasets import load_iris

# Load the Iris dataset
data = load_iris()

# Get feature values
print(data.data[0])
```

(continues on next page)

(continued from previous page)

```
[ 5.1  3.5  1.4  0.2]

# Get feature names
print(data.feature_names)
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']

# Get target value of first observation
print(data.target[0])
0

# Get target name of first observation
print(data.target_names[data.target[0]])
setosa

# Get minimum feature values
print(np.min(data.data, axis = 0))
[ 4.3  2.   1.   0.1]

# Get maximum feature values
print(np.max(data.data, axis = 0))
[ 7.9  4.4  6.9  2.5]

# Get unique target values
print(np.unique(data.target))
[0 1 2]
```

From this we can see that all features in the Iris data set are numeric, albeit with different ranges, and that the class labels have been represented by integers.

In the next few sections we will show how mlrose can be used to fit a neural network and a logistic regression model to this dataset, to predict the species of an iris flower given its feature values.

1.4.3 Data Pre-Processing

Before we can fit any sort of machine learning model to a dataset, it is necessary to manipulate our data into the form expected by mlrose. Each of the three machine learning models supported by mlrose expect to receive feature data in the form of a numpy array, with one row per observation and numeric features only (any categorical features must be one-hot encoded before passing to the machine learning models).

The models also expect to receive the target values as either: a list of numeric values (for regression data); a list of 0-1 indicator values (for binary classification data); or as a numpy array of one-hot encoded labels, with one row per observation (for multi-class classification data).

In the case of the Iris dataset, all of our features are numeric, so no one-hot encoding is required. However, it is necessary to one-hot encode the class labels.

In keeping with standard machine learning practice, it is also necessary to split the data into training and test subsets, and since the range of the Iris data varies considerably from feature to feature, to standardize the values of our feature variables.

These pre-processing steps are implemented below.

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler, OneHotEncoder

# Split data into training and test sets
```

(continues on next page)

(continued from previous page)

```
X_train, X_test, y_train, y_test = train_test_split(data.data, data.target, \
                                                    test_size = 0.2, random_state = 3)

# Normalize feature data
scaler = MinMaxScaler()

X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# One hot encode target values
one_hot = OneHotEncoder()

y_train_hot = one_hot.fit_transform(y_train.reshape(-1, 1)).todense()
y_test_hot = one_hot.transform(y_test.reshape(-1, 1)).todense()
```

1.4.4 Neural Networks

Once the data has been preprocessed, fitting a neural network in mlrose simply involves following the steps listed above.

Suppose we wish to fit a neural network classifier to our Iris dataset with one hidden layer containing 2 nodes and a ReLU activation function (mlrose supports the ReLU, identity, sigmoid and tanh activation functions).

For this example, we will use the Randomized Hill Climbing algorithm to find the optimal weights, with a maximum of 1000 iterations of the algorithm and 100 attempts to find a better set of weights at each step. We will also include a bias term; use a step size (learning rate) of 0.0001; and limit our weights to being in the range -5 to 5 (to reduce the landscape over which the algorithm must search in order to find the optimal weights).

This model is initialized and fitted to our preprocessed data below:

```
# Initialize neural network object and fit object
nn_model1 = mlrose.NeuralNetwork(hidden_nodes = [2], activation = 'relu', \
                                 algorithm = 'random_hill_climb', max_iters = 1000, \
                                 bias = True, is_classifier = True, learning_rate = 0.
↪0001, \
                                 early_stopping = True, clip_max = 5, max_attempts =␣
↪100, \
                                 random_state = 3)

nn_model1.fit(X_train_scaled, y_train_hot)
```

Once the model is fitted, we can use it to predict the labels for our training and test datasets and use these prediction to assess the model's training and test accuracy.

```
from sklearn.metrics import accuracy_score

# Predict labels for train set and assess accuracy
y_train_pred = nn_model1.predict(X_train_scaled)

y_train_accuracy = accuracy_score(y_train_hot, y_train_pred)

print(y_train_accuracy)
0.45

# Predict labels for test set and assess accuracy
```

(continues on next page)

(continued from previous page)

```
y_test_pred = nn_model1.predict(X_test_scaled)

y_test_accuracy = accuracy_score(y_test_hot, y_test_pred)

print(y_test_accuracy)
0.533333333333
```

In this case, our model achieves training accuracy of 45% and test accuracy of 53.3%. These accuracy levels are better than if the labels were selected at random, but still leave room for improvement.

We can potentially improve on the accuracy of our model by tuning the parameters we set when initializing the neural network object. Suppose we decide to change the optimization algorithm to gradient descent, but leave all other model parameters unchanged.

```
# Initialize neural network object and fit object
nn_model2 = mlrose.NeuralNetwork(hidden_nodes = [2], activation = 'relu', \
                                 algorithm = 'gradient_descent', max_iters = 1000, \
                                 bias = True, is_classifier = True, learning_rate = 0.
↪0001, \
                                 early_stopping = True, clip_max = 5, max_attempts =␣
↪100, \
                                 random_state = 3)

nn_model2.fit(X_train_scaled, y_train_hot)

# Predict labels for train set and assess accuracy
y_train_pred = nn_model2.predict(X_train_scaled)

y_train_accuracy = accuracy_score(y_train_hot, y_train_pred)

print(y_train_accuracy)
0.625

# Predict labels for test set and assess accuracy
y_test_pred = nn_model2.predict(X_test_scaled)

y_test_accuracy = accuracy_score(y_test_hot, y_test_pred)

print(y_test_accuracy)
0.566666666667
```

This results in a 39% increase in training accuracy to 62.5%, but a much smaller increase in test accuracy to 56.7%.

1.4.5 Linear and Logistic Regression Models

Linear and logistic regression models are special cases of neural networks. A linear regression is a regression neural network with no hidden layers and an identity activation fuction, while a logistic regression is a classification neural network with no hidden layers and a sigmoid activation function. As a result, we could fit either of these models to our data using the `NeuralNetwork()` class with parameters set appropriately.

For example, suppose we wished to fit a logistic regression to our Iris data using the randomized hill climbing algorithm and all other parameters set as for the example in the previous section. We could do this by initializing a `NeuralNetwork()` object like so:

```
lr_nn_model1 = mlrose.NeuralNetwork(hidden_nodes = [], activation = 'sigmoid', \
                                    algorithm = 'random_hill_climb', max_iters = 1000,
↪ \
                                    bias = True, is_classifier = True, learning_rate␣
↪= 0.0001, \
                                    early_stopping = True, clip_max = 5, max_attempts␣
↪= 100, \
                                    random_state = 3)
```

However, for convenience, mlrose provides the `LinearRegression()` and `LogisticRegression()` wrapper classes, which simplify model initialization.

In our Iris dataset example, we can, thus, initialize and fit our logistic regression model as follows:

```
# Initialize logistic regression object and fit object
lr_model1 = mlrose.LogisticRegression(algorithm = 'random_hill_climb', max_iters =␣
↪1000, \
                                      bias = True, learning_rate = 0.0001, \
                                      early_stopping = True, clip_max = 5, max_
↪attempts = 100, \
                                      random_state = 3)

lr_model1.fit(X_train_scaled, y_train_hot)

# Predict labels for train set and assess accuracy
y_train_pred = lr_model1.predict(X_train_scaled)

y_train_accuracy = accuracy_score(y_train_hot, y_train_pred)

print(y_train_accuracy)
0.191666666667

# Predict labels for test set and assess accuracy
y_test_pred = lr_model1.predict(X_test_scaled)

y_test_accuracy = accuracy_score(y_test_hot, y_test_pred)

print(y_test_accuracy)
0.0666666666667
```

This model achieves 19.2% training accuracy and 6.7% test accuracy, which is worse than if we predicted the labels by selecting values at random.

Nevertheless, as in the previous section, we can potentially improve model accuracy by tuning the parameters set at initialization.

Suppose we increase our learning rate to 0.01.

```
# Initialize logistic regression object and fit object
lr_model2 = mlrose.LogisticRegression(algorithm = 'random_hill_climb', max_iters =␣
↪1000, \
                                      bias = True, learning_rate = 0.01, \
                                      early_stopping = True, clip_max = 5, max_
↪attempts = 100, \
                                      random_state = 3)

lr_model2.fit(X_train_scaled, y_train_hot)
```

(continues on next page)

(continued from previous page)

```
# Predict labels for train set and assess accuracy
y_train_pred = lr_model2.predict(X_train_scaled)

y_train_accuracy = accuracy_score(y_train_hot, y_train_pred)

print(y_train_accuracy)
0.683333333333

# Predict labels for test set and assess accuracy
y_test_pred = lr_model2.predict(X_test_scaled)

y_test_accuracy = accuracy_score(y_test_hot, y_test_pred)

print(y_test_accuracy)
0.7
```

This results in signficant improvements to both training and test accuracy, with training accuracy levels now reaching 68.3% and test accuracy levels reaching 70%.

1.4.6 Summary

In this tutorial we demonstrated how mlrose can be used to find the optimal weights of three types of machine learning models: neural networks, linear regression models and logistic regression models.

Applying randomized optimization algorithms to the machine learning weight optimization problem is most certainly not the most common approach to solving this problem. However, it serves to demonstrate the versatility of the mlrose package and of randomized optimization algorithms in general.

CHAPTER 2

API Reference

2.1 Algorithms

Functions to implement the randomized optimization and search algorithms.

hill_climb (*problem*, *max_iters=inf*, *restarts=0*, *init_state=None*, *curve=False*, *random_state=None*)

Use standard hill climbing to find the optimum for a given optimization problem.

Parameters

- **problem** (*optimization object*) – Object containing fitness function optimization problem to be solved. For example, `DiscreteOpt()`, `ContinuousOpt()` or `TSPOpt()`.
- **max_iters** (*int, default: np.inf*) – Maximum number of iterations of the algorithm for each restart.
- **restarts** (*int, default: 0*) – Number of random restarts.
- **init_state** (*array, default: None*) – 1-D Numpy array containing starting state for algorithm. If `None`, then a random state is used.
- **curve** (*bool, default: False*) – Boolean to keep fitness values for a curve. If `False`, then no curve is stored. If `True`, then a history of fitness values is provided as a third return value.
- **random_state** (*int, default: None*) – If random_state is a positive integer, random_state is the seed used by np.random.seed(); otherwise, the random seed is not set.

Returns

- **best_state** (*array*) – Numpy array containing state that optimizes the fitness function.
- **best_fitness** (*float*) – Value of fitness function at best state.
- **fitness_curve** (*array*) – Numpy array containing the fitness at every iteration. Only returned if input argument `curve` is `True`.

References

Russell, S. and P. Norvig (2010). *Artificial Intelligence: A Modern Approach*, 3rd edition. Prentice Hall, New Jersey, USA.

random_hill_climb(*problem*, *max_attempts=10*, *max_iters=inf*, *restarts=0*, *init_state=None*, *curve=False*, *random_state=None*)

Use randomized hill climbing to find the optimum for a given optimization problem.

Parameters

- **problem** (*optimization object*) – Object containing fitness function optimization problem to be solved. For example, `DiscreteOpt()`, `ContinuousOpt()` or `TSPOpt()`.
- **max_attempts** (*int, default: 10*) – Maximum number of attempts to find a better neighbor at each step.
- **max_iters** (*int, default: np.inf*) – Maximum number of iterations of the algorithm.
- **restarts** (*int, default: 0*) – Number of random restarts.
- **init_state** (*array, default: None*) – 1-D Numpy array containing starting state for algorithm. If `None`, then a random state is used.
- **curve** (*bool, default: False*) – Boolean to keep fitness values for a curve. If `False`, then no curve is stored. If `True`, then a history of fitness values is provided as a third return value.
- **random_state** (*int, default: None*) – If random_state is a positive integer, random_state is the seed used by np.random.seed(); otherwise, the random seed is not set.

Returns

- **best_state** (*array*) – Numpy array containing state that optimizes the fitness function.
- **best_fitness** (*float*) – Value of fitness function at best state.
- **fitness_curve** (*array*) – Numpy array containing the fitness at every iteration. Only returned if input argument `curve` is `True`.

References

Brownlee, J (2011). *Clever Algorithms: Nature-Inspired Programming Recipes*. http://www.cleveralgorithms.com.

simulated_annealing(*problem*, *schedule=<mlrose.decay.GeomDecay object>*, *max_attempts=10*, *max_iters=inf*, *init_state=None*, *curve=False*, *random_state=None*)

Use simulated annealing to find the optimum for a given optimization problem.

Parameters

- **problem** (*optimization object*) – Object containing fitness function optimization problem to be solved. For example, `DiscreteOpt()`, `ContinuousOpt()` or `TSPOpt()`.
- **schedule** (schedule object, default: `mlrose.GeomDecay()`) – Schedule used to determine the value of the temperature parameter.
- **max_attempts** (*int, default: 10*) – Maximum number of attempts to find a better neighbor at each step.
- **max_iters** (*int, default: np.inf*) – Maximum number of iterations of the algorithm.
- **init_state** (*array, default: None*) – 1-D Numpy array containing starting state for algorithm. If `None`, then a random state is used.

- **curve** (*bool, default: False*) – Boolean to keep fitness values for a curve. If `False`, then no curve is stored. If `True`, then a history of fitness values is provided as a third return value.
- **random_state** (*int, default: None*) – If random_state is a positive integer, random_state is the seed used by np.random.seed(); otherwise, the random seed is not set.

Returns

- **best_state** (*array*) – Numpy array containing state that optimizes the fitness function.
- **best_fitness** (*float*) – Value of fitness function at best state.
- **fitness_curve** (*array*) – Numpy array containing the fitness at every iteration. Only returned if input argument `curve` is `True`.

References

Russell, S. and P. Norvig (2010). *Artificial Intelligence: A Modern Approach*, 3rd edition. Prentice Hall, New Jersey, USA.

genetic_alg(*problem*, *pop_size=200*, *mutation_prob=0.1*, *max_attempts=10*, *max_iters=inf*, *curve=False*, *random_state=None*)

Use a standard genetic algorithm to find the optimum for a given optimization problem.

Parameters

- **problem** (*optimization object*) – Object containing fitness function optimization problem to be solved. For example, `DiscreteOpt()`, `ContinuousOpt()` or `TSPOpt()`.
- **pop_size** (*int, default: 200*) – Size of population to be used in genetic algorithm.
- **mutation_prob** (*float, default: 0.1*) – Probability of a mutation at each element of the state vector during reproduction, expressed as a value between 0 and 1.
- **max_attempts** (*int, default: 10*) – Maximum number of attempts to find a better state at each step.
- **max_iters** (*int, default: np.inf*) – Maximum number of iterations of the algorithm.
- **curve** (*bool, default: False*) – Boolean to keep fitness values for a curve. If `False`, then no curve is stored. If `True`, then a history of fitness values is provided as a third return value.
- **random_state** (*int, default: None*) – If random_state is a positive integer, random_state is the seed used by np.random.seed(); otherwise, the random seed is not set.

Returns

- **best_state** (*array*) – Numpy array containing state that optimizes the fitness function.
- **best_fitness** (*float*) – Value of fitness function at best state.
- **fitness_curve** (*array*) – Numpy array of arrays containing the fitness of the entire population at every iteration. Only returned if input argument `curve` is `True`.

References

Russell, S. and P. Norvig (2010). *Artificial Intelligence: A Modern Approach*, 3rd edition. Prentice Hall, New Jersey, USA.

mimic(*problem*, *pop_size=200*, *keep_pct=0.2*, *max_attempts=10*, *max_iters=inf*, *curve=False*, *random_state=None*)

Use MIMIC to find the optimum for a given optimization problem.

Parameters

- **problem** (*optimization object*) – Object containing fitness function optimization problem to be solved. For example, `DiscreteOpt()` or `TSPOpt()`.
- **pop_size** (*int, default: 200*) – Size of population to be used in algorithm.
- **keep_pct** (*float, default: 0.2*) – Proportion of samples to keep at each iteration of the algorithm, expressed as a value between 0 and 1.
- **max_attempts** (*int, default: 10*) – Maximum number of attempts to find a better neighbor at each step.
- **max_iters** (*int, default: np.inf*) – Maximum number of iterations of the algorithm.
- **curve** (*bool, default: False*) – Boolean to keep fitness values for a curve. If `False`, then no curve is stored. If `True`, then a history of fitness values is provided as a third return value.
- **random_state** (*int, default: None*) – If random_state is a positive integer, random_state is the seed used by np.random.seed(); otherwise, the random seed is not set.

Returns

- **best_state** (*array*) – Numpy array containing state that optimizes the fitness function.
- **best_fitness** (*float*) – Value of fitness function at best state.
- **fitness_curve** (*array*) – Numpy array containing the fitness at every iteration. Only returned if input argument `curve` is `True`.

References

De Bonet, J., C. Isbell, and P. Viola (1997). MIMIC: Finding Optima by Estimating Probability Densities. In *Advances in Neural Information Processing Systems* (NIPS) 9, pp. 424–430.

Note: MIMIC cannot be used for solving continuous-state optimization problems.

2.2 Decay Schedules

Classes for defining decay schedules for simulated annealing.

class GeomDecay(*init_temp=1.0*, *decay=0.99*, *min_temp=0.001*)

Schedule for geometrically decaying the simulated annealing temperature parameter T according to the formula:

$$T(t) = \max(T_0 \times r^t, T_{min})$$

where:

- T_0 is the initial temperature (at time t = 0);
- r is the rate of geometric decay; and
- T_{min} is the minimum temperature value.

Parameters

- **init_temp** (*float, default: 1.0*) – Initial value of temperature parameter T. Must be greater than 0.

- **decay** (*float, default: 0.99*) – Temperature decay parameter, r. Must be between 0 and 1.
- **min_temp** (*float, default: 0.001*) – Minimum value of temperature parameter. Must be greater than 0.

Example

```
>>> import mlrose
>>> schedule = mlrose.GeomDecay(init_temp=10, decay=0.95, min_temp=1)
>>> schedule.evaluate(5)
7.73780...
```

evaluate(*t*)

Evaluate the temperature parameter at time t.

Parameters **t** (*int*) – Time at which the temperature paramter T is evaluated.

Returns **temp** (*float*) – Temperature parameter at time t.

class ArithDecay(*init_temp=1.0*, *decay=0.0001*, *min_temp=0.001*)

Schedule for arithmetically decaying the simulated annealing temperature parameter T according to the formula:

$$T(t) = \max(T_0 - rt, T_{min})$$

where:

- T_0 is the initial temperature (at time t = 0);
- r is the rate of arithmetic decay; and
- T_{min} is the minimum temperature value.

Parameters

- **init_temp** (*float, default: 1.0*) – Initial value of temperature parameter T. Must be greater than 0.
- **decay** (*float, default: 0.0001*) – Temperature decay parameter, r. Must be greater than 0.
- **min_temp** (*float, default: 0.001*) – Minimum value of temperature parameter. Must be greater than 0.

Example

```
>>> import mlrose
>>> schedule = mlrose.ArithDecay(init_temp=10, decay=0.95, min_temp=1)
>>> schedule.evaluate(5)
5.25
```

evaluate(*t*)

Evaluate the temperature parameter at time t.

Parameters **t** (*int*) – Time at which the temperature paramter T is evaluated.

Returns **temp** (*float*) – Temperature parameter at time t.

class ExpDecay(*init_temp=1.0*, *exp_const=0.005*, *min_temp=0.001*)

Schedule for exponentially decaying the simulated annealing temperature parameter T according to the formula:

$$T(t) = \max(T_0 e^{-rt}, T_{min})$$

where:

- T_0 is the initial temperature (at time t = 0);
- r is the rate of exponential decay; and
- T_{min} is the minimum temperature value.

Parameters

- **init_temp** (*float, default: 1.0*) – Initial value of temperature parameter T. Must be greater than 0.
- **exp_const** (*float, default: 0.005*) – Exponential constant parameter, r. Must be greater than 0.
- **min_temp** (*float, default: 0.001*) – Minimum value of temperature parameter. Must be greater than 0.

Example

```
>>> import mlrose
>>> schedule = mlrose.ExpDecay(init_temp=10, exp_const=0.05, min_temp=1)
>>> schedule.evaluate(5)
7.78800...
```

evaluate(*t*)

Evaluate the temperature parameter at time t.

Parameters **t** (*int*) – Time at which the temperature paramter T is evaluated.

Returns **temp** (*float*) – Temperature parameter at time t.

class CustomSchedule(*schedule*, ***kwargs*)

Class for generating your own temperature schedule.

Parameters

- **schedule** (*callable*) – Function for calculating the temperature at time t with the signature `schedule(t, **kwargs)`.
- **kwargs** (*additional arguments*) – Additional parameters to be passed to schedule.

Example

```
>>> import mlrose
>>> def custom(t, c): return t + c
>>> kwargs = {'c': 10}
>>> schedule = mlrose.CustomSchedule(custom, **kwargs)
>>> schedule.evaluate(5)
15
```

evaluate(*t*)

Evaluate the temperature parameter at time t.

Parameters **t** (*int*) – Time at which the temperature paramter T is evaluated.

Returns **temp** (*float*) – Temperature parameter at time t.

2.3 Optimization Problem Types

Classes for defining optimization problem objects.

class DiscreteOpt(*length*, *fitness_fn*, *maximize=True*, *max_val=2*)
Class for defining discrete-state optimization problems.

Parameters

- **length** (*int*) – Number of elements in state vector.
- **fitness_fn** (*fitness function object*) – Object to implement fitness function for optimization.
- **maximize** (*bool, default: True*) – Whether to maximize the fitness function. Set `False` for minimization problem.
- **max_val** (*int, default: 2*) – Number of unique values that each element in the state vector can take. Assumes values are integers in the range 0 to (max_val - 1), inclusive.

class ContinuousOpt(*length*, *fitness_fn*, *maximize=True*, *min_val=0*, *max_val=1*, *step=0.1*)
Class for defining continuous-state optimisation problems.

Parameters

- **length** (*int*) – Number of elements in state vector.
- **fitness_fn** (*fitness function object*) – Object to implement fitness function for optimization.
- **maximize** (*bool, default: True*) – Whether to maximize the fitness function. Set `False` for minimization problem.
- **min_val** (*float, default: 0*) – Minimum value that each element of the state vector can take.
- **max_val** (*float, default: 1*) – Maximum value that each element of the state vector can take.
- **step** (*float, default: 0.1*) – Step size used in determining neighbors of current state.

class TSPOpt(*length*, *fitness_fn=None*, *maximize=False*, *coords=None*, *distances=None*)
Class for defining travelling salesperson optimisation problems.

Parameters

- **length** (*int*) – Number of elements in state vector. Must equal number of nodes in the tour.
- **fitness_fn** (*fitness function object, default: None*) – Object to implement fitness function for optimization. If `None`, then `TravellingSales(coords=coords, distances=distances)` is used by default.
- **maximize** (*bool, default: False*) – Whether to maximize the fitness function. Set `False` for minimization problem.
- **coords** (*list of pairs, default: None*) – Ordered list of the (x, y) co-ordinates of all nodes. This assumes that travel between all pairs of nodes is possible. If this is not the case, then use distances instead. This argument is ignored if fitness_fn is not `None`.
- **distances** (*list of triples, default: None*) – List giving the distances, d, between all pairs of nodes, u and v, for which travel is possible, with each list item in the form (u, v, d). Order of the nodes does not matter, so (u, v, d) and (v, u, d) are considered to be the same. If a pair is missing from the list, it is assumed that travel between the two nodes is not possible. This argument is ignored if fitness_fn or coords is not `None`.

2.4 Fitness Functions

Classes for defining fitness functions.

class OneMax

Fitness function for One Max optimization problem. Evaluates the fitness of an n-dimensional state vector $x = [x_0, x_1, \ldots, x_{n-1}]$ as:

$$Fitness(x) = \sum_{i=0}^{n-1} x_i$$

Example

```
>>> import mlrose
>>> import numpy as np
>>> fitness = mlrose.OneMax()
>>> state = np.array([0, 1, 0, 1, 1, 1, 1])
>>> fitness.evaluate(state)
5
```

Note: The One Max fitness function is suitable for use in either discrete or continuous-state optimization problems.

evaluate(*state*)

Evaluate the fitness of a state vector.

Parameters **state** (*array*) – State array for evaluation.

Returns **fitness** (*float*) – Value of fitness function.

class FlipFlop

Fitness function for Flip Flop optimization problem. Evaluates the fitness of a state vector x as the total number of pairs of consecutive elements of x, (x_i and x_{i+1}) where $x_i \neq x_{i+1}$.

Example

```
>>> import mlrose
>>> import numpy as np
>>> fitness = mlrose.FlipFlop()
>>> state = np.array([0, 1, 0, 1, 1, 1, 1])
>>> fitness.evaluate(state)
3
```

Note: The Flip Flop fitness function is suitable for use in discrete-state optimization problems *only*.

evaluate(*state*)

Evaluate the fitness of a state vector.

Parameters **state** (*array*) – State array for evaluation.

Returns **fitness** (*float*) – Value of fitness function.

class FourPeaks (*t_pct=0.1*)

Fitness function for Four Peaks optimization problem. Evaluates the fitness of an n-dimensional state vector x, given parameter T, as:

$$Fitness(x, T) = \max(tail(0, x), head(1, x)) + R(x, T)$$

where:

- $tail(b, x)$ is the number of trailing b's in x;
- $head(b, x)$ is the number of leading b's in x;
- $R(x, T) = n$, if $tail(0, x) > T$ and $head(1, x) > T$; and
- $R(x, T) = 0$, otherwise.

Parameters **t_pct** (*float, default: 0.1*) – Threshold parameter (T) for Four Peaks fitness function, expressed as a percentage of the state space dimension, n (i.e. $T = t_{pct} \times n$).

Example

```
>>> import mlrose
>>> import numpy as np
>>> fitness = mlrose.FourPeaks(t_pct=0.15)
>>> state = np.array([1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0])
>>> fitness.evaluate(state)
16
```

References

De Bonet, J., C. Isbell, and P. Viola (1997). MIMIC: Finding Optima by Estimating Probability Densities. In *Advances in Neural Information Processing Systems* (NIPS) 9, pp. 424–430.

Note: The Four Peaks fitness function is suitable for use in bit-string (discrete-state with `max_val = 2`) optimization problems *only*.

---

evaluate (*state*)

Evaluate the fitness of a state vector.

Parameters **state** (*array*) – State array for evaluation.

Returns **fitness** (*float.*) – Value of fitness function.

class SixPeaks (*t_pct=0.1*)

Fitness function for Six Peaks optimization problem. Evaluates the fitness of an n-dimensional state vector x, given parameter T, as:

$$Fitness(x, T) = \max(tail(0, x), head(1, x)) + R(x, T)$$

where:

- $tail(b, x)$ is the number of trailing b's in x;
- $head(b, x)$ is the number of leading b's in x;
- $R(x, T) = n$, if ($tail(0, x) > T$ and $head(1, x) > T$) or ($tail(1, x) > T$ and $head(0, x) > T$); and

- $R(x, T) = 0$, otherwise.

Parameters **t_pct** (*float, default: 0.1*) – Threshold parameter (T) for Six Peaks fitness function, expressed as a percentage of the state space dimension, n (i.e. $T = t_{pct} \times n$).

Example

```
>>> import mlrose
>>> import numpy as np
>>> fitness = mlrose.SixPeaks(t_pct=0.15)
>>> state = np.array([0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1])
>>> fitness.evaluate(state)
12
```

References

De Bonet, J., C. Isbell, and P. Viola (1997). MIMIC: Finding Optima by Estimating Probability Densities. In *Advances in Neural Information Processing Systems* (NIPS) 9, pp. 424–430.

Note: The Six Peaks fitness function is suitable for use in bit-string (discrete-state with `max_val = 2`) optimization problems *only*.

---

evaluate(*state*)

Evaluate the fitness of a state vector.

Parameters **state** (*array*) – State array for evaluation.

Returns **fitness** (*float*) – Value of fitness function.

class ContinuousPeaks(*t_pct=0.1*)

Fitness function for Continuous Peaks optimization problem. Evaluates the fitness of an n-dimensional state vector x, given parameter T, as:

$$Fitness(x, T) = \max(max\_run(0, x), max\_run(1, x)) + R(x, T)$$

where:

- $max\_run(b, x)$ is the length of the maximum run of b's in x;
- $R(x, T) = n$, if ($max\_run(0, x) > T$ and $max\_run(1, x) > T$); and
- $R(x, T) = 0$, otherwise.

Parameters **t_pct** (*float, default: 0.1*) – Threshold parameter (T) for Continuous Peaks fitness function, expressed as a percentage of the state space dimension, n (i.e. $T = t_{pct} \times n$).

Example

```
>>> import mlrose
>>> import numpy as np
>>> fitness = mlrose.ContinuousPeaks(t_pct=0.15)
>>> state = np.array([0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1])
>>> fitness.evaluate(state)
17
```

Note: The Continuous Peaks fitness function is suitable for use in bit-string (discrete-state with `max_val = 2`) optimization problems *only*.

evaluate(*state*)
Evaluate the fitness of a state vector.

Parameters **state** (*array*) – State array for evaluation.

Returns **fitness** (*float*) – Value of fitness function.

class Knapsack(*weights*, *values*, *max_weight_pct=0.35*)
Fitness function for Knapsack optimization problem. Given a set of n items, where item i has known weight w_i and known value v_i; and maximum knapsack capacity, W, the Knapsack fitness function evaluates the fitness of a state vector $x = [x_0, x_1, \ldots, x_{n-1}]$ as:

$$Fitness(x) = \sum_{i=0}^{n-1} v_i x_i, \text{ if } \sum_{i=0}^{n-1} w_i x_i \leq W, \text{ and 0, otherwise,}$$

where x_i denotes the number of copies of item i included in the knapsack.

Parameters

- **weights** (*list*) – List of weights for each of the n items.
- **values** (*list*) – List of values for each of the n items.
- **max_weight_pct** (*float, default: 0.35*) – Parameter used to set maximum capacity of knapsack (W) as a percentage of the total of the weights list (W = max_weight_pct $\times$ total_weight).

Example

```
>>> import mlrose
>>> import numpy as np
>>> weights = [10, 5, 2, 8, 15]
>>> values = [1, 2, 3, 4, 5]
>>> max_weight_pct = 0.6
>>> fitness = mlrose.Knapsack(weights, values, max_weight_pct)
>>> state = np.array([1, 0, 2, 1, 0])
>>> fitness.evaluate(state)
11
```

Note: The Knapsack fitness function is suitable for use in discrete-state optimization problems *only*.

evaluate(*state*)
Evaluate the fitness of a state vector.

Parameters **state** (*array*) – State array for evaluation. Must be the same length as the weights and values arrays.

Returns **fitness** (*float*) – Value of fitness function.

class TravellingSales(*coords=None*, *distances=None*)
Fitness function for Travelling Salesman optimization problem. Evaluates the fitness of a tour of n nodes, represented by state vector x, giving the order in which the nodes are visited, as the total distance travelled on the tour (including the distance travelled between the final node in the state vector and the first node in the state

vector during the return leg of the tour). Each node must be visited exactly once for a tour to be considered valid.

Parameters

- **coords** (*list of pairs, default: None*) – Ordered list of the (x, y) coordinates of all nodes (where element i gives the coordinates of node i). This assumes that travel between all pairs of nodes is possible. If this is not the case, then use `distances` instead.
- **distances** (*list of triples, default: None*) – List giving the distances, d, between all pairs of nodes, u and v, for which travel is possible, with each list item in the form (u, v, d). Order of the nodes does not matter, so (u, v, d) and (v, u, d) are considered to be the same. If a pair is missing from the list, it is assumed that travel between the two nodes is not possible. This argument is ignored if coords is not `None`.

Examples

```
>>> import mlrose
>>> import numpy as np
>>> coords = [(0, 0), (3, 0), (3, 2), (2, 4), (1, 3)]
>>> dists = [(0, 1, 3), (0, 2, 5), (0, 3, 1), (0, 4, 7), (1, 3, 6),
             (4, 1, 9), (2, 3, 8), (2, 4, 2), (3, 2, 8), (3, 4, 4)]
>>> fitness_coords = mlrose.TravellingSales(coords=coords)
>>> state = np.array([0, 1, 4, 3, 2])
>>> fitness_coords.evaluate(state)
13.86138...
>>> fitness_dists = mlrose.TravellingSales(distances=dists)
>>> fitness_dists.evaluate(state)
29
```

Note:

1. The TravellingSales fitness function is suitable for use in travelling salesperson (tsp) optimization problems *only*.
2. It is necessary to specify at least one of `coords` and `distances` in initializing a TravellingSales fitness function object.

`evaluate`(*state*)

Evaluate the fitness of a state vector.

Parameters **state** (*array*) – State array for evaluation. Each integer between 0 and (len(state) - 1), inclusive must appear exactly once in the array.

Returns **fitness** (*float*) – Value of fitness function. Returns `np.inf` if travel between two consecutive nodes on the tour is not possible.

`class Queens`

Fitness function for N-Queens optimization problem. Evaluates the fitness of an n-dimensional state vector $x = [x_0, x_1, \ldots, x_{n-1}]$, where x_i represents the row position (between 0 and n-1, inclusive) of the 'queen' in column i, as the number of pairs of attacking queens.

Example

```
>>> import mlrose
>>> import numpy as np
>>> fitness = mlrose.Queens()
>>> state = np.array([1, 4, 1, 3, 5, 5, 2, 7])
>>> fitness.evaluate(state)
6
```

References

Russell, S. and P. Norvig (2010). *Artificial Intelligence: A Modern Approach*, 3rd edition. Prentice Hall, New Jersey, USA.

Note: The Queens fitness function is suitable for use in discrete-state optimization problems *only*.

---

evaluate(*state*)

Evaluate the fitness of a state vector.

Parameters **state** (*array*) – State array for evaluation.

Returns **fitness** (*float*) – Value of fitness function.

class MaxKColor(*edges*)

Fitness function for Max-k color optimization problem. Evaluates the fitness of an n-dimensional state vector $x = [x_0, x_1, \ldots, x_{n-1}]$, where x_i represents the color of node i, as the number of pairs of adjacent nodes of the same color.

Parameters **edges** (*list of pairs*) – List of all pairs of connected nodes. Order does not matter, so (a, b) and (b, a) are considered to be the same.

Example

```
>>> import mlrose
>>> import numpy as np
>>> edges = [(0, 1), (0, 2), (0, 4), (1, 3), (2, 0), (2, 3), (3, 4)]
>>> fitness = mlrose.MaxKColor(edges)
>>> state = np.array([0, 1, 0, 1, 1])
>>> fitness.evaluate(state)
3
```

Note: The MaxKColor fitness function is suitable for use in discrete-state optimization problems *only*.

---

evaluate(*state*)

Evaluate the fitness of a state vector.

Parameters **state** (*array*) – State array for evaluation.

Returns **fitness** (*float*) – Value of fitness function.

class CustomFitness(*fitness_fn*, *problem_type='either'*, ***kwargs*)

Class for generating your own fitness function.

Parameters

- **fitness_fn** (*callable*) – Function for calculating fitness of a state with the signature `fitness_fn(state, **kwargs)`.
- **problem_type** (*string, default: 'either'*) – Specifies problem type as 'discrete', 'continuous', 'tsp' or 'either' (denoting either discrete or continuous).
- **kwargs** (*additional arguments*) – Additional parameters to be passed to the fitness function.

Example

```
>>> import mlrose
>>> import numpy as np
>>> def cust_fn(state, c): return c*np.sum(state)
>>> kwargs = {'c': 10}
>>> fitness = mlrose.CustomFitness(cust_fn, **kwargs)
>>> state = np.array([1, 2, 3, 4, 5])
>>> fitness.evaluate(state)
150
```

evaluate(*state*)

Evaluate the fitness of a state vector.

Parameters **state** (*array*) – State array for evaluation.

Returns **fitness** (*float*) – Value of fitness function.

2.5 Machine Learning Weight Optimization

Classes for defining neural network weight optimization problems.

class NeuralNetwork(*hidden_nodes=None, activation='relu', algorithm='random_hill_climb', max_iters=100, bias=True, is_classifier=True, learning_rate=0.1, early_stopping=False, clip_max=10000000000.0, restarts=0, schedule=<mlrose.decay.GeomDecay object>, pop_size=200, mutation_prob=0.1, max_attempts=10, random_state=None*)

Class for defining neural network classifier weights optimization problem.

Parameters

- **hidden_nodes** (*list of ints*) – List giving the number of nodes in each hidden layer.
- **activation** (*string, default: 'relu'*) – Activation function for each of the hidden layers. Must be one of: 'identity', 'relu', 'sigmoid' or 'tanh'.
- **algorithm** (*string, default: 'random_hill_climb'*) – Algorithm used to find optimal network weights. Must be one of:'random_hill_climb', 'simulated_annealing', 'genetic_alg' or 'gradient_descent'.
- **max_iters** (*int, default: 100*) – Maximum number of iterations used to fit the weights.
- **bias** (*bool, default: True*) – Whether to include a bias term.
- **is_classifer** (*bool, default: True*) – Whether the network is for classification or regression. Set `True` for classification and `False` for regression.
- **learning_rate** (*float, default: 0.1*) – Learning rate for gradient descent or step size for randomized optimization algorithms.

- **early_stopping** (*bool, default: False*) – Whether to terminate algorithm early if the loss is not improving. If `True`, then stop after max_attempts iters with no improvement.
- **clip_max** (*float, default: 1e+10*) – Used to limit weights to the range [-1*clip_max, clip_max].
- **restarts** (*int, default: 0*) – Number of random restarts. Only required if `algorithm = 'random_hill_climb'`.
- **schedule** (*schedule object, default = mlrose.GeomDecay()*) – Schedule used to determine the value of the temperature parameter. Only required if `algorithm = 'simulated_annealing'`.
- **pop_size** (*int, default: 200*) – Size of population. Only required if `algorithm = 'genetic_alg'`.
- **mutation_prob** (*float, default: 0.1*) – Probability of a mutation at each element of the state vector during reproduction, expressed as a value between 0 and 1. Only required if `algorithm = 'genetic_alg'`.
- **max_attempts** (*int, default: 10*) – Maximum number of attempts to find a better state. Only required if `early_stopping = True`.
- **random_state** (*int, default: None*) – If random_state is a positive integer, random_state is the seed used by np.random.seed(); otherwise, the random seed is not set.

Variables

- **`fitted_weights`** (*`array`*) – Numpy array giving the fitted weights when `fit` is performed.
- **`loss`** (*`float`*) – Value of loss function for fitted weights when `fit` is performed.
- **`predicted_probs`** (*`array`*) – Numpy array giving the predicted probabilities for each class when `predict` is performed for multi-class classification data; or the predicted probability for class 1 when `predict` is performed for binary classification data.

class LinearRegression(*algorithm='random_hill_climb', max_iters=100, bias=True, learning_rate=0.1, early_stopping=False, clip_max=10000000000.0, restarts=0, schedule=<mlrose.decay.GeomDecay object>, pop_size=200, mutation_prob=0.1, max_attempts=10, random_state=None*)

Class for defining linear regression weights optimization problem. Inherits `fit` and `predict` methods from `NeuralNetwork()` class.

Parameters

- **algorithm** (*string, default: 'random_hill_climb'*) – Algorithm used to find optimal network weights. Must be one of:'random_hill_climb', 'simulated_annealing', 'genetic_alg' or 'gradient_descent'.
- **max_iters** (*int, default: 100*) – Maximum number of iterations used to fit the weights.
- **bias** (*bool, default: True*) – Whether to include a bias term.
- **learning_rate** (*float, default: 0.1*) – Learning rate for gradient descent or step size for randomized optimization algorithms.
- **early_stopping** (*bool, default: False*) – Whether to terminate algorithm early if the loss is not improving. If `True`, then stop after max_attempts iters with no improvement.
- **clip_max** (*float, default: 1e+10*) – Used to limit weights to the range [-1*clip_max, clip_max].

- **restarts** (*int, default: 0*) – Number of random restarts. Only required if `algorithm = 'random_hill_climb'`.
- **schedule** (*schedule object, default = mlrose.GeomDecay()*) – Schedule used to determine the value of the temperature parameter. Only required if `algorithm = 'simulated_annealing'`.
- **pop_size** (*int, default: 200*) – Size of population. Only required if `algorithm = 'genetic_alg'`.
- **mutation_prob** (*float, default: 0.1*) – Probability of a mutation at each element of the state vector during reproduction, expressed as a value between 0 and 1. Only required if `algorithm = 'genetic_alg'`.
- **max_attempts** (*int, default: 10*) – Maximum number of attempts to find a better state. Only required if `early_stopping = True`.
- **random_state** (*int, default: None*) – If random_state is a positive integer, random_state is the seed used by np.random.seed(); otherwise, the random seed is not set.

Variables

- **`fitted_weights`** (*`array`*) – Numpy array giving the fitted weights when `fit` is performed.
- **`loss`** (*`float`*) – Value of loss function for fitted weights when `fit` is performed.

class LogisticRegression(*algorithm='random_hill_climb', max_iters=100, bias=True, learning_rate=0.1, early_stopping=False, clip_max=10000000000.0, restarts=0, schedule=<mlrose.decay.GeomDecay object>, pop_size=200, mutation_prob=0.1, max_attempts=10, random_state=None*)

Class for defining logistic regression weights optimization problem. Inherits `fit` and `predict` methods from `NeuralNetwork()` class.

Parameters

- **algorithm** (*string, default: 'random_hill_climb'*) – Algorithm used to find optimal network weights. Must be one of:'random_hill_climb', 'simulated_annealing', 'genetic_alg' or 'gradient_descent'.
- **max_iters** (*int, default: 100*) – Maximum number of iterations used to fit the weights.
- **bias** (*bool, default: True*) – Whether to include a bias term.
- **learning_rate** (*float, default: 0.1*) – Learning rate for gradient descent or step size for randomized optimization algorithms.
- **early_stopping** (*bool, default: False*) – Whether to terminate algorithm early if the loss is not improving. If `True`, then stop after max_attempts iters with no improvement.
- **clip_max** (*float, default: 1e+10*) – Used to limit weights to the range [-1*clip_max, clip_max].
- **restarts** (*int, default: 0*) – Number of random restarts. Only required if `algorithm = 'random_hill_climb'`.
- **schedule** (*schedule object, default = mlrose.GeomDecay()*) – Schedule used to determine the value of the temperature parameter. Only required if `algorithm = 'simulated_annealing'`.
- **pop_size** (*int, default: 200*) – Size of population. Only required if `algorithm = 'genetic_alg'`.

- **mutation_prob** (*float, default: 0.1*) – Probability of a mutation at each element of the state vector during reproduction, expressed as a value between 0 and 1. Only required if `algorithm = 'genetic_alg'`.
- **max_attempts** (*int, default: 10*) – Maximum number of attempts to find a better state. Only required if `early_stopping = True`.
- **random_state** (*int, default: None*) – If random_state is a positive integer, random_state is the seed used by np.random.seed(); otherwise, the random seed is not set.

Variables

- **`fitted_weights`** (*`array`*) – Numpy array giving the fitted weights when `fit` is performed.
- **`loss`** (*`float`*) – Value of loss function for fitted weights when `fit` is performed.

Python Module Index

m

Index