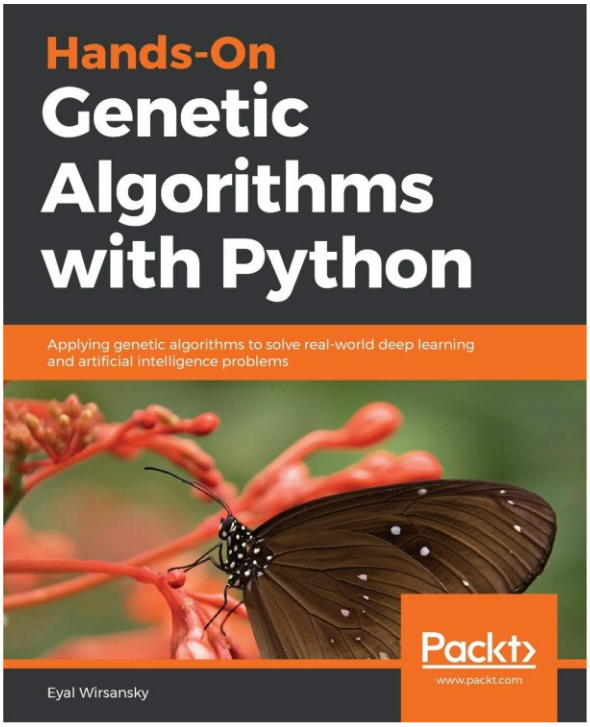
Hands-On
Genetic Algorithms with Python

Chapter 7: Enhancing Machine Learning Models Using Feature Selection



Enhancing Machine Learning Models Using Feature Selection

This chapter describes how genetic algorithms can be used to improve the performance of supervised machine learning models by selecting the best subset of features from the provided input data. This chapter will start with a brief introduction to machine learning and then describe the two main types of supervised machine learning tasks – regression and classification. We will then discuss the potential benefits of feature selection when it comes to the performance of these models. Next, we will demonstrate how genetic algorithms can be utilized to pinpoint the genuine features that are generated by the *Friedman-1 Test* regression problem. Then, we will use the real-life Zoo dataset to create a classification model and improve its accuracy – again by applying genetic algorithms to isolate the best features for the task.

In this chapter, we will cover the following topics:

- Understand the basic concepts of supervised machine learning, as well as regression and classification tasks
- Understand the benefits of feature selection on the performance of supervised learning models
- Enhance the performance of a regression model for the Friedman-1 Test regression problem, using feature selection carried out by a genetic algorithm coded with the DEAP framework
- Enhance the performance of a classification model for the Zoo dataset classification problem, using feature selection carried out by a genetic algorithm coded with the DEAP framework

We will start this chapter with a quick review of supervised machine learning. If you are a seasoned data scientist, feel free to skip the next section.

Technical requirements

In this chapter, we will be using Python 3 with the following supporting libraries:

- `deap`
- `numpy`
- `pandas`
- `matplotlib`
- `seaborn`
- `sklearn` – introduced in this chapter

In addition, we will be using the UCI *Zoo Dataset* (`https://archive.ics.uci.edu/ml/datasets/zoo`).

The programs that will be used in this chapter can be found in this book's GitHub repository at `https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/tree/master/Chapter07`.

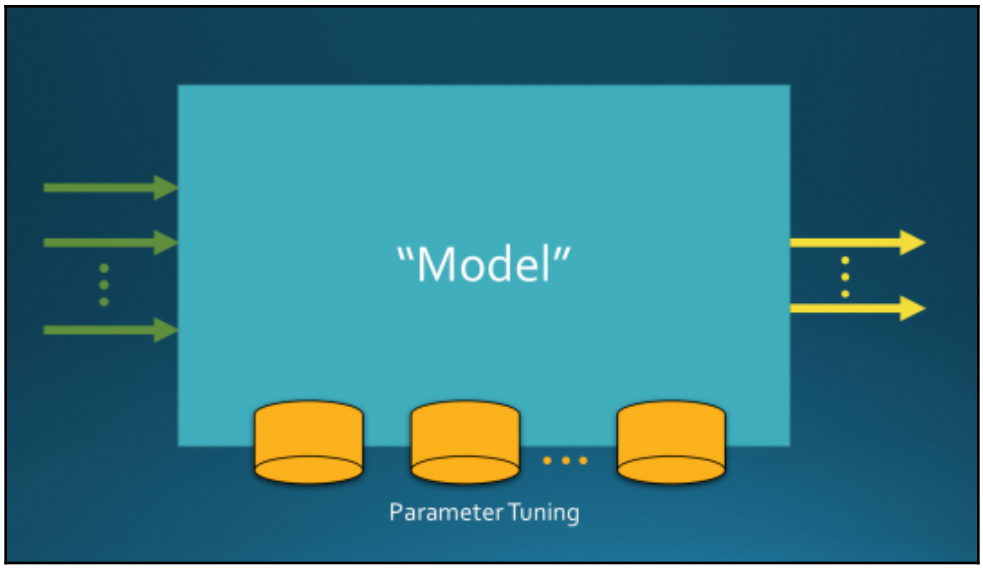
Check out the following video to see the Code in Action:
Placeholder link

Supervised machine learning

The term **machine learning** typically refers to a computer program that receives inputs and produces outputs. Our goal is to train this program, also known as the **model**, to produce the correct outputs for the given inputs, without explicitly programming them.

During this training process, the model learns the mapping between the inputs and the outputs by adjusting its internal parameters. One common way to train the model is by providing it with a set of inputs, for which the correct output is known. For each of these inputs, we tell the model what the correct output is so that it can adjust, or tune itself, aiming to eventually produce the desired output for each of the given inputs. This tuning is at the heart of the learning process.

Over the years, many types of machine learning models have been developed. Each model has its own particular internal parameters that can affect the mapping between the input and the output, and the values of these parameters can be tuned, as illustrated in the following image:



Parameter tuning of a machine learning model

For example, if the model was implementing a decision tree, it could contain several IF-THEN statements, which can be formulated as follows:

IF <input value> IS LESS THEN <some threshold value> THEN <go to some target branch>

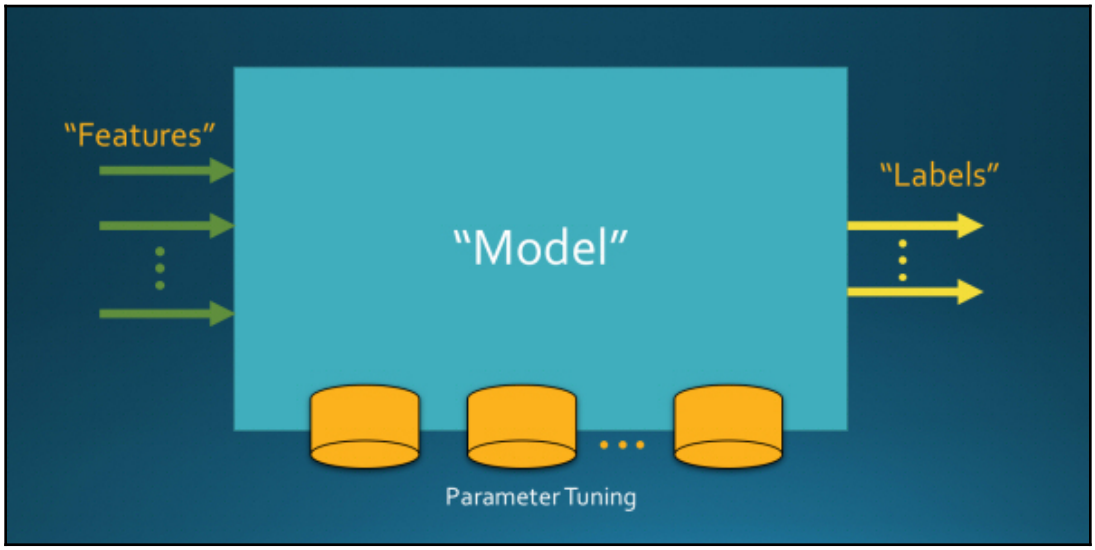
In this case, both the threshold value and the identity of the target branch are parameters that can be adjusted, or tuned, during the learning process.

To tune the internal parameters, each type of model has an accompanying **learning algorithm** that iterates over the given input and output values and seeks to match the given output for each of the given inputs. To accomplish this goal, a typical learning algorithm will measure the difference (or error) between the actual output and the desired output; the algorithm will then attempt to minimize this error by adjusting the model's internal parameters.

The two main types of supervised machine learning are **classification** and **regression**, and will be described in the following subsections.

Classification

When carrying out a classification task, the model needs to decide which category a certain input belongs to. Each category is represented by a single output (called a label), while the inputs are called **features**:

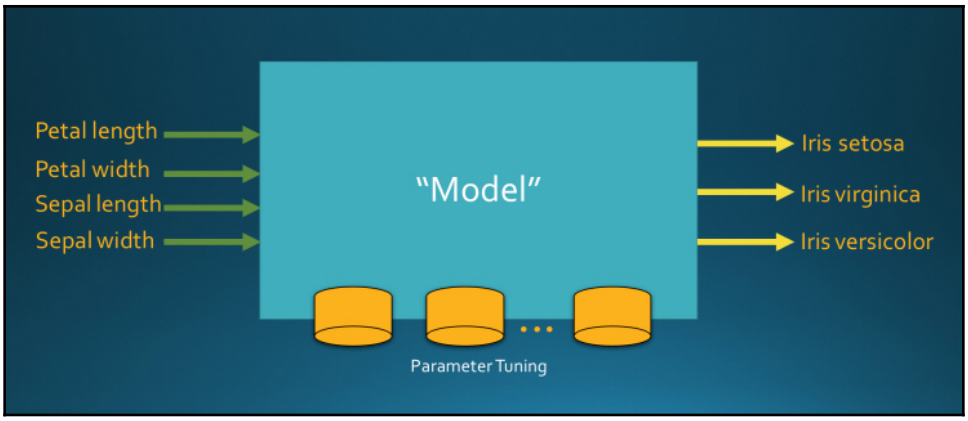


Machine learning classification model

For example, in the well-known Iris Flower dataset (`https://archive.ics.uci.edu/ml/datasets/Iris`), there are four features: **Petal length**, **Petal width**, **Sepal length**, and **Sepal width**. These represent the measurements that have been manually taken of actual Iris flowers.

In terms of the output, there are three labels: **Iris setosa**, **Iris virginica**, and **Iris versicolor**. These represent the three different types of Iris.

When the input values are present, which represent the measurements that were taken from a given Iris flower, we expect the output of the correct label to go **high** and the other two to go **low**:



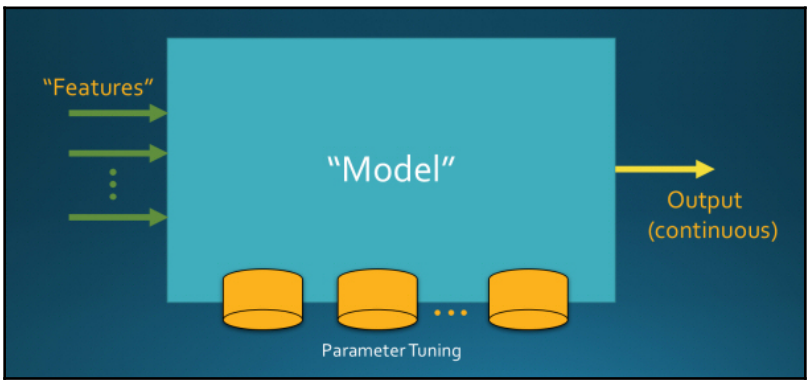
Iris Flower classifier illustrated

Classification tasks have a multitude of real-life applications, such as approval of bank loans and credit cards, email spam detection, handwritten digit recognition, and face recognition. Later in this chapter, we will be demonstrating the classification of animal types using the Zoo dataset.

The second main type of supervised machine learning, regression, will be described in the next subsection.

Regression

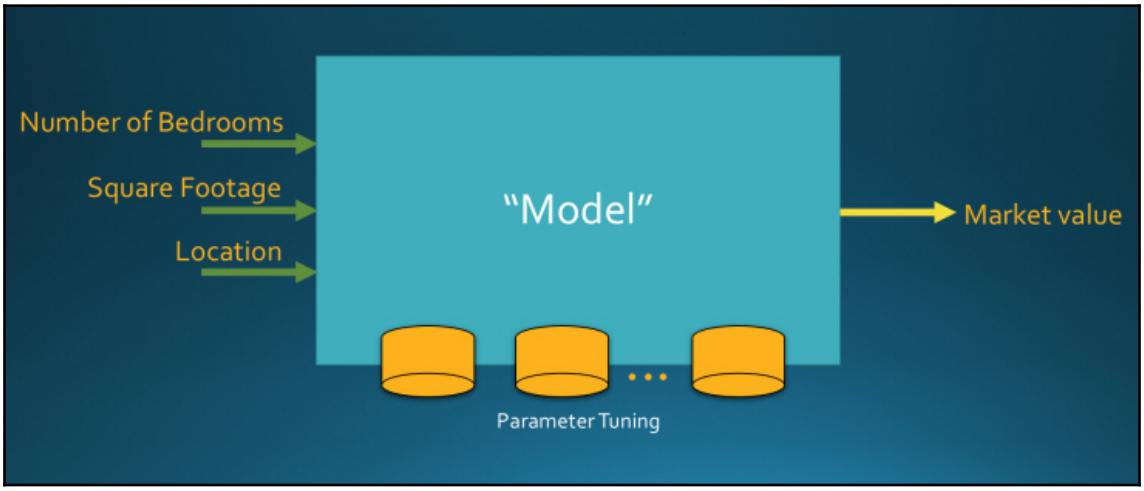
In contrast to classification tasks, the model for regression tasks maps the input values into a single output to provide a continuous value, as illustrated in the following image:



Machine learning regression model

Given the input values, the model is expected to predict the correct value of the output.

Real-life examples of regression include predicting the value of stocks, the quality of wine, or the market price of a house, as depicted in the following image:



House pricing regressor

In the preceding image, the inputs are features that provide information that describes a given house, while the output is the predicted value of the house.

Many types of models exist for carrying out classification and regression tasks – some of them are described in the following subsection.

Supervised learning algorithms

As we mentioned previously, each supervised learning model consists of a set of internal tunable parameters and an algorithm that tunes these parameters in an attempt to achieve the required result.

Some common supervised learning models/algorithms include the following:

- **Decision Trees**: A family of algorithms that utilizes a tree-like graph, where branching points represent decisions and the branches represent their consequences.
- **Random Forests**: Algorithms that create a large number of decision trees during the training phase and use a combination of their outputs.

- **Support Vector Machines**: Algorithms that map the given inputs as points in space so that the inputs that belong to separate categories are divided by the largest possible gap.
- **Artificial Neural Networks**: Models that consist of multiple simple nodes, or neurons, which can be interconnected in various ways. Each connection can have a weight that controls the level of the signal that's carried from one neuron to the next.

There are certain techniques that can be used to improve and enhance the performance of such models. One interesting technique – *feature selection* – will be discussed in the next section.

Feature selection in supervised learning

As we saw in the previous section, a supervised learning model receives a set of inputs, called **features**, and maps them to a set of outputs. The assumption is that the information described by the features is useful for determining the value of the corresponding outputs. At first glance, it may seem that the more information we can use as input, the better our chances of predicting the output(s) correctly. However, in many cases, the opposite holds true; if some of the features we use are irrelevant or redundant, the consequence could be a (sometimes significant) decrease in the accuracy of the models.

Feature selection is the process of selecting the most beneficial and essential set of features out of the entire given set of features. Besides increasing the accuracy of the model, a successful feature selection can provide the following advantages:

- The training times of the models are shorter.
- The resulting trained models are simpler and easier to interpret.
- The resulting models are likely to provide better generalization, that is, they perform better with new input data that is dissimilar to the data that was used for training.

When looking at methods to carry out feature selection, genetic algorithms are a natural candidate. We will demonstrate how they can be applied to find the best features out of an artificially generated dataset in the next section.

Selecting the features for the Friedman-1 regression problem

The Friedman-1 regression problem, which was created by Friedman and Breiman, describes a single output value, y, which is a function of five input values, $x_0..x_4$, and randomly generated noise, according to the following formula:

$$y(x_0, x_1, x_2, x_3, x_4) = 10 \cdot sin(\pi \cdot x_0 \cdot x_1) + 20(x_2 - 0.5)^2 + 10x_3 + 5x_4 + noise \cdot N(0, 1)$$

The input variables, $x_0..x_4$, are independent, and uniformly distributed over the interval [0, 1]. The last component in the formula is the randomly generated noise. The noise is normally distributed and multiplied by the constant noise, which determines its level.

In Python, the scikit-learn (`sklearn`) library provides us with the `make_friedman1()` function, which can be used to generate a dataset containing the desired number of samples. Each of the samples consists of randomly generated $x_0..x_4$ values and their corresponding calculated y value. The interesting part, however, is that we can tell the function to add an arbitrary number of irrelevant input variables to the five original ones by setting the `n_features` parameter to a value larger than five. If, for example, we set the value of `n_features` to `15`, we will get a dataset containing the original five input variables (or features) that were used to generate the y values according to the preceding formula and an additional 10 features that are completely irrelevant to the output. This can be used, for example, to test the resilience of various regression models on the noise and presence of irrelevant features in the dataset.

We can take advantage of this function to test the effectiveness of genetic algorithms as a feature selection mechanism. In our test, we will use the `make_friedman1()` function to create a dataset with 15 features and use the genetic algorithm to search for the subset of features that provides the best performance. As a result, we expect the genetic algorithm to pick the first five features and drop the rest, assuming that the model's accuracy is better when only the relevant features are used as input. The fitness function of the genetic algorithm will utilize a regression model that, for each potential solution – a subset of the feature to use – will be trained using the dataset containing only the selected features.

As usual, we will start by choosing an appropriate representation for the solution, as described in the next subsection.

Solution representation

The objective of our algorithm is to find a subset of features that yield the best performance. Therefore, a solution needs to indicate which features are chosen and which are dropped. One obvious way to go about this is to represent each individual using a list of binary values. Every entry in that list corresponds to one of the features in the dataset. A value of 1 represents selecting the corresponding feature, while a value of 0 means that the feature has not been selected. This is very similar to the approach we used in the knapsack 0-1 problem we described in `Chapter 4`, *Combinatorial Optimization*.

The presence of each 0 in the solution will be translated into dropping the corresponding feature's data column from the dataset, as we will see in the next subsection.

Python problem representation

To encapsulate the Friedman-1 feature selection problem, we've created a Python class called `Friedman1Test`. This class can be found in the `friedman.py` file, which is located at `https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter07/friedman.py`.

The main parts of this class are as follows:

1. The `__init__()` method of the class creates the dataset, as follows:

```
self.X, self.y =
datasets.make_friedman1(n_samples=self.numSamples,
n_features=self.numFeatures,
                                                noise=self.NOISE,
random_state=self.randomSeed)
```

2. Then, it divides the data into two subsets –a training set and a validation set – using the **scikit-learn** `model_selection.train_test_split()` method:

```
self.X_train, self.X_validation, self.y_train,
self.y_validation = \
    model_selection.train_test_split(self.X, self.y,
test_size=self.VALIDATION_SIZE, random_state=self.randomSeed)
```

Dividing the data to a test set and a validation set allows us to train the regression model on the train set, where the correct prediction is given to the model for training purposes, and then test it with the separate validation set, where the correct predictions are not given to the model and are, instead, compared to the predictions it produces. This way, we can test how well the model is able to generalize, rather than memorize the training data.

3. Next, we create the regression model, which is of the **Gradient Boosting Regressor** (**GBR**) type. This model creates an ensemble (or aggregation) of decision trees during the training phase:

```
self.regressor =
GradientBoostingRegressor(random_state=self.randomSeed)
```

Note that we are passing the random seed along so that it can be used internally by the regressor. This way, we can make sure the results that we obtain are repeatable.

4. The `getMSE()` method of the class is used to determine the performance of our gradient boosting regression model for a set of selected features. It accepts a list of binary values corresponding to the features in the dataset – a value of 1 represents selecting the corresponding feature, while a value of 0 means that the feature is dropped. The method then deletes the columns in the training and validation sets that correspond to the unselected features:

```
zeroIndices = [i for i, n in enumerate(zeroOneList) if n == 0]
currentX_train = np.delete(self.X_train, zeroIndices, 1)
currentX_validation = np.delete(self.X_validation, zeroIndices,
1)
```

5. The modified train set – containing only the selected features – is then used to train the regressor, while the modified validation set is used to evaluate its predictions:

```
self.regressor.fit(currentX_train, self.y_train)
prediction = self.regressor.predict(currentX_validation)
return mean_squared_error(self.y_validation, prediction)
```

The metric that's used here to evaluate the regressor is called the **mean square error** (**MSE**), which finds the average squared difference between the model's predicted values and the actual values. A lower value of this measurement indicates better performance of the regressor.

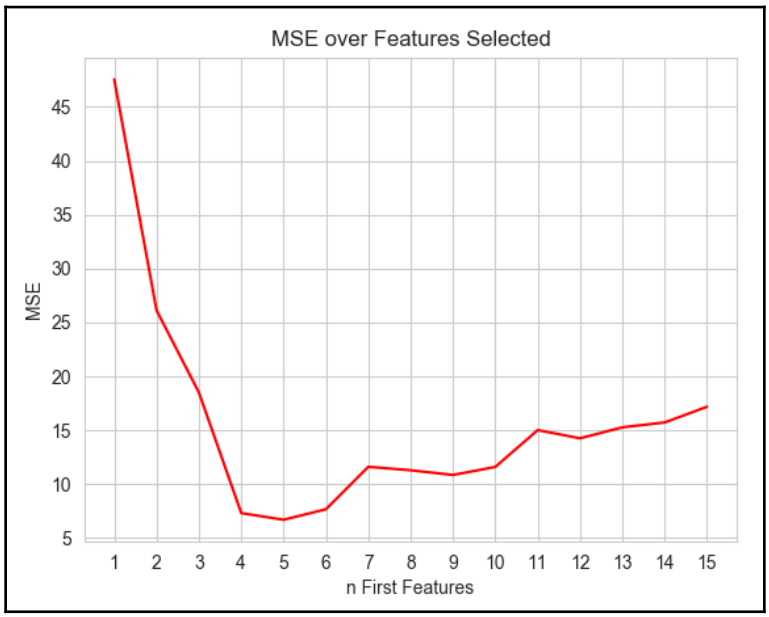
6. The `main()` method of the class creates an instance of the `Friedman1Test` class with 15 features. Then, it repeatedly uses the `getMSE()` method to evaluate the performance of the regressor with the first *n* features, since *n* is incremented from 1 to 15:

```
for n in range(1, len(test) + 1):
    nFirstFeatures = [1] * n + [0] * (len(test) - n)
    score = test.getMSE(nFirstFeatures)
```

When running the main method, the results show that, as we add the first five features one by one, the performance improves. However, afterward, each additional feature degrades the performance of the regressor:

```
1 first features: score = 47.553993
2 first features: score = 26.121143
3 first features: score = 18.509415
4 first features: score = 7.322589
5 first features: score = 6.702669
6 first features: score = 7.677197
7 first features: score = 11.614536
8 first features: score = 11.294010
9 first features: score = 10.858028
10 first features: score = 11.602919
11 first features: score = 15.017591
12 first features: score = 14.258221
13 first features: score = 15.274851
14 first features: score = 15.726690
15 first features: score = 17.187479
```

This is further illustrated by the generated plot, showing the minimum MSE value where the first five features are used:



Plot of error values for the Friedman-1 regression problem

In the next subsection, we will find out if a genetic algorithm can successfully identify these first five features.

Genetic algorithms solution

To identify the best set of features to be used for our regression test using a genetic algorithm, we've created the Python program `01-solve-friedman.py`, which is located at `https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter07/01-solve-friedman.py`.

As a reminder, the chromosome representation that's being used here is a list of integers with the values of 0 or 1, denoting whether a feature should be used or dropped. This makes our problem, from the point of view of the genetic algorithm, similar to the OneMax problem or the knapsack 0-1 problem we solved previously. The difference is in the fitness function returning the regression model's MSE, which is calculated within the `Friedman1Test` class.

The following steps describe the main parts of our solution:

1. First, we need to create an instance of the `Friedman1Test` class with the desired parameters:

```
friedman = friedman.Friedman1Test(NUM_OF_FEATURES,
NUM_OF_SAMPLES, RANDOM_SEED)
```

2. Since our goal is to minimize the MSE of the regression model, we define a single objective, minimizing the fitness strategy:

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
```

3. Since the solution is represented by a list of 0 or 1 integer values, we use the following toolbox definitions to create the initial population:

```
toolbox.register("zeroOrOne", random.randint, 0, 1)
toolbox.register("individualCreator", tools.initRepeat,
creator.Individual, toolbox.zeroOrOne, len(friedman))
toolbox.register("populationCreator", tools.initRepeat, list,
toolbox.individualCreator)
```

4. Then, we instruct the genetic algorithm to use the `getMSE()` method of the `Friedman1Test` instance for fitness evaluation:

```
def friedmanTestScore(individual):
    return friedman.getMSE(individual),  # return a tuple

toolbox.register("evaluate", friedmanTestScore)
```

5. As for the genetic operators, we use tournament selection with a tournament size of 2 and crossover and mutation operators that are specialized for binary list chromosomes:

```
toolbox.register("select", tools.selTournament, tournsize=2)
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutFlipBit,
indpb=1.0/len(friedman))
```

6. In addition, we continue to use the elitist approach, where the **hall of fame** (**HOF**) members – the current best individuals – are always passed untouched to the next generation:

```
population, logbook = elitism.eaSimpleWithElitism(population, toolbox,
cxpb=P_CROSSOVER,                mutpb=P_MUTATION, ngen=MAX_GENERATIONS,
stats=stats, halloffame=hof, verbose=True)
```

By running the algorithm for 30 generations with a population size of 30, we get the following outcome:

```
-- Best Ever Individual = [1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
-- Best Ever Fitness = 6.702668910463287
```

This indicates that the first five features have been selected to provide the best MSE (about 6.7) for our test. Note that the genetic algorithm makes no assumptions about the set of features that it was looking for, meaning it did not know that we are looking for a subset of the *first n* features. It simply searched for the best possible subset of features.

In the next section, we will advance from using artificially generated data to an actual dataset and utilize the genetic algorithm to select the best features for a classification problem.

Selecting the features for the classification Zoo dataset

The UCI *Machine Learning Repository* (`https://archive.ics.uci.edu/ml/index.php`) maintains over 350 datasets as a service to the machine learning community. These datasets can be used for experimentation with various models and algorithms. A typical dataset contains a number of features (inputs) and the desired output, in a form of columns, with a description of their meaning.

In this section, we will use the UCI Zoo dataset (`https://archive.ics.uci.edu/ml/datasets/zoo`). This dataset describes 101 different animals using the following 18 features:

No.	Feature Name	Data Type
1	animal name	Unique for each instance
2	hair	Boolean
3	feathers	Boolean
4	eggs	Boolean
5	milk	Boolean
6	airborne	Boolean
7	aquatic	Boolean
8	predator	Boolean
9	toothed	Boolean
10	backbone	Boolean

11	breathes	Boolean
12	venomous	Boolean
13	fins	Boolean
14	legs	Numeric (set of values {0,2,4,5,6,8})
15	tail	Boolean
16	domestic	Boolean
17	catsize	Boolean
18	type	Numeric (integer values in range [1,7])

Most features are boolean (value of 1 or 0), indicating the presence or absence of a certain attribute, such as hair, fins, and so on. The first feature, **animal name**, is just to provide us with some information and does not participate in the learning process.

This dataset is used for testing classification tasks, where the input features need to be mapped into two or more categories/labels. In this dataset, the last feature – called **type** – represents the category, and is used as the output value. In this dataset, there are seven categories altogether. A type value of 5, for instance, represents an animal category that includes frog, newt, and toad. To sum this up, a classification model trained with this dataset will use features 2-17 (hair, feathers, fins, and so on) to predict the value of feature 18 (animal type).

Once again, we want to use a genetic algorithm to select the features that will give us the best predictions. Let's start by creating a Python class that represents a classifier that's been trained with this dataset.

Python problem representation

To encapsulate the feature selection process for the Zoo dataset classification task, we've created a Python class called `Zoo`. This class is contained in the `zoo.py` file, which is located at `https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter07/zoo.py`.

The main parts of this class are highlighted as follows:

1. The `__init__()` method of the class loads the Zoo dataset from the web while skipping the first feature – animal name – as follows:

```
self.data = read_csv(self.DATASET_URL, header=None,
usecols=range(1, 18))
```

2. Then, it separates the data to input features (first remaining 16 columns) and the resulting category (last column):

```
self.X = self.data.iloc[:, 0:16]
self.y = self.data.iloc[:, 16]
```

3. Instead of just separating the data into a training set and a test set, like we did in the previous section, we're using *k-fold cross-validation*. This means that the data is split into *k* equal parts and the model is evaluated *k* times, each time using *(k-1)* parts for training and the remaining part for testing (or validation). This is easy to do in Python using the scikit-learn library's `model_selection.KFold()` method:

```
self.kfold = model_selection.KFold(n_splits=self.NUM_FOLDS,
random_state=self.randomSeed)
```

4. Next, we create a classification model based on a decision tree. This type of classifier creates a tree structure during the training phase that splits the dataset into smaller subsets, eventually resulting in a prediction:

```
self.classifier =
DecisionTreeClassifier(random_state=self.randomSeed)
```

Note that we are passing the random seed along so that it can be used internally by the classifier. This way, we can make sure the results that are obtained are repeatable.

5. The `getMeanAccuracy()` method of the class is used to evaluate the performance of the classifier for a set of selected features. Similar to the `getMSE()` method in the `Friedman1Test` class, this method accepts a list of binary values corresponding to the features in the dataset – a value of 1 represents selecting the corresponding feature, while a value of 0 means that the feature is dropped. The method then drops the columns in the dataset that correspond to the unselected features:

```
zeroIndices = [i for i, n in enumerate(zeroOneList) if n == 0]
currentX = self.X.drop(self.X.columns[zeroIndices], axis=1)
```

6. This modified dataset – containing only the selected features – is then used to perform the k-fold cross-validation process and determine the classifier's performance over the data partitions. The value of *k* in our class is set to `5`, so five evaluations take place each time:

```
cv_results = model_selection.cross_val_score(self.classifier,
currentX, self.y, cv=self.kfold, scoring='accuracy')
return cv_results.mean()
```

   The metric that's being used here to evaluate the classifier is **accuracy** – the portion of the cases that were classified correctly. An accuracy of 0.85, for example, means that 85% of the cases were classified correctly. Since, in our case, we train and evaluate the classifier k times, we use the average (mean) accuracy value that was obtained over these evaluations.

7. The `main()` method of the class creates an instance of the `Zoo` class and evaluates the classifier with all 16 features that are present using the **all-one** solution representation:

```
allOnes = [1] * len(zoo)
print("-- All features selected: ", allOnes, ", accuracy = ",
zoo.getMeanAccuracy(allOnes))
```

When running the main method of the class, the printout shows that, when testing our classifier with 5-fold cross-validation using all 16 features, the classification accuracy that's achieved is about 91%:

```
-- All features selected:  [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
, accuracy =  0.9099999999999999
```

In the next subsection, we will attempt to improve the accuracy of the classifier by selecting a subset of features from the dataset, instead of using all the features. We will use – you guessed it – a genetic algorithm to select these features for us.

Genetic algorithms solution

To identify the best set of features to be used for our Zoo classification task using a genetic algorithm, we've created the Python program `02-solve-zoo.py`, which is located at `https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter07/02-solve-zoo.py`.

As in the previous section, the chromosome representation that's being used here is a list of integers with the values of 0 or 1, denoting whether a feature should be used or dropped.

The following steps highlight the main parts of the program:

1. First, we need to create an instance of the `Zoo` class and pass our random seed along for the sake of producing repeatable results:

```
zoo = zoo.Zoo(RANDOM_SEED)
```

2. Since our goal is to maximize the accuracy of the classifier model, we define a single objective, maximizing the fitness strategy:

```
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
```

3. Just like in the previous section, we use the following toolbox definitions to create the initial population of individuals, each constructed as a list of `0` or `1` integer values:

```
toolbox.register("zeroOrOne", random.randint, 0, 1)
toolbox.register("individualCreator", tools.initRepeat,
creator.Individual, toolbox.zeroOrOne, len(zoo))
toolbox.register("populationCreator", tools.initRepeat, list,
toolbox.individualCreator)
```

4. Then, we instruct the genetic algorithm to use the `getMeanAccuracy()` method of the `Zoo` instance for fitness evaluation. To do this, we had to make two modifications:
    - We eliminated the possibility of no features being selected (all-zeros individual) since our classifier will throw an exception in such a case.
    - We added a small penalty for each feature being used to encourage the selection of fewer features. The penalty value is very small (0.001), so it only comes into play as a tie-breaker between two equally performing classifiers, leading the algorithm to prefer the one that uses fewer features:

```
def zooClassificationAccuracy(individual):
    numFeaturesUsed = sum(individual)
    if numFeaturesUsed == 0:
        return 0.0,
    else:
        accuracy = zoo.getMeanAccuracy(individual)
        return accuracy - FEATURE_PENALTY_FACTOR *
numFeaturesUsed,  # return a tuple


toolbox.register("evaluate",
zooClassificationAccuracy)
```

5. For the genetic operators, we again use tournament selection with a tournament size of `2` and crossover and mutation operators that are specialized for binary list chromosomes:

```
toolbox.register("select", tools.selTournament, tournsize=2)
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutFlipBit,
indpb=1.0/len(zoo))
```

6. And once again, we continue to use the elitist approach, where HOF members – the current best individuals – are always passed untouched to the next generation:

```
population, logbook = elitism.eaSimpleWithElitism(population,
toolbox, cxpb=P_CROSSOVER,                mutpb=P_MUTATION,
ngen=MAX_GENERATIONS, stats=stats, halloffame=hof,
verbose=True)
```

7. At the end of the run, we print out all the members of the HOF so that we can see the top results that were found by the algorithm. We print both the fitness value, which includes the penalty for the number of features, and the actual accuracy value:

```
print("- Best solutions are:")
for i in range(HALL_OF_FAME_SIZE):
    print(i, ": ", hof.items[i], ", fitness = ",
hof.items[i].fitness.values[0],
          ", accuracy = ", zoo.getMeanAccuracy(hof.items[i]),
", features = ", sum(hof.items[i]))
```

By running the algorithm for 50 generations with a population size of 50 and HOF size of 5, we get the following outcome:

```
- Best solutions are:
0 : [0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0] , fitness = 0.964 ,
accuracy = 0.97 , features = 6
1 : [0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1] , fitness = 0.963 ,
accuracy = 0.97 , features = 7
2 : [0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0] , fitness = 0.963 ,
accuracy = 0.97 , features = 7
3 : [1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0] , fitness = 0.963 ,
accuracy = 0.97 , features = 7
4 : [0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0] , fitness = 0.963 ,
accuracy = 0.97 , features = 7
```

These results indicate that all five top solutions achieved an accuracy value of 97%, using either six or seven features out of the available 16. Thanks to the penalty factor on a number of features, the top solution is the set of six features, which are as follows:

- feathers
- milk
- airborne
- backbone
- fins
- tail

In conclusion, by selecting these particular features out of the 16 given in the dataset, not only did we reduce the dimensionality of the problem, but we were also able to improve our model accuracy from 91% to 97%. If this does not seem like a large enhancement at first glance, think of it as reducing the error rate from 9% to 3% – a very significant improvement in terms of classification performance.

Summary

In this chapter, you were introduced to machine learning and the two main types of supervised machine learning tasks – regression and classification. Then, you were presented with the potential benefits of feature selection on the performance of the models carrying out these tasks. At the heart of this chapter were two demonstrations of how genetic algorithms can be utilized to enhance the performance of such models via feature selection. In the first case, we pinpointed the genuine features that were generated by the Friedman-1 Test regression problem, while, in the other case, we selected the most beneficial features of the Zoo classification dataset.

In the next chapter, we will look at another possible way of enhancing the performance of supervised machine learning models, namely hyperparameter tuning.

Further reading

For more information about the topics that were covered in this chapter, please refer to the following resources:

- *Applied Supervised Learning with Python*, *Benjamin Johnston* and *Ishita Mathur*, April 26, 2019
- *Feature Engineering Made Easy*, *Sinan Ozdemir* and *Divya Susarla*, January 22, 2018
- **Feature selection for classification, M.Dash and H.Liu, 1997**: `https://doi.org/10.1016/S1088-467X(97)00008-5`
- **UCI Machine Learning Repository**: `https://archive.ics.uci.edu/ml/index.php`

Index