Dynamic Evolution of Simulated Autonomous Cars in the Open World Through Tactics

Joe R. Sylnice and Germán H. Alférez(✉)

School of Engineering and Technology, Universidad de Montemorelos, Apartado 16-5, Montemorelos, N.L. 67500, Mexico
1140134@alumno.um.edu.mx, harveyalferez@um.edu.mx

Abstract. There is an increasing level of interest in self-driving cars. In fact, it is predicted that fully autonomous cars will roam the streets by 2020. For an autonomous car to drive by itself, it needs to learn. A safe and economic way to teach a self-driving car to drive by itself is through simulation. However, current car simulators are based on closed world assumptions, where all possible events are already known as design time. Nevertheless, during the training of a self-driving car, it is impossible to account for all the possible events in the open world, where several unknown events may arise (i.e., events that were not considered at design time). Instead of carrying out particular adaptations for known context events in the closed world, the system architecture should evolve to safely reach a new state in the open world. In this research work, our contribution is to extend a car simulator trained by means of machine learning to evolve at runtime with tactics when the simulation faces unknown context events.

Keywords: Autonomous car · Tactics · Dynamic evolution
Open world · Machine learning

1 Introduction

A human driver learns by practicing how to drive and how to detect problems in the car and on the road. It is basically the same in the case of autonomous cars. These cars learn from historical data to learn how to drive.

However, a self-driving vehicle is really expensive to build and maintain. In fact, there are reports informing that NVIDIA is selling its self-driving processing unit for about $15,000 [1]. That is really expensive taking into account that this is the price of only the processing unit. Also, it is dangerous and careless to unleash a self-driving car without proper training and testing. Simulations to prove new approaches in autonomous cars could be used to solve the aforementioned problems in the academic world, and especially in developing countries with limited financial resources.

In the closed world, all the possible context events are known beforehand (i.e., at design time or during training under a machine-learning approach). However, in the open world, unknown context events can arise (e.g. a sudden malfunction in one of the car sensors). This kind of events have to be controlled efficiently in order to prevent problems with the driver and passengers. Moreover, although there are open-source simulators, these simulators do not manage uncertainty in the open world.

In this research work, our goal is to extend the applicability of machine learning by means of tactics to carry out the dynamic evolution of simulated autonomous cars in the open world. Tactics are last-resort surviving actions to be used when the simulated car does not have predefined adaptation actions to deal with arising problematic context events in the open world [2]. In order to apply tactics in the open world, the source code of a car video game was modified. First, the car was trained with the following supervised learning algorithms: K-Nearest Neighbors, Logistic Regression, Support Vector Machines, and Decision Trees. Then, unknown context events were injected at runtime to evaluate how the car faces those events with tactics.

This paper is organized as follows. Section 2 presents the theoretical foundation of this research work. Section 3 presents the results. Finally, Sect. 4 presents the conclusions and future work.

2 Justification

The research field of self-driving cars is a hot topic nowadays. However, the technology behind a self-driving car relies heavily on state-of-the-art software and really expensive hardware. That is why simulation tools are being increasingly used in the field because they provide the mechanisms to test and evaluate the system of a self-driving car without having to buy (or even damage) really expensive hardware [3].

Predefined adaptation actions for known context events in the closed world are not enough in the open world where several unknown context events can arise. Despite the recognized need for handling unexpected events in self-adapting systems (SAS) [4], the dynamic evolution of SAS in the open world is still and open and challenging research topic.

In order to visualize the impact of unknown context events in the open world, let us imagine a self-driving car that has been trained with machine learning. The training was carried out with datasets composed of known historical data (e.g. data related to sonar and LiDAR sensors). In other words, the training was applied in the closed world. However, at runtime several unknown events may arise in the open world. For instance, although the sensors are highly calibrated and thoroughly revised, it is possible that a sensor starts recording inaccurate data (e.g. because of a broken sonar sensor). This is a dangerous situation because inaccurate data could lead to an accident. If the car was not trained to face this kind of situations, then the following question arises: what will the car do? In order to answer this question, in addition to applying machine learning to train self-driving cars, it is necessary to count on mechanisms to lead the car to make the best decision despite unknown context events.

3 Underpinnings of Our Approach

Our approach is based on the following concepts (*Fig.* 1).

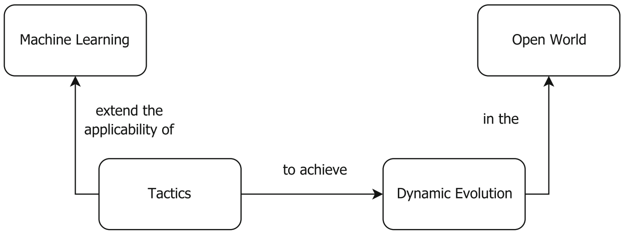

Fig. 1. Underpinnings of our approach.

3.1 Machine Learning

Machine learning can be defined as computational methods using experience to improve performance or to make predictions accurately. Experience can refer to past data that is used by the learner. The quality and size of the data are very important for the accuracy of the predictions made by the learner [5].

3.2 Tactics

Tactics are last-resort surviving actions to be used when a system does not have predefined adaptation actions to deal with arising problematic context events in the open world [2]. The use of tactics is common in sports, war, or even in daily matters to accomplish an end. For example, the most important goal during a battle is to win. However, unknown or unforeseen events, such as surprise assaults, may arise. These events may negatively affect the expected goal. Therefore, it is necessary to choose among a set of tactics to reach the goal (e.g. to escape vs. to do a frontal attack). Tactics are predefined at design time and are used at runtime to trigger the dynamic evolution of the self-driving car. The tactics are required to be known beforehand in order for the self-driving car to face uncertainty. However, these tactics are not associated with any specific reconfiguration actions (as dynamic adaptation does) [6].

3.3 Dynamic Evolution

A self-driving car has to go from dynamic adaptation in the closed world to dynamic evolution in the open world in order to respond to unforeseen ongoing events. *Dynamic adaptation* can be referred to as punctual changes made to face particular events by activating and deactivating system features based on the current context. Meanwhile, *dynamic evolution* is not just about applying punctual adaptations to concrete events but it is the gradual growth of the system to a better state depending on the current context events [2].

3.4 Open World

Open world can be referred to as a context where events are unpredictable, requiring that software reacts to these events by adapting and organizing its behavior by itself [7]. As far as we know, current simulated autonomous cars are based on the closed world assumption where the relationship between the car and the surroundings are known and unchanging. Nevertheless, in the open world where the aforementioned relationship is unknown, unpredictable, and constantly changing, the simulated car has to be able to evolve.

4 Related Work

A fully autonomous car or self-driving vehicle is a car that is designed to be able to do all the work of maneuvering the car without the passenger never having to or is not expected to take control of the car at any time or any given moment [8]. A self-driving vehicle has to be able to identify faults in its system. If the faults are critical, the vehicle has to either fix these faults or isolate them so that the system is not compromised [9].

Self-driving vehicles are equipped with state-of-the art sensors and cameras. Also, they use powerful software behind the hardware to maneuver themselves. The software learns how to drive through machine learning and the software sees through computer vision.

There are several self-driving cars in development. For example, the Google Car is being developed by Google. Google hopes to have self-driving cars on the road by 2020. However, this company does not intend to become a car manufacturer. Uber also entered the world of self-driving cars in April 2015. In addition, Tesla expects to launch a fully autonomous car anytime in 2018. Also, in April 2015, BMW has partnered with Baidu the "Chinese Google", to develop self-driving technology.

There are several research works that propose simulations of autonomous cars. For instance, in [10] the authors propose a shader-based sensor to simulate the LiDAR and Radar sensors instead of the common method of ray tracing. They mention that sensor simulations are very important in the field of self-driving cars. In this way, the sensors can be evaluated, tested and optimized. The authors state that ray tracing is an intensive task for the CPU. It is not problematic when the number of simulated rays and detected objects are small. However, in reality it becomes problematic or even impossible. According to the authors, a shader-based sensor simulation is an efficient alternative to ray casting because it uses parallelism in the GPU and this helps in sparing CPU resources that the software can use in other areas.

In [11], the authors mention that they have used a simulation tool called Scene Suite to generate simulated scenes of traffic scenarios. The tool allows 2.5D simulations and uses patented virtual sensor models. The goal of this work is to show how the data from real world sensor models could be extracted and then to simulate the results using a scene based pattern recognition. Also, this paper introduced an approach for learning sensor models with a manageable demand on computational power based on a statistical analysis of measurement data clustered into scene primitives.

In [12], the authors focus on the use of the agent-based simulation framework MATsim and how it could be applied to the field of self-driving cars. Agent-based simulations are state-of-the-art transport models. Agent-based approaches combine activity-based demand generation and dynamic traffic assignments. MATSim is a simulation of multi-agent transport based on activity. It is an open source framework written in JAVA under the GNU license. MATSim's strength is the modular design around a core, allowing new users to customize it without much effort. This work is based on the simulation of autonomous vehicles in a realistic environment at a large scale with individual travelers (vehicles) that adapt their movement dynamically with the others.

In [13], the author uses an open source simulator to carry out the evaluation and application of a reinforcement learning approach to the problem of controlling the steering of a vehicle. Reinforcement Learning (RL) is an area of machine learning in which an agent is placed into a certain environment and is required to learn how to take proper actions without having any previous knowledge about the environment itself. If the agent's behavior is right, it is rewarded. If the behavior is wrong, the agent is punished. This learning system of reinforcement learning is called trial and error. In order to evaluate this approach, the Open Racing Car Simulator (TORCS) was used. In the TORCS environment a car is referred to as a Robot.

In [3], the authors use an integrated architecture that is comprised of both a traffic simulator and a robotics simulator in order to contribute to the self-driving cars simulation. Specifically, the proposed approach uses the traffic simulator SUMO and the robotics simulator USARSim. These tools are open source and have good community support. In one hand, SUMO is a microscopic road traffic simulator written in C++. It was designed by the Institute of Transportation Systems at the German Aerospace Center to handle large road networks. On the other hand, USARSim is an open-source robotics simulator written in Unreal Script, which is the language of the Unreal game engine. It has high quality sensor simulation and physics rendering. The authors modified the SUMO and USARSim simulators in order to be able to implement the architecture for the self-driving car simulation. The result is a simulator in which a self-driving vehicle can be deployed in a realistic traffic flow.

In [14], the authors describe the global architecture of the simulation/prototyping tool named Virtual Intelligent Vehicle Urban Simulator (VIVUS) developed by the SeT Laboratory. The VIVUS simulator simulates vehicles and sensors. It also takes into account the physical properties of the simulated vehicle while prototyping the artificial intelligence algorithms such as platoon solutions and obstacle avoidance devices. The goal of VIVUS is therefore overcoming the general drawbacks of classical solutions by providing the possibility of designing a vehicle virtual prototype with simulated embedded sensors.

In [15], the authors combine a traffic simulator and a driving simulator into an integrated framework. They have used the driving simulator SCANeR developed by Renault and Oktal, and the AIsum traffic simulator developed by TSS-Transport Simulation Systems. The framework enables a driver to use the simulator with a local traffic situation managed by a nano traffic model that is realistic for the driver and that also provides a realistic global traffic situation in terms of flow and density. The framework can provide information on the simulated vehicles and the traffic situation for the short-ranged sensors: camera and radar and also the long-ranged sensors: wireless and embedded navigation. It also enables the driver and other systems to be involved in an extensive assortment of traffic situations, accidents, rerouting, road-work zones, and so on.

5 Results

5.1 Methodology

This project has been broken down in the following steps:

Looking for an Open Source Car Simulator: To find the open source car simulator, *Google Search* was used with the term *"open source car simulator*" in December 2017. The following is the list of the open source car simulators found:

- **TORCS**[1]**:** *TORCS* is a multi-platform car racing simulation. It is used as an ordinary car racing game, as an artificial intelligence (AI) racing game, and as a research platform.
- **Apollo**[2]**:** *Apollo* is an open-source autonomous driving platform created by *Baidu*. It has a high performance and flexible architecture that supports fully autonomous driving capabilities and also has car simulation functionalities.
- **Udacity's Self-Driving Car Simulator**[3]**:** This simulator was built for *Udacity*'s Self-Driving Car nanodegree to teach students how to train cars and how to navigate road courses using deep learning.

Comparing Different Open Source Car Simulators: The criteria for choosing the car simulator were the following: (1) it had to be open source to find the points in which it could be extended; (2) it had to be mature enough in terms of documentation; (3) it had to be supported by the developer community; and (4) it had to be easily extensible in terms of programming. The results of the comparison are as follows:

1. *TORCS* meets three of the four criteria. Although, it is open source, mature, well known in the scientific world, and is greatly supported by the developer community, it misses the fourth criteria because it is not easily extensible in terms of programming.

[1] http://torcs.sourceforge.net/index.php?name=Sections&op=viewarticle&artid=1.
[2] https://github.com/ApolloAuto/apollo.
[3] https://github.com/udacity/self-driving-car-sim.

2. *Apollo* is a fully fledged open autonomous driving platform that meets two of our criteria: it is open source and mature. However, it is a fully autonomous driving platform, much more complex than a simulator. Also, since it was released a couple of months prior to our search, it does not yet have a wide developer community support. Also, the documentation, written in Chinese is not yet translated.
3. *Udacity's self-driving car simulator* falls short when it comes to documentation. As a result, although it is an open source software, the lack of free documentation makes it difficult to extend the code.

According to the evaluation, none of these simulators fulfilled our needs. Therefore, instead of searching for open source autonomous car simulators, we looked for an open source car game, which could be trained by means of machine learning and extended for usage in the open world.

We found an open source car game named Lapmaster[4]. It is a simple car game designed with the *pygame* Python library. It consists of a car running around a circuit for a certain amount of laps. Also, the player is able to shift the gears. The goal of the game is to complete the laps as fast as possible. *Fig.* 2 shows a screenshot of this game.

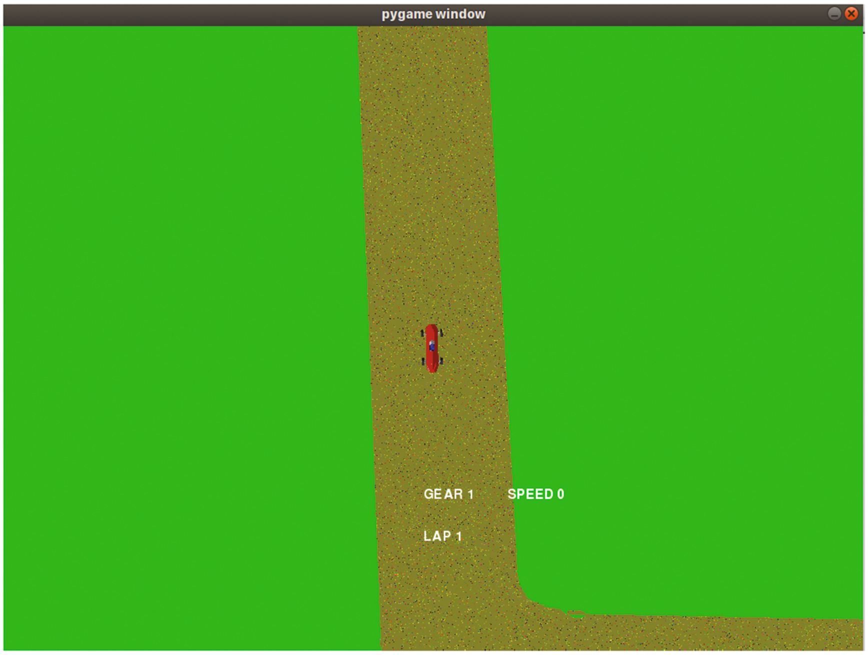

Fig. 2. Screenshot of the Lapmaster game.

[4] http://pygame.org/project-Lap+Master-2923-4798.html.

Extending the Car Simulator: In this step, the Lapmaster car simulator was extended for the open world. Specifically, two steps were carried out: (1) collecting data from the context of the car for training; and (2) training the simulated car with machine learning. These steps are described as follows.

1. *Collecting data from the context of the car:* The source code of the car game was modified to collect the position (x and y coordinates) and the direction (0 - forward, 1 - right, and 2 - left) of the car in every frame. *Listing 1* shows the modified lines of the car's source code. On line 1, a while loop indicates that the code is executed while the car simulator is running. On line 2, the program detects the key that is pressed. On line 3, if the car is moving, then the program checks if the key "*d*" (right) or key "*a*" (left) is pressed. These values are stored in the *l_data* list. Specifically, three values are stored in this list: the x and y coordinates, and the direction (0 for forward, 1 for right, and 2 for left). If no key is pressed, then the program stores a 0 in the *l_data* list. On line 12, if the *l_data* list is not empty, then it is passed to the *Writer* function with the log's path in which the contextual data is to be written. *Listing 2* presents the *Writer* function which writes the data in the comma-separated values (CSV) format. The CSV file contains 4,149 instances. This number of instances was obtained by running the game four times. The x and y coordinates were taken as the features for training, and the *direction* as the class.

```
 1  while running:
 2    key = pygame.key.get_pressed()
 3    if red.gear > 0:
 4      if key[K_d]:
 5        red.view = (red.view + 2) % 360
 6        d = 1
 7      elif key[K_a]:
 8        red.view = (red.view + 358) % 360
 9        d = 2
10      else:
11        d = 0
12    l_data = [red.xc, red.yc, d]
13      if l_data:
14      data.Writer(l_data, path)
```

Listing 1.1. A fragment of the modified code of the Lapmater's source file.

```
1  import csv
2  def Writer(data, path):
3    with open(path, "a") as c_file:
4      write = csv.writer(c_file, delimiter=',')
5      write.writerow(data)
```

Listing 1.2. Implemented function for data writing.

2. *Training the simulated car:* For the training of the simulated car, four supervised machine learning algorithms from the *scikit-learn*[5] Python library were employed. The algorithms are the following [16]:
   (a) **K-Nearest Neighbor (KNN):** It is a simple algorithm that stores all available cases and classifies new cases by a majority vote of its k neighbors.

[5] http://scikit-learn.org/stable/#.

(b) **Logistic Regression (LR):** It is a classification algorithm used to estimate discrete values based on a given set of independent variables. It predicts the probability of occurrence of an event by fitting data to a *logit function.*
(c) **Support Vector Machine (SVM):** In this classification algorithm, each data point is plotted in an n-dimensional (n being the number of features) space where the value of each feature is the value of a particular coordinate. Then a line called *separating hyper plane* or (*decision boundary*) splits the data points between two or more groups of data. The further the data points from the *decision boundary*, the more confident the algorithm is about the prediction. The closest data points to the separating hyper plane are known as *support vectors*.
(d) **Decision Trees (DT):** In this classification algorithm, the data is split into two or more homogeneous sets based on most significant attributes that makes the sets distinct.

The following are the steps used to train the simulated car: (1) a user ran the game to generate a dataset; (2) the KNN, LR, SVM, and DT algorithms were executed to get a classification for each class. The classes were 0 for forward, 1 for right, and 2 for left; (3) the models were evaluated in terms of cross validation; and (4) the simulated car was extended to use the most accurate classifier.

A fragment of the script to generate the classification models from the data collected is presented in *Listing 3.* The first line declares a list containing the information of the four classifiers used in the experiments. Next, a for loop is used to iterate over this list in order to train and generate a model for each algorithm. Line 9 specifies the location and the name of the model that is going to be trained. In Lines 12 and 13, the program splits the data into training and test sets. The code in Line 11 indicates that the values are going to be taken randomly from the dataset. On lines 14–15, a classification model is created and the cross-validation score is evaluated. In Line 17, the accuracy of each algorithm is computed. In Lines 18–21, each model is evaluated and a classification report is generated. Finally, the model generated by each algorithm is saved.

```
 1  classifiers = [
 2    ('kNN', KNeighborsClassifier(n_neighbors=4)),
 3    ('LR', LogisticRegression()),
 4    ('SVM', SVC()),
 5    ('DT', DecisionTreeClassifier())
 6  ]
 7
 8  for name, clf in classifiers:
 9    filename = 'models/%s_%s.pickle' % (name, data.filename)
10    print('training: %s' % name)
11    rs = np.random.RandomState(42)
12    X_train, X_test, y_train, y_test =
13      train_test_split(X, y, test_size=0.2, random_state=rs)
14    model = clf.fit(X_train, y_train)
15    cv = cross_val_score(clf, X_test, y_test, cv=10,
16                         scoring='accuracy')
17    acc = np.mean(cv)
18    predictions = clf.predict(X_test)
19    report = classification_report(y_test, predictions)
20    print('training %s done... acc= %f' % (name, acc))
21    pickle.dump(model, open(filename, 'wb'))
22    bm.append('%s %s' % (name, report))
```

Listing 1.3. A fragment of code to train and generate classification models.

Injecting Dynamic Evolution Through Tactics: In this step, we emulated that a sonar sensor was malfunctioning. This situation can cause accidents since the car will not be able to "see" properly its environment (e.g. other cars). To trigger this event, a button on the keyboard was pressed. When the car system recognizes that an unknown context event has arisen, then the "*decelerate tactic*" is triggered. This tactic progressively slows down the car until it reaches the state of a full stop. The reasoning behind this tactic is to prevent that the car keeps going on without properly detecting its surroundings. The implemented tactic is shown in *Listing 4*. Specifically, when the "*s*" key is pressed on the keyboard, the *slow* variable is set to true to indicate that the car has to reduce the speed until if fully stops.

```
slow = False

key = pygame.key.get_pressed()
if key[K_s]:
  slow = True
if slow:
  red.speed = .95 * red.speed - .05 * (2.5 * red.gear)
```

Listing 1.4. A fragment of the source code for the decelerate tactic.

5.2 Outcomes

The accuracy of the models generated with the four algorithms are as follows: kNN = 0.9313, LR = 0.8927, SVM = 0.8927, DT = 0.929. Table 1 shows the cross validation results of each model generated with the four classifiers. Also, in Table 1, only two classes are shown: 0 for forward and 1 for right. That is because the circuit in the Lapmaster game only has right turns. Although the kNN algorithm has the best accuracy, the DT algorithm has better results in terms of *precision*, *recall*, and *f1-score*. The three aforementioned terms are defined as follows [17]:

- *Precision* is the ability of the classifier not to identify as positive a sample that is negative.
- *Recall* is the ability of the classifier to find all the positive samples.
- *F1-score* is a weighted mean of the precision and recall.

5.3 Discussion

We published a video[6] in which the "*decelerate tactic*" is effectively triggered at runtime. Although machine learning works fine in the closed world, i.e., where there are no unknown events (e.g. malfunctioning sensors), in the open world it is necessary to count with additional mechanisms to face uncertainty. Therefore, we argue that autonomous cars that are trained by means of machine learning need to be extended with highly general tactics that try to defend the car in extreme conditions of uncertainty.

[6] www.harveyalferez.com/autonomous-car-demo.html.

Table 1. Report for each of the algorithm models.

	Precision	Recall	f1-score
kNN			
0	0.95	0.99	0.97
1	0.83	0.56	0.67
Avg/Total	0.94	0.94	0.94
LR			
0	0.89	1.00	0.94
1	0.00	0.00	0.00
Avg/Total	0.80	0.89	0.84
SVM			
0	0.90	1.00	0.95
1	1.00	0.03	0.07
Avg/Total	0.91	0.90	0.85
DT			
0	0.97	0.98	0.97
1	0.82	0.71	0.76
Avg/Total	0.95	0.95	0.95

6 Conclusions and Future Work

This research work extended the applicability of machine learning by means of tactics to carry out the dynamic evolution of a simulated self-driving car in the open world. To this end, four classifiers were executed and four models were generated and evaluated. The DT model was used in the simulated car after evaluation. Then, a tactic to face a simulated unknown context event in the open world was implemented. This tactic was used to prevent a situation in which the life of the passengers could be put in jeopardy.

Since this research work was limited to the implementation and application of one tactic, as future work we would like to propose additional tactics. For example, tactics related to non-functional requirements, such as availability and performance, could be used to keep or improve service levels. Also, these tactics could be handled during execution by means of models at runtime as proposed in our previous work [2]. Moreover, we plan to test our approach in other tracks in which complex unknown context events could arise.

References

1. Frederic, L.: All new Teslas are equipped with NVIDIA's new drive PX 2 AI platform for self-driving. https://goo.gl/xNSo8B
2. Alférez, G.H., Pelechano, V.: Achieving autonomic web service compositions with models at runtime. Comput. Electr. Eng. **63**, 332–352 (2017)
3. Pereira, J.L., Rossetti, R.J.: An integrated architecture for autonomous vehicles simulation. In: Proceedings of the 27th Annual ACM Symposium on Applied Computing, pp. 286–292. ACM (2012)
4. Cheng, B.H., De Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., et al.: Software engineering for self-adaptive systems: a research roadmap. Software engineering for self-adaptive systems, pp. 1–26. Springer, Heidelberg (2009)
5. Mohri, M., Rostamizadeh, A., Talwalkar, A.: Foundations of machine learning. MIT press (2012)
6. Alférez, G.H., Pelechano, V.: Facing uncertainty in web service compositions. In: 2013 IEEE 20th International Conference on Web Services (ICWS), pp. 219–226. IEEE (2013)
7. Baresi, L., Di Nitto, E., Ghezzi, C.: Toward open-world software: issues and challenges. Computer **39**(10), 36–43 (2006)
8. Coles, C.: Automated vehicles: a guide for planners and policymakers (2016)
9. Maurer, M., Gerdes, J.C., Lenz, B., Winner, H.: Autonomous driving: technical, legal and social aspects. Springer, Heidelberg (2016)
10. Wang, S., Heinrich, S., Wang, M., Rojas, R.: Shader-based sensor simulation for autonomous car testing. In: 2012 15th International IEEE Conference on Intelligent Transportation Systems, pp. 224–229. IEEE (2012)
11. Simon, C., Ludwig, T., Kruse, M.: Extracting sensor models from a scene based simulation. In: 2016 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI), pp. 259–264. IEEE (2016)
12. Boesch, P.M., Ciari, F.: Agent-based simulation of autonomous cars. IEEE Am. Control Conf. (ACC) **2015**, 2588–2592 (2015)
13. Piovan, A.G.: A neural network for automatic vehicles guidance. ACE **10**, 2 (2012)
14. Gechter, F., Contet, J.-M., Galland, S., Lamotte, O., Koukam, A.: Virtual intelligent vehicle urban simulator: application to vehicle platoon evaluation. Simul. Modell. Pract. Theory **24**, 103–114 (2012)
15. That, T.N., Casas, J.: An integrated framework combining a traffic simulator and a driving simulator. Procedia-Soc. Behav. Sci. **20**, 648–655 (2011)
16. Harrington, P.: Machine Learning in Action. Manning Publications (2012)
17. Scikit-Learn: sklearn.metrics.precision_recall_fscore_support. https://goo.gl/4xxkGJ