Artificial Intelligence – Programming Assignment 3

UNIZG FER, academic year 2018/19

Handed out: 14.5.2019. Due: 26.5.2019. at 23:59

Introduction

In this lab assignment you will solve problems from the domains of machine learning and reinforcement learning. The code you will be using can be downloaded as a zip archive at the web page of the university here or at the github repository of the class here.

This lab assignment consists of two independent tasks – a naive Bayes classifier and reinforcement learning algorithms. Each assignment is stored in a separate directory. The code for the assignments consists of Python files, some of which are required for you to read through and understand in order to implement the lab assignment, some which you will yourselves edit and a some which you can ignore.

Task 1: naive Bayes classifier (12 points)

After unpacking the zip archive and opening the naive Bayes subdirectory, the files and folders you will encounter are listed as follows:

Files you will edit:	
naiveBayesClassifier.py	Naive Bayes classifier implementation
Files you should study:	
util.py	Helper classes and methods
dataLoader.py	Data loading and feature extraction
classifier.py	Runnng and testing for the classifier
Helper files you can ignore:	
layout.py	Class definitions for the maps Pacman navigates
game.py	The model for the game

The autograder isn't available for the first task, however compatability tests exist. Fixed outputs of a reference implementation are hardcoded at the top of *classifier.py*, and you can check whether your output is correct by running `python classifier.py -t 1`, which will write the absolute error between your *a posteriori* probabilities and the correct ones. Obtaining zero error does not entail full points on the assignment, as the outputs are checked only for a small subset of the testing examples.

A naive Bayes classifier is defined in the style of a popular Python machine learning framework, scikit-learn. Each machine learning model accepts its *hyperparameters* as inputs to the constructor, and has a *fit* method, where the classifier is tuned on the training data, and a *predict* method, which predicts the outputs for new instances.

In the scope of the lab assignment, we will tackle the problem of predicting Pacman's future moves. In the *pacmandata* directory compressed data regarding the current state

of the game and the chosen next step is stored. The data is split in training and testing sets, and belongs to four different types of agents: a hungry Pacman which always moves towards the closest food (prefix *food*), a scard Pacman which always stands at the same spot (prefix *stop*), a brave Pacman which always moves towards the closest ghost (prefix *suicide*) and a Pacman learned through techniques of reinforcement learning which successfully avoids ghosts and eats food (prefix *contest*).

As you will see in the second part of the lab assignment, learning behavior from scratch is a difficult process – the model needs to play a lot of games until the behavior of Pacman gets accustomed to the game and starts consistently winning. This can to some extent be made easier through imitation learning, and we can adapt our model to the behavior of another agent, copying its moves instead of learning to play the game from scratch

The main problem with such models is that we know the move which the agent made and the state of the game, but we have no knowledge of the criteria based on which the decision has been made. However, we hope that after enough observations (training examples) this problem will be manageable. The first step of implementing imitation learning is already implemented, and features which indicate the next potential move have already been extracted from the game state data.

In the *classifier_data* directory the extracted data is stored in tab-separated values *".tsv"* format which is compatible with the classifier implementation. The first row, or the so-called header contains the names of the features, while the last column contains the target variable – the decisions that the agent has made in the current game state. For the basic setup, only two features were extracted for each possible move – whether by making that move, we eat a food dot and whether that move reduces the distance to the closest ghost.

An example of the data found in the *"contest_training.tsv"* file looks as follows:

foodEaten_East	foodEaten_North	...	ghostCloser_Stop	ghostCloser_West	target
True	False	...	False	True	West
False	False	...	False	False	West

Problem 1.1: Classifier implementation (12 bodova)

In the *naiveBayesClassifier.py* file a skeleton of the naive Bayes classifier is implemented but some methods are left to you to complete. Your task in the first part of the lab assignment is to complete the implementation and obtain a fully functioning classifier.

Let us remind ourselves o the Bayes rule, and assume we have a dataset originating from the Pacworld, and we are looking for the most likely move that the Pacman agent we are imitating has made.

$$P(h_i|data) = \frac{P(data|h_i)P(h_i)}{P(data)}$$

Where h_i is the hypothesis, e.g. every possible move i, $P(h_i)$ the **a priori** probability of that move, $P_(h_i|data)$ the **a posteriori** probabilities of moves **for the input data** and $P(data|h_i)$ the **likelihood** of the data for a given move. $P(data)$ is the probability of the data instance itself, and serves as a normalization constant.

In our example, the hypotheses are the **moves** we can make (West, North, ...) and the data are the **state features** – foodEaten_East, foodEaten_North, ... based on which

we make our decision. The naive assumption of the Bayes classifier is that the features corresponding to a single instance are mutually independent, which allows us to factorize the total probability for all features $P(data|h_i)$ as the product of the probability for each feature!

The naive Bayes classifier chooses the hypothesis which maximizes the **a posteriori** probability as the correct one.

In short, the naive Bayes classifier can be formulated as follows:

$$h_{MAP} = \arg\max_{h_i \in H} P(h_i|f) = \arg\max_{h_i \in H} \frac{P(f|h_i)P(h_i)}{P(f)} = \arg\max_{h_i \in H} P(f|h_i)P(h_i)$$

$$h_{MAP} = \arg\max_{h_i \in H} P(h_i) \prod_j P(f_j|h_i)$$

Where f_j are the features and h_i the hypotheses (target variable). Eliminating the normalization constant doesn't affect the maximization, and the last formula given is the model of the classifier which we need to implement.

`NB.fit(self, trainingData, trainingLabels):`

In the *fit* method of our classifier you need to compute the likelihood of each feature $P(f_j|h_i)$ and the a priori probabilities of each hypothesis $P(h_i)$ based on the input data. The learned values should be stored in the class variables `self.prior` and `self.conditionalProb`. You also need to implement smoothing with the factor k, which is given as a model hyperparameter. For $k = 0$ there should be no smoothing.

The inputs to the method are two lists – *trainingData* and *trainingLabels*, which contain instances from the training set. We consider a pair (features, label) as an instance, where the features are mapped to the correct label.

TrainingData consists of a list of dictionaries, where the keys are the names of the features (ex. foodEaten_East), while the values are the values of these features for the specific instance (ex. False). TrainingLabels consists of a list of strings which represent the labels (ex. West).

Hint: Notice that it is necessary to count how many times a value of each feature occurred along with every label (ex. what is the frequency of foodEaten_East=True when the label is West)

Hint: the `self.featureValues` class variable will contain all possible feature values for each feature. featureValues itself is a dictionary containing feature names as keys, while the values are sets of all possible values these features can have.

`NB.calculateJointProbabilities(self, instance):`

In the *calculateJointProbabilities* method you need to compute the **a posteriori** probabilities for each possible hypothesis. This method is iteratively called by the *predict* method, where on the basis of the a posteriori probabilities the MAP hypothesis is selected.

The input to the method is a single *instance* in the form of a dictionary, where the keys are feature names and the values the values of those features. The output of the method should be a dictionary (precisely, the util.Counter subclass of a dictionary), where for the key of every possible hypothesis (`self.legalLabels`) its a posteriori probability is stored.

`NB.calculateLogJointProbabilities(self, instance):`

The *calculateLogJointProbabilities* method has the same functionality of computing the a posteriori probabilities for each hypothesis, however it implements the logarithm trick for numerical stability. Your task is to implement the log trick posterior probability computation and provide an argument as to why the log transform is a good choice here.

The inputs and outputs should be same as for the *calculateJointProbabilities* method.

For a correct implementation, your classifier should pass all tests and reach the following performance on the *contest* dataset. The key aspect is to match the prediction accuracy (percentages).

```
Performance on train set for k=0.000000, log=False: (78.8%)
Performance on test set for k=0.000000, log=False: (82.2%)
Performance on train set for k=0.000000, log=True: (78.8%)
Performance on test set for k=0.000000, log=True: (82.2%)
Performance on train set for k=1.000000, log=True: (78.7%)
Performance on test set for k=1.000000, log=True: (82.0%)
```

The tests, as well as the performance check for your model on a dataset can be ran through the main method of the `classifier.py` file, for which the instructions are as follows:

```
USAGE:      python classifier.py <options>
EXAMPLES: >> python classifier.py -t 1
              // runs the implementation tests on the 'contest' dataset
          >> python classifier.py --train contest_training --test contest_test -s 1 -l 1
              // run the naive Bayes classifier on the
                contest dataset with smoothing=1 and log scale transformation
          >> python classifier.py -h
              // display the help docstring

Options:
  -h, --help            show this help message and exit
  --train=TRAIN_DATA    the TRAINING DATA for the model [Default:
                        stop_training]
  --test=TEST_DATA      the TEST DATA for the model [Default: stop_test]
  -s SMOOTHING, --smoothing=SMOOTHING
                        smoothing [Default: 0]
  -l LOGTRANSFORM, --logtransform=LOGTRANSFORM
                        Compute a log transformation of the joint probability
                        [Default: 0]
  -t RUNTESTS, --test_implementation=RUNTESTS
                        Disregard all previous arguments and check if the
                        predicted values match the gold ones
```

Task 2: Reinforcement learning (12 points)

After unpacking the zip archive and opening the reinforcement learning subdirectory, the files and folders you will encounter are listed as follows:

Files you will edit:	
valueIterationAgents.py	The logic of the value iteration algorithm
qlearningAgents.py	The logic of Q-learning based algorithms
Files you should study:	
mdp.py	Description of the general markov decision process
learningAgents.py	Logic describing the agents
util.py	Helper classes and methods
gridworld.py	GridWorld description
featureExtractors.py	File that extracts features for approximate Q-learning
Helper files you can ignore:	
environment.py	Abstract class for reinforcement learning problems
graphicsUtils.py	Helper functions for graphic display
graphicsGridworldDisplay.py	Graphic display of the GridWorld
crawler.py	Crawler logic
graphicsCrawlerDisplay.py	Graphic display of the crawler
autograder.py	Helper file for running tests

The code for the second part of the lab assignment was adapted from the course "Intro to AI" at Berkeley, which allowed the usage of their Pacman environment for other universities for educational purposes. The code is written in Python 2.7, which you require an interpreter for in order to run the lab assignment.

After you've downloaded and unpacked the code and positioned your console in the subdirectory where you have unpacked the archive, you can test the GridWorld by manually controlling the movement with arrows by typing:

```
python gridworld.py -m
```

GridWorld is a world where your movement is made difficult by the fact that you only have an 80% chance to actually do the action that you wanted to do, and a 20% chance to make a random action. The noise can be modified with the parameter '-n', such as:

```
python gridworld.py -m -n 0.5
```

In which case we set the noise rather high, to 50%. This parameter as well as the rest of them can be seen by calling the parameter '-h (help)'. The output is as follows:

```
Options:
  -h, --help            show this help message and exit
  -d DISCOUNT, --discount=DISCOUNT
                        Discount on future (default 0.9)
  -r R, --livingReward=R
                        Reward for living for a time step (default 0.0)
  -n P, --noise=P       How often action results in unintended direction
                        (default 0.2)
  -e E, --epsilon=E     Chance of taking a random action in q-learning
                        (default 0.3)
  -l P, --learningRate=P
                        TD learning rate (default 0.5)
  -i K, --iterations=K  Number of rounds of value iteration (default 10)
```

```
-k K, --episodes=K    Number of epsiodes of the MDP to run (default 1)
-g G, --grid=G        Grid to use (case sensitive; options are BookGrid,
                      BridgeGrid, CliffGrid, MazeGrid, default BookGrid)
-w X, --windowSize=X  Request a window width of X pixels *per grid cell*
                      (default 150)
-a A, --agent=A       Agent type (options are 'random', 'value' and 'q',
                      default random)
-t, --text            Use text-only ASCII display
-p, --pause           Pause GUI after each time step when running the MDP
-q, --quiet           Skip display of any learning episodes
-s S, --speed=S       Speed of animation, S > 1.0 is faster, 0.0 < S < 1.0
                      is slower (default 1.0)
-m, --manual          Manually control agent
-v, --valueSteps      Display each step of value iteration
```

As we have covered in the lectures, the problem we are solving in GridWorld is the one of optimizing the total utility while moving through the map. The difficulty in this problem is that you have noise while moving, and the actions that you choose don't need to necessarily take you to the same place. In the GridWorld, this manifests in the way that you have a fixed chance of $1 - n$ to take the action that you chose, and a chance of n to move randomly.

As usual, the states are defined by coordinates (x,y), while the transitions are defined by the sides of the world towards which the agent is moving (south, east, north, west).

The lab assignment consists of four subtasks, and the correct implementation of each one of them carries the same weight. To ease the grading, each of the subtasks carries five points, with a total sum of 20. Your final points will be scaled and the actual maximum for the third lab assignment is 6.25.

Problem 1: Value iteration (3 points)

In the file valueIterationAgents.py fill in the code missing to implement value iteration. A reminder from the lectures - value iteration is an algorithm that tries to calculate the utility of every state of the map by using the recursive Bellman equation:

$$V_0(s) = 0$$
$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V_k(s')]$$

As a parameter to the constructor of your *ValueIterationAgent* you receive an argument *mdp*, which is the definition of the underlying Markov Decision Process. In the case of GridWorld, this is the implementation of the abstract class *mdp.MarkovDecisionProcess* which is located in *gridworld.py*. The methods that describe the Markov Decision Process are:

`mdp.getStates():`

Returns the list of all possible states as a list of tuples (int, int)

`mdp.getPossibleActions(state):`

Returns the list of strings representing all possible actions for a given state

`mdp.getTransitionStatesAndProbs(state, action):`

Returns a list of pairs(tuples) consisting of (nextState: tuple(int, int), transitionProbability: float)

`mdp.getReward(state, action, nextState):`

Returns the float precision reward for a given state, action, nextState triple

`mdp.isTerminal(state):`

Checks if a state is final (returns True or False). Final states don't have any transitions (so you should handle the case where the list of transitions is empty)!

The methods you need to complete are: *__init__*, *computeQValueFromValues*, *computeActionFromValues*. SInce you have pre-defined values of the transition and reward functions, most of your computation should be done in the initialization. While solving use the already initialized class *util.Counter*, which is an extension of the Python's default dictionary with default values of all keys set to zero - so you don't need to initialize the value for each state.

Note 1.1 Make sure that while updating the values you use a helper structure where you will store the intermediate result - in the stap $k+1$ of value iteration, the values V_{k+1} depend only on the values from the previous step V_k - make sure you don't use the values that you have calculated in step $k+1$.

In order to test your implementation, the result of running the following command:

```
python gridworld.py -a value -i 5
```

should look exactly like this:

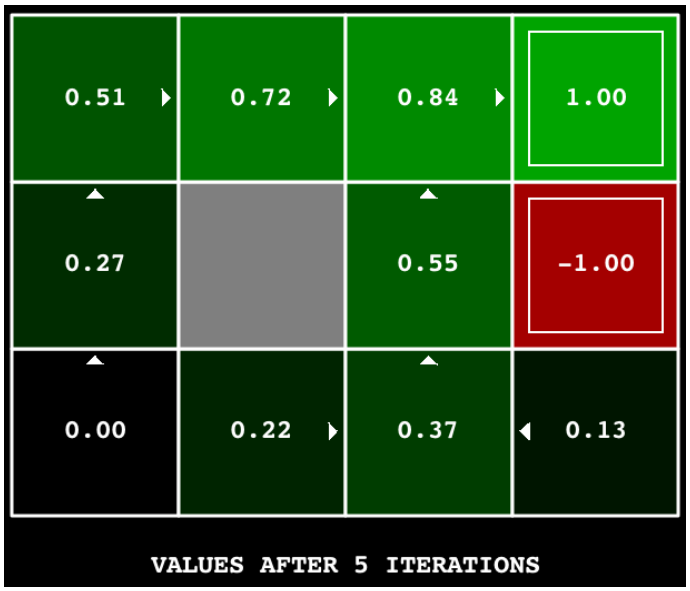

Figure 1: Value iteration example

Problem 2: Q-learning (3 points)

In the file qlearningAgents.py, fill in the code for the Q-learning algorithm. A reminder from the classes, the Q-learning algorithm works on the running average principle, and the update formula looks as follows:

$$Q_0(s,a) = 0$$
$$Q_{k+1}(s,a) \leftarrow Q_k(s,a) + (\alpha)[R(s,a,s') + \gamma \max_{a'} Q_k(s',a') - Q_k(s,a)]$$

Where α is the learning rate, while γ is the decaying factor.

The methods you need to fill in in the QLearningAgent class are: *__init__* , *getQValue*, *computeValueFromQValues*, *computeActionFromQValues*, *getAction* and *update*. The QlearningAgent class has variables that it inherits from the ReinforcementAgent class, which even though you can't see explicitly, exist. The variables that were inherited, and you will need in the scope of the assignment are: *self.epsilon* - exploration probability for the ϵ-greedy approach (Only in assignment 3.), *self.alpha* - learning rate, and *self.discount* - the decaying factor. All the falues are in float precision

Another useful function which is not explicitely seen is `self.getLegalActions(state)`, which returns the list of possible actions for a given state.

Note 2.1 Take care of the fact that if you visited just one Q-state of a state, and it has a negative value, the optimal move can be one of the moves that you have not explored yet (due to the implicit default value of zero). Take care to explicitly initialize the moves so you can handle these cases. In case of ties in the Q-values of multiple Q-states, you should solve the ties by taking a random action from the ones with the maximum values. For this, the method `random.choice()` is useful, as it takes a list of elements as an argument and returns a random one with uniform probability.

Note 2.2 Take care of the fact that when accessing Q-values, you use the method`getQValue`, since it is used later on in approximate Q-learning.

Note 2.3 You can use tuples as keys to your dictionaries and Counters!

Problem 3: ϵ-greedy (3 points)

Modify your Q-learning algorithm by implementing the ϵ-greedy approach. As a reminder, the ϵ is a small probability of taking a random action while learning the Q-values. The probability ϵ is available as a class variable *self.epsilon*, while the method util.flipCoin(p), might also prove useful, as it returns True with probability p, and False with probability $1 - p$.

Without any additional changes to your code after adding the ϵ-greedy approach, you can test your implementation by running the crawler and playing around with the parameters:

```
python crawler.py
```

Problem 4: Approximate Q-learning (3 points)

Implement approximate Q-learning in the file qlearningAgents.py in the class *ApproximateAgent*.

As a reminder, an approximate Q-value is the form:

$$Q(s,a) = \sum_{i=1}^{n} f_i(s,a) w_i$$

Where W is the weight vector, while $f(s, a)$ is the feature vector for a Q-state (s,a). Every weight w_i from the vector is mapped to one feature f_i. The functions that will calculate the features are already defined and implemented in the file *featureExtractors.py*, and the class variable *self.featExtractor* which is available in your ApproximateAgent has a method *getFeatures(state, action)* which returns a Counter with feature values for a given Q-state.

At the beginning of the process, you should initialize the weight to zero (hint: use the Counter), and then update them by using the formula from the classes:

$$w_i \leftarrow w_i + \alpha \cdot difference \cdot f_i(s, a)$$
$$difference = (r + \gamma \max_{a'} Q(s', a')) - Q(s, a)$$

If your implementation works, the learned pacman controller should without any problems solve the following map:

```
python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor
-x 50 -n 60 -l mediumGrid
```

While with a bit longer training time, he should have no more problems with a larger one:

```
python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor
-x 50 -n 60 -l mediumClassic
```