Artificial Intelligence – Programming Assignment 3

UNIZG FER, academic year 2016/17

Handed out: 10.5.2016. Due: 21.5.2016. at 23.59.

Introduction

In this lab assignment you will solve problems from the domain of reinforcement learning and machine 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.

The lab assignment consists of two independent components – a naive Bayes classifier and reinforcement learning algorithms. Each of the components is stored in a separate subfolder. The code for the assignment consists of Python files, a part of which you just need to read and understand, a part of which you need to modify yourselves and a part that you can ignore.

Task 1: Naive Bayes classifier (12 points)

After you download and unpack the zip archive and position yourself in the naive Bayes folder, you will see a list of files and folders which we briefly describe below:

Files you will edit:	
naiveBayesClassifier.py	Implementation of the naive Bayes classifier
Files you should study:	
pacman.py	Logic of how Pacman's world works
util.py	Useful helper functions and classes
dataLoader.py	Data input/output and feature extraction
classifier.py	The main file which starts the classifier and runs tests
Helper files you can ignore:	
layout.py	Description of the layouts of maps in the Pacman world
game.py	Model of the game

In the scope of the first task, the autograder isn't available, however automatic compatibility tests are availbable.s The fixed outputs of an already implemented model are hard-coded at the top of the *classifier.py* file, and the tests can evaluate an implemented model by running `python classifier.py -t 1`, as a result of which the absolute difference from the correct *a posteriori* probabilities will be logged. We emphasise that passing the automatic testing does not mean you will receive all the points on the task, as the outputs are checked only for 5 samples from the training and testing set. However, they are a good indicator of your model working well.

The naive Bayes classifier is defined in the style of a popular machine learning framework, scikit-learn. Every machine learning model implemented in scikit-learn accepts *hyperparameters* through its constructor and contains a *fit* method, which learns the parameters of the model on the training data, as well as a *predict* method, which predicts the output values for a set of new and unseen values.

The problem which we will solve is predicting Pacman's future moves. In the *pacmandata* directory you can find compressed files which contain samples of the current "GameState" as well as the next move that Pacman made. The samples are split in train and test sets, and come from four different agent implementations: a hungry Pacman that always moves towards the closest food (prefix *food*), a scared Pacman which never moves (prefix *stop*), a brave, but not bright Pacman which always moves towards the closest ghost (prefix *suicide*) and a Pacman which through reinforcement learning successfully navigates the map, avoids the ghosts and eats food (prefix *contest*).

As you will see in the second part of the lab assignment, learning behavior from zero is a very difficult process – a number of games needs to be played until Pacman's behavior adapts to the game and he starts consistently winning. This can be mitigated through imitation learning, and we can learn our behavior just by following an alrady pre-trained Pacman and attempting to imitate the choices he made, without actually playing the game.

The largest problem with learning imitation based models is that the actual criteria for making the choices is not available to us – we can observe the world state as well as the actions that the agent takes, however the features are unknown. We hope that through enough observation we will overcome this problem and learn the important features. The first part of the assignment is already implemented for you, and some features indicative of the next move have been extracted from the stored game states.

In the directory *classifier_data* you can find extracted features in the tab-separated values *('.tsv')* format which is compatible with the classifier. The first line, so called *header* contains the names of the features considered important for the decision, while the last column contains the target variables – the choice that the agent made for each input. Only two features have been extracted for every possible move – did we eat any food by making that move and did we reduce the distance to the closest ghost by making that move. These features are extracted for every possible future move we can make.

The example of the data found in the file *'contest_training.tsv'* can be seen in the folllowing table:

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 points)

In the *naiveBayesClassifier.py* file a skeleton of a naive Bayes classifier class has been implemented, however it is missing a few key methods. Your task in the first part of the lab assignment is to complete the implementations of these methods and create a fully functional classifier.

Let's remind ourselves of the Bayes rule and assume that we have a dataset from the Pacman universe, and are looking for the most probable move that some observed agent is going to make:

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

Where h_i is our hypothesis, or every possible move i, $P(h_i)$ is the **a priori** probability of hat move, $P(h_1|data)$ the **a posteriori** probability of a move given data, and $P(data|h_1)$ the **likelihood** of the data sample given the move. $P(data)$ is the probability of the data sample, and serves just as a normalisation constant.

In our example, the hypothesis are the moves that Pacman can make (West, North, ...) and the data is the **features** gathered from the world state – foodEaten_East, foodEaten_North, ... based on which Pacman makes his decision. The naive assumption of the naive Bayes classifier is that all the features contained in a single data sample are independent of each other, which allows us to write the likelihood as a product of the probabilities for each feature!

The naive Bayes classifier chooses the hypothesis with the maximum **a posteriori** probability as the correct one. In short, the naive Bayes classifier can be formulated in the following way:

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

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

Where f_j are the features, and h_i the hypothesis (target variable), Eliminating the normalization constant is allowed due to it not affecting the maximization, and the last formula is the classifier that you will need to implement.

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

Inside the *fit* method of ourclassifier, it is required to learn the likelihoods (conditional probabilities of each feature) $P(f_j|h_i)$ and the a priori probabilities for each hypothesis $P(h_i)$ based on our input data. The learned values should be stored in the class variables `self.prior` and `self.conditionalProb`. Also, you need to implement Laplace smoothing with the factor k which is provided as a hyperparameter of the algorithm in the constructor. $k = 0$ indicates no smoothing.

The input data to the fit method are two lists – *trainingData* and *trainingLabels* which contain the sequence of observations in the training set. The observations are pairs of (features, target), where for each instance of features we know the correct target label.

TrainingData is a list of dictionaries, where the keys are the names of the features (ex. foodEaten_East). while the values are exactly the values of those features in that training instance (ex. False). TrainingLabels is a list of strings which represent the choices that were made (ex. West).

Hint: Notice that it is necessary to keep track of how many times every specific value of every feature has occurred for a given class (ex: how many times foodEaten_East=True has occured for the target West).

Hint: inside the class variable `self.featureValues` you can find every possible value of every feature. FeatureValues is a dictionary where the keys are the feature names and the values are sets of all possible values the feature can have.

`NB.calculateJointProbabilities(self, instance):`

Inside the *calculateJointProbabilities* method you have to compute the **a posteriori** probabilities for each possible target (hypothesis). The aforementioned method is called in a loop from the *predict* method, where the actual classification choice is made based on the a posteriori probabilities for each possible choice.

The input to the method is a single instance in the form of a dictionary, where the keys are the names of features, and the values are the values of the features for that instance. The output of the method has to be a dictionary (subclass util.Counter), which will contain the a posteriori probabilities as the values on the locations of each possible hypothesis (`self.legalLabels`)

`NB.calculateLogJointProbabilities(self, instance):`

The *calculateLogJointProbabilities* method implements the same functionality of computing the a posteriori probability for each possible choice, however it uses the log trick for numerical stability. Your task is to implement the a posteriori probability computation with the log trick, and check if you can deduce which of the results hardcoded in the *classifier.py* class belong to the log transformed model, and which belong to the regular one. Think and argument (first to yourself, and later to us while submitting the assignment) why the log transform is a good choice in this case.

The inputs and outputs are the same as in the *calculateJointProbabilities* method.

After successfully implementing all methods, your classifier should pass all tests and have the following performance on the *contest* dataset:

```
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 a performance test of your model on a dataset can be ran through the main file `classifier.py`, for which the instructions can be seen 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 laplace 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
                        Laplace 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 you download and unpack the zip archive and position yourself in the reinforcement learning folder, you will see a list of files and folders which we briefly describe below:

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 task 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.

In the scope of the second task, you can use the autograder, which will give you adequate feedback regarding the correctness of your implementation. You run the autograder by typing `python autograder.py`. If you want to run the autograder for a specific question, do it as follows: `python autograder.py -q q1`.

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 (5 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:

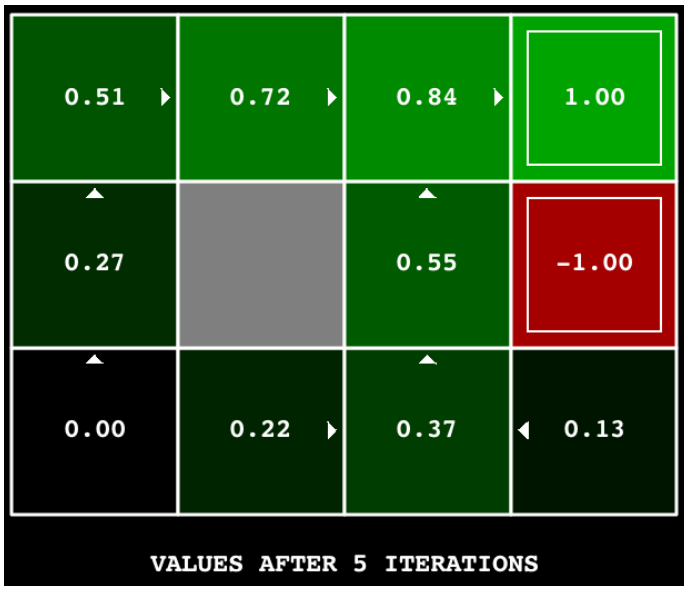

Figure 1: Value iteration example

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
```