Pacman - Coursework 1

Olivier Roncalez

February 2018

1 Introduction

The aim of this coursework was to use use supervised learning classification to allow Pacman to learn the best set of moves based on previous data generated when Pacman had previously played (and sometimes won) the game.

The Pacman AI projects were developed at UC Berkley[1] by John DeNero (denero@cs.berkeley.edu) and Dan Klein (klein@cs.berkeley.edu). The ClassifierAgent used for this coursework found in the classiferAgents.py file were written by Simon Parsons, based on code in pacmanAgents.py, and extended by Olivier Roncalez to include the classification algorithm with accompanying metrics.

The array of training data provided consisted of 25 binary (0-1) features based on information pertaining to the pacman Game State. The target classification was composed of 4 classes (0-3), and related to the particular direction Pacman moved given a Game State. As such, the aim of this project was to write an accurate classifier that could be learned on the data, and generalize effectively on new instances in order to classify them and help Pacman select an appropriate move.

2 Exploratory Phase

The initial exploratory phase consisted of choosing between a set of supervised learning algorithms that would be effective in learning the data and generalizing to new instances. Based on current knowledge of supervised learning algorithms, this included the possibility of Decision Trees (and/or Random Forests), Naive Bayes, and k-Nearest Neighbors classification algorithms.

Prior to building my own selected algorithm, scikit-learn algorithms [1] were tested on the data in python, in order to determine both accuracy and generalizability (using both 10-fold cross validation, and a test/train split of 0.2). The results of these classifiers are below:

[1] http://ai.berkeley.edu.

Classifier	Training Accuracy	Train/Test Split (0.2)	10-Fold Cross Validation
kNN (k = 1)	0.85	0.64	0.77
kNN (k = 5)	0.82	0.72	0.77
Naive Bayes	0.85	0.76	0.82
Decision Tree	0.87	0.68	0.79

Table 1: General Scikit-Learn Metrics

Both 10-fold cross validation and train/test split were used to assess the generalizability of the classifier to new instances. While train/test split avoided the increase in variance in the error estimate introduced by 10-fold cross validation, it restricted the amount of data used for learning, something which the latter technique minimized.

Results indicated that while all classifiers performed well in terms of training accuracy (accuracy determined by using the generated model to re-classify the training data), Naive Bayes was best in terms of generalizability, obtaining scores of 0.76 and 0.82 on a 0.2 train test split, and 10-fold cross validation, respectively. As such, the decision to build a Naive Bayes classifier was made.

3 Code

In order to build an optimal Naive Bayes classifier, two decisions were made. The first was to code my own k-fold cross validation and train/test split algorithms, in order to compare my model against the built in scikit-learn. The second was to assess my algorithm and refine it until it had highly comparable accuracy metrics with scikit-learn's classifier, with similar confusion-matrix results.

3.1 K-Fold Cross Validation Algorithm

The K-Fold algorithm was done by creating a function which took in 3 parameters: the learning data, the learning target, and K specifying the number of cross folds to take. The algorithm first generated an array of integers of same length as the target data. This list was then shuffled and used to arbitrarily re-order the learning data and target (while maintaining their respective positions in regards to each other). Data was then split into training and testing sets by using list comprehensions to split the data into K folds, and selecting K-1 as training, with the last fold reserved for testing. The classifier was then learned on the learning data and tested on the fold reserved for testing. The score was appended to an arbitrary list and the process repeated K times, with the average accuracy score returned.

3.2 Train-Test Split

The train/test split algorithm was virtually identical to the k-folds algorithm with the exception that the data was only split once, with a user-specified percentage of the data held for testing (0.2 in this case).

3.3 Naive Bayes

3.3.1 Theory

The algorithm for Naive Bayes attempts to calculate the probability of a hypothesis (in this case, class), based on the observed data. The formula for this equation according to Bayes Theorem is:

$$p(h|D) = \frac{p(D|h)p(h)}{p(D)} \tag{1}$$

The aim of the algorithm is to find the hypothesis (i.e., class) which maximizes this probability. This is the maximum a posteriori hypothesis.

$$h_{MAP} = \arg\max_{h \in H} p(h|D) \tag{2}$$

In using Naive Bayes, we make two assumptions: that the probability of observing the data is a normalizing constant which can be removed from the equation, and that each attribute is conditionally independent given the target:

$$p(D|h) = p(a_1, a_2, ..., a_n|\mathrm{v}_i) = \prod_j p(a_j|\mathrm{v}_i) \tag{3}$$

This allows us to rewrite (1) as :

$$\mathrm{v}_{NB} = \arg\max_{\mathrm{v_i} \in \mathrm{V}} \; p(\mathrm{v}_i) \prod_j p(a_j|\mathrm{v}_i) \tag{4}$$

As such, in Naive Bayes the hypothesis is the particular values of $p(\mathrm{v}_i)$ and $p(a_j|\mathrm{v_i})$, with the data being all the examples from which we compute those values. In order to train the classifier, we would need to compute all of these probabilities, which is exactly what the algorithm aims to do.

There is one issue which needed to be taken into account when computing the probabilities: the learning data is significantly sparse. This is to say that while we would generally estimate the probabilities of $p(a_j|\mathrm{v_i})$ by $\frac{n_c}{n}$ where n is the total number of counts of class v_i and n_c is the total number of cases of a particular attribute value for a given attribute belonging in the class v_i, certain values of $\frac{n_c}{n}$ may be close to 0. In order to prevent over-fitting the data, the distribution was estimated using a particular smoothed version of maximum likelihood (i.e., relative frequency counting). The new regularized estimate was thus calculated as below, and formed the basis for my algorithm:

$$p(a_j|\mathrm{v_i}) = \frac{n_c + \alpha}{n + \alpha m} \tag{5}$$

where m is the number of features, and α is the smoothing prior. Setting $\alpha = 1$ is called Laplace smoothing, and was the selected value in my algorithm. As such, all conditional probabilities were estimate according to equation (5).

3.3.2 Algorithm

The Naive Bayes training algorithm was implemented by defining 3 separate functions. The first formed the basis for creating the model based on the training data. In order to do this, the function took in two parameters: the training data and associated target value. The first step was to calculate the number of instances each class appeared in the training data, which was stored in a list. An empty dictionary was created with the class as key, in order to split the data according to their respective classification. Once the data split, each conditional probability was estimated according to equation (5), and stored in a separate dictionary.

The second function was defined as the actual Naive Bayes testing algorithm. This function took in a single parameter, the new instance to be classified, and calculated the probability of the data for each class, selecting the arg max class as the classification decision according to equation (4). To do this, an empty dictionary was created in order to store the results of equation (4) for each class. By creating an outer loop for each class, and an inner loop for each attribute, the probabilities were estimated by multiplying the probability of observing each attribute value (independently of all others) for a given class, with the probability of observing that given class. All resulting probabilities (1 for each class) were saved in the dictionary and the class whose probability was maximized, was selected. This class was then translated back into a direction based on the convertNumberToMove function which was provided. This value was stored in an object called 'bestmove', which indicated the ideal choice, should this choice be possible given the game constraints.

Finally, the last function consisted of a scoring algorithm which determined the accuracy of the classifier on the data. This function took in both the testing data and target (2 parameters). First the function created an empty confusion matrix, and estimated the classification for each instance in the test data according to the Naive Bayes algorithm. Next, the confusion matrix was populated by using the resulting values for both the prediction, and actual values, as index in the two dimension confusion matrix, incrementing the value by 1 for each run of the for loop. Values on the diagonal (i.e., when both predicted and actual class values were the same and thus formed the same index for the confusion matrix) represented accurate classification, while those in the off-diagonal were incorrectly classified. The resulting accuracy score was computed as an average

of correctly classified instances over total number of instances.

In order to move Pacman utilized the classification decision which was provided by the classifier and stored in the object 'bestmove'. This decision was then assessed in the getAction function in order to determine if it was a legal move. If so, the action was passed and Pacman moved. If not, Pacman was programmed to then make a random choice within the legal options and try the classification again the following turn.

3.4 Results

My Naive Bayes algorithm was compared with scikit learn with results in table 2. For all metrics pertaining to my Naive Bayes classifier, my algorithms were used (accuracy score, train/test split, and 10-fold cross validation). For metrics pertaining to scikit learn's Naive Bayes algorithm the scikit learn metrics were used.

Classifier	Training Accuracy	Train/Test Split (0.2)	10-Fold Cross Validation
Naive Bayes (scratch)	0.83	0.72	0.76
Naive Bayes (scikit learn)	0.85	0.76	0.80

Table 2: Algorithm Metrics for Pacman

Results indicate that my algorithm performed with comparable similarity to the one used in scikit learn. Small differences in training accuracy are most likely due to subtle differences in the smoothing parameter used to estimate the values, along with the penalization of non-occurence of features used by scikit learn's function. Nevertheless, the differences did not appear significant. The confusion matrices of both my classifier and scikit learn's classifier were also examined and showed to provide very similar outputs indicating that they performed roughly the same. Differences in the 10-fold cross validation, and train/test split may be the result of differences in the way these algorithms were coded, and as such may be simply chance differences, rather than actual significant functional ones.

	Actual Class 0	Actual Class 1	Actual Class 2	Actual Class 3
Predicted Class 0	25	1	3	2
Predicted Class 1	1	28	0	6
Predicted Class 2	1	2	22	1
Predicted Class 3	1	2	1	30

Table 3: Personal Confusion Matrix

	Actual Class 0	Actual Class 1	Actual Class 2	Actual Class 3
Predicted Class 0	26	1	0	1
Predicted Class 1	1	29	2	1
Predicted Class 2	3	0	22	1
Predicted Class 3	2	6	1	30

Table 4: Scikit Learn Confusion Matrix

In terms of Pacman performance, the classification algorithm allowed Pacman to move, but the choices were not necessarily the best from a gaming point of view. In other words, Pacman would frequently run into ghosts, prefer to move in a location with no food rather than consume the capsules, and in general, never won the game. Furthermore, Pacman would sometimes get stuck in the right hand corner where the best moves based on the classifier told him to move left, then back right, in an infinite loop. Pacman performance is largely due to the information provided within the training data. Given that the attributes were binary, there is a strong possibility that there was a lack in possible discrimination between some of the many elements within the game. In other words, it may be impossible for Pacman to prioritize discrete elements such as food over capsules, or even avoiding the ghosts in general. Finer discrimination within the training data may have allowed Pacman to not only make better decisions, but eventually win the game.

References

[1] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.