Assignment 5

B351 / Q351

Comprehension Questions Due:
Tuesday, March 10th, 2020 @ 11:59PM

Initial Due:
Tuesday, March 17th, 2020 @ 11:59PM

1 Summary

- Gain a basic understanding of machine learning
- Implement entropy, information gain, and classification methods for a decision tree algorithm, in addition to seeing how to recursively build a decision tree from an input node.

We will be using Python 3, so be sure you arent using older versions. Code that will compile in Python 2.7 may not compile in Python 3. See our installation guide for help removing old versions and installing Python 3.

Please submit your completed files to your private Github repository for this class. You may not make further revisions to your files beyond the above deadline without incurring a late penalty.

This assignment must be completed **individually**. You must submit your own files to your own repository. Any shared code will not be tolerated and those involved will be subjected to the university's cheating and plagiarism policy. **If you discuss ideas, each person must write a comment at the top of the assignment file naming the students with whom ideas were shared.**

2 Background

Decision Tree Learning is a machine learning algorithm often used for classification and regression problems. Decision trees offer several advantages over other algorithms, such as ease of implementation and reasonable interpretability, while facing several drawbacks, such as a tendency towards overfitting. Our reasoning behind choosing decision trees as the subject for this assignment is that they require minimal mathematical background as compared to other machine learning methods (Linear Regression, Neural Networks, etc). In this assignment, we hope to provide you with some general background knowledge regarding machine learning, as well as give you a chance to work with a real, commonly used machine learning algorithm.

In order to discuss decision trees, it will help to have a common background on machine learning terminology. The following videos provide some background on machine learning, and we highly recommend you watch them. (Having this baseline of machine learning knowledge is a prerequisite for a good machine learning-related final project!)

1. Machine Learning Basics: What is Machine Learning? (7:51)
2. Introduction to Supervised Learning (12:29)

Decision trees can be used for both classification and regression problems, but for this assignment, we will be limiting our scope to classification. Even further, we limit the type of data which our algorithm can receive to categorical data.

How does a decision tree work? There are two main phases:

1. Learning the tree
2. Classifying data

2.1 Learning the Tree

Learning the tree is a recursive process. We start with a root node, which contains all the data points. We will use some criterion to decide which attribute to use to split the data into 2 subnodes. Repeat this process at each of the subnodes, using the data they receive from their parent node. The process continues until a node that has all the data of the same class is reached, or has identical attributes. Your job is to implement the criterion that decides what to split. For implementation details of the full algorithm itself to learn the tree, refer to the `train` function at the bottom of `a5.py`.

In the textbook, there are detailed explanations on how to build a decision tree. For further information, please refer to section 8.4 (page 184) in the book (you can find the book in syllabus).

2.2 Classifying Data

Once the decision tree is built, classifying new datapoints is trivial. Simply propagate that datapoint through the tree, and when you arrive at a leaf node, return the class of the most common data in that node.

3 Programming Component

In this assignment's programming component, instead of building a whole decision tree, you will just be implementing specific functions that are used in the creation of a decision tree.

3.1 Data Structures

You will utilize the following classes. You should read and understand this section **before** embarking on the assignment.

3.1.1 a5.py

To help with your implementations, we imported the `math` package so you can easily access mathematical functions. We also provide a useful function called `unique`, based on numpy's `unique` function, which takes an iterable and returns

- a list of the unique items from that iterable
- a corresponding list of the counts of the number of times each item appeared in the original iterable

Example: unique([5,3,5,2,3,4,2,5,4,2]) would return ([2,3,4,5] , [3,2,2,3])

You will also see the functions for `entropy` and `information_gain`. It is your job to implement these, and their criteria is explained under the Objectives section of this documentation.

3.1.2 OtherKey Class

This class is in the `a5.py` file. An OtherKey object is a special key that represents any unexpected value for an attribute in question. This will be useful in the `classify` method.

3.1.3 Node Class

This class is in the `a5.py` file. `Node` is a node in a decision tree. It keeps track of the attribute split for its children. It encapsulates the following data members:

- `attribute` - the attribute used to split at the Node. It is also a key used in a data point's dictionary.
- `children` - a dictionary mapping possible values of the above attribute to child nodes for recursive classification
- `classification` - the selected class for a leaf node. This will be None for non-leaf nodes.

The following are Node's provided object methods:

1. `__init__(self, attribute=None, children={}, classification=None)` - constructor for the Node class
2. `__repr__(self)` - provides a string-based representation of the tree under any Node, suitable both for a human to read and for a machine to use to reconstruct the tree.

3. `classify_point(self, point)` - classifies a given point. It is your job to implement this, and its criteria is explained under the Objectives section.

4. `train(self, points, labels)` - trains the decision tree. You won't need to do anything with this, but it will give you an idea on how the functions you'll be implementing will come together to create a decision tree.

3.1.4 KNN_Classifier Class

This class is in the `a5.py` file. This class contains all the needed methods for classification using the KNN algorithm. It encapsulates the following data members:

- `k` - the 'K' in K-NN. used to determine the number of neighbors to use.

The following are KNN_Classifier's provided object methods:

1. `__init__(self, k)` - constructor for the KNN_Classifier class

2. `calc_euclidean_distance(self, point_a, point_b) -` calculates the euclidean distance of two points. It is your job to implement this, and its criteria is explained under the Objectives section.

3. `get_top_label(self, top_k_labels)` - picks the most frequent label in a list of labels, generally the closest k labels. It is your job to implement this, and its criteria is explained under the Objectives section.

4. `classify_point(self, point, training_points, training_labels)` - classifies a given point using the training points and labels given. It is your job to implement this, and its criteria is explained under the Objectives section.

3.1.5 debug.py

This file aims to help you see whether or not your functions are correct and get the intended entropy, information gain, and classification values when they're run on the given examples. Further debugging is encouraged to be done on your own, but the grading tool will also provide some feedback as to what you might be missing. Simply run the main function at the bottom of the file to check your results compared to the intended answers.

3.2 Objective

As stated before, the actual creation of the decision tree has been implemented for you already. Your goal is to provide the implementations to the following functions:

1. `calc_entropy`

2. `calc_information_gain`

3. `Node.classify_point`
4. `KNN_Classifier.calc_euclidean_distance`
5. `KNN_Classifier.get_top_label`
6. `KNN_Classifier.classify_point`

to the following specifications:

3.2.1 calc_entropy(classifications)

Entropy is a concept from information theory, which in vague terms, describes the amount of information missing from a system. This function should use the information contained in the given classifications to determine the entropy value for a dataset in a decision tree. It should utilize the entropy formula (as seen in lecture), which involves the probability of each class occurring.

- `classifications`: a list of the classifications that are assigned to the points of a dataset

3.2.2 calc_information_gain(parent_classifications, classifications_by_val, val_freqs)

Information Gain is a metric for choosing decision tree splits, returning a value based on the difference between the entropy of the parent and the normalized entropy of all the children.

- `parent_classifications`: a list of the classifications for the points of the parent node
- `classifications_by_val`: a dictionary of each unique value of an attribute mapped to a list of the classifications of each data point with that unique value.
  Example: From the sunburn example in class, `classifications_by_val` for the attribute "Hair" would be {'Blonde':['Burn', 'None', 'Burn', 'None'], 'Brown':['None', 'None', 'None'], 'Red': ['Burn']}
- `val_freqs`: a dictionary of each unique value of an attribute mapped to the number of times that value occurs

It may be helpful to refresh on Python dictionary methods when implementing this function.

3.2.3 Node.classify_point(self, point)

Write a function that classifies a given point going through the decision tree process. It should:

1. If the node has a classification, return that classification

Otherwise:

1. Get the value of the node's attribute at the point
2. If this value maps to one of the node's child nodes, get the child node corresponding to the value
3. Otherwise, get the child node corresponding to `OTHER`
4. Recursively call `classify` on the child node using the provided point argument

- `point`: a dictionary of each attribute for the point mapped to the attribute's value at that point
  Example: From the sunburn example, if we chose "Annie" as our point, it would look like {'Hair':'Blonde', 'Height':'Short', 'Weight':'Average', 'Lotion':'No'}

3.2.4 KNN_Classifier.calc_euclidean_distance(self, point1, point2)

Write a function that calculates the euclidean distance between two given points.

- `point_a & point_b`: a 2-tuple of floats representing a point in the classifiers data set. Example: (x, y) where x,y are floats.

3.2.5 KNN_Classifier.get_top_label(self, top_k_labels)

Write a function that chooses the most frequent label from a list of labels.

- `top_k_labels`: a list of labels in the classifiers data set. When calling `pick_label`, this needs to be the closest k labels to the point you're trying to classify.

3.2.6 KNN_Classifier.classify_point(self, point, training_points, training_labels)

Write a function that classifies a given point in the given data set going through the K-nearest neighbors algorithm. It should:

1. Order the given sample points by distance, calling the `calc_euclidean_distance` function to find the distance from the sample point to the new point, `point`.
2. Find the k-nearest neighbors using the above ordering

3. Call `get_top_label` on the list of labels for the k-nearest neighbors in order to determine the most frequent occurring label.

4. Return the result of `pick_label` exactly.

- `point`: the point we are wanting to classify
- `training_points`: sample points from a data set. this is a list of 2-tuples representing (x, y) points.
- `training_labels`: labels of the training points. will be the same length `n` as `training_points`. for each i from 0 to n, `training_labels[i]` is the label of `training_points[i]`

4 Grading

Problems are weighted according to the percentage assigned in the problem title. For each problem, points will be given based on the criteria met in the corresponding table on the next page and whether your solutions pass the test cases on the online grader.

In addition, the students in the tool-assisted group will have their solutions assessed for clarity (15 points), control flow and time complexity (5 points), and making use of clean and suitable syntax features (5 points).

Finally, remember that this is an individual assignment.

4.1 calc_entropy (20%)

Criteria	**Points**
Counts how many times each class in `classifications` occurs	5
Gets the total number of classifications	2
Goes through each count and gets the probability distribution	5
Returns the entropy using the entropy formula used in lecture, utilizing the probability distributions found	8
Total	20

4.2 clac_information_gain (20%)

Criteria	**Points**
Gets the total number of data points	2.5
Gets the weighted entropy by going through each value and their classifications (a) gets the entropy at that value (the child entropy) (b) finds the probability of that value occurring	5 5
Gets the entropy of the parent	2.5
Returns the information gain using the information gain formula used in lecture	5
Total	20

4.3 classify_point (10%)

Criteria	**Points**
If the node has a classification, returns that classification	2
Otherwise, Gets the value of the node's attribute at the point	2
If this value maps to one of the node's child nodes, get the child node corresponding to the value	2
Otherwise, get the child node corresponding to `OTHER`	2
Recursively calls `classify` on the child node using the provided point argument.	2
Total	10

4.4 KNN_Classifier (40%)

4.4.1 calc_euclidean_distance (10%)

Criteria	**Points**
Returns the euclidean distance between the two given points	10
Total	10

4.4.2 get_top_label (10%)

Criteria	**Points**
Returns one of the most frequent labels	8
Always returns the same value given the same top_k_labels list	2
Total	10

4.4.3 classify_point (20%)

Criteria	**Points**
Calls calc_euclidean_distance on each sample point with the classification point	5
Calls get_top_label on closest k labels	10
Returns the label determined by get_top_label	5
Total	20

5 Bonus (10%)

So far, we have worked with discrete variables as the only attributes for our decision tree. This is data that can only take a particular value, and has no grey area between it's choices. However, another type of data commonly seen in machine learning / AI practices is called continuous data, or data that can occupy any value over a continuous range. A good example of this would be age, as a person can have an age anywhere between 1 and around 125. Decision trees deal with continuous variables by choosing a pivot number (for our age example, a good pivot would be 50) and splits the data on whether or not the attribute is above or below the pivot value.

5.1 Splitting

Since continuous variables are able to be almost any value, picking a good pivot value is important in order to make the best splits. Let's say we are trying to find the best split for age. We have 4 nodes, with ages [15, 30, 45, and 60]. To find the best pivot, we would first find the midpoints between each adjacent value in our data (in this case, our midpoints would be [22.5, 37.7, and 52.5]). One we have found the midpoints, we then evaluate each split using information gain, and whichever split gives best information gain on the training data is used.

5.2 Submitting

Currently, our program does not support the use of continuous variables. For this bonus assignment, you are tasked in making a new file called `a5_Bonus.py` that uses `a5.py` as a base but is able to work with continuous variables. Comments should be added to the file to indicate where changes were made in the base file, and the finished program should be pushed to your respective Github account. Email Kyle(kjyohler@iu.edu) to submit.