sheet1

October 27, 2017

1 Exercise Sheet 1: Python Basics

This first exercise sheet tests the basic functionalities of the Python programming language in the context of a simple prediction task. We consider the problem of predicting health risk of subjects from personal data and habits. We first use for this task a decision tree

adapted from the webpage http://www.refactorthis.net/post/2013/04/10/Machine-Learning-tutorial-How-to-create-a-decision-tree-in-RapidMiner-using-the-Titanic-passenger-data-set.aspx. For this exercise sheet, you are required to use only pure Python, and to not import any module, including numpy. In exercise sheet 2, the nearest neighbor part of this exercise sheet will be revisited with numpy.

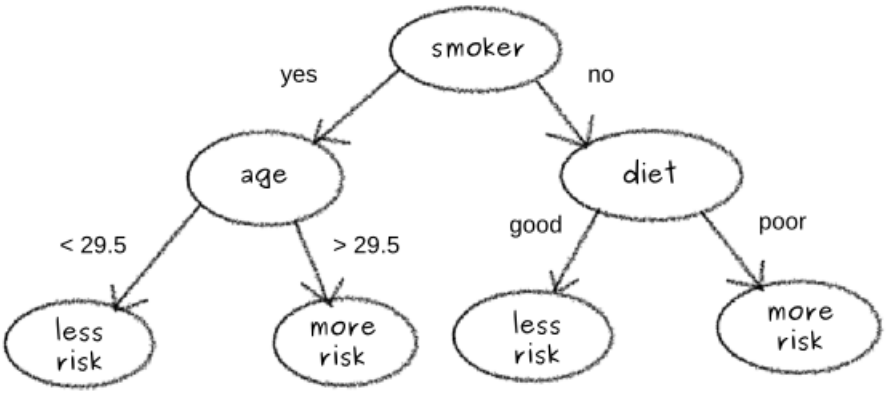
1.1 Classifying a single instance (15 P)

- Create a function that takes as input a tuple containing values for attributes (smoker,age,diet), and computes the output of the decision tree.
- Test your function on the tuple `('yes',31,'good')`,

```
In [1]: def classify(input_tuple):
            """Classify people's health status by comparing their age,
            smoking habits and diet."""
            smoker, age, diet = input_tuple
            if smoker == 'yes':
                if int(age) <  29.5:
                    return 'less'
                else:
                    return 'more'
            else:
                if diet == 'good':
                    return 'less'
                else:
                    return 'more'

        test_tuple = ('yes', 31, 'good')
        test = classify(test_tuple)
        assert test == 'more'
        answer = ((test_tuple, test))
        answer
```

```
Out[1]: (('yes', 31, 'good'), 'more')
```



1.2 Reading a dataset from a text file (10 P)

The file `health-test.txt` contains several fictious records of personal data and habits.

- Read the file automatically using the methods introduced during the lecture.
- Represent the dataset as a list of tuples.

```
In [26]: file_test = 'health-test.txt'
         health_test = []

         with open(file_test, 'r') as con:
             for line in con:
                 extention = line.strip().split(',')
                 extention[1] = int(extention[1])
                 health_test.append(tuple(extention))
         health_test

Out[26]: [('yes', 21, 'poor'),
          ('no', 50, 'good'),
          ('no', 23, 'good'),
          ('yes', 45, 'poor'),
          ('yes', 51, 'good'),
          ('no', 60, 'good'),
          ('no', 15, 'poor'),
          ('no', 18, 'good')]
```

1.3 Applying the decision tree to the dataset (15 P)

- Apply the decision tree to all points in the dataset, and compute the percentage of them that are classified as "more risk".

```
In [13]: health_tree = []
         for subject in health_test:
```

```
            health_tree.append(classify(subject))

        more_perc = float(health_tree.count('more')) / len(health_tree)
        print("predictions: %s" % health_tree)
        print("percentage of 'more': %s" % more_perc)
        print('------------------------------------------------------------')
        more_perc
```

```
predictions: ['less', 'less', 'less', 'more', 'more', 'less', 'more', 'less']
percentage of 'more': 0.375
------------------------------------------------------------
```

```
Out[13]: 0.375
```

1.4 Learning from examples (10 P)

Suppose that instead of relying on a fixed decision tree, we would like to use a data-driven approach where data points are classified based on a set of training observations manually labeled by experts. Such labeled dataset is available in the file `health-train.txt`. The first three columns have the same meaning than for `health-test.txt`, and the last column corresponds to the labels.

- Write a procedure that reads this file and converts it into a list of pairs. The first element of each pair is a triplet of attributes, and the second element is the label.

```
In [27]: file_train = 'health-train.txt'
         health_train = []
         with open(file_train) as con:
             for line in con:
                 content = line.strip().split(',')
                 content[1] = int(content[1])
                 health_train.append((tuple(content[:3]), content[-1]))

         print("train-set:\n")
         health_train
```

```
train-set:
```

```
Out[27]: [(('yes', 54, 'good'), 'less'),
          (('no', 55, 'good'), 'less'),
          (('no', 26, 'good'), 'less'),
          (('yes', 40, 'good'), 'more'),
          (('yes', 25, 'poor'), 'less'),
          (('no', 13, 'poor'), 'more'),
          (('no', 15, 'good'), 'less'),
          (('no', 50, 'poor'), 'more'),
          (('yes', 33, 'good'), 'more'),
```

```
 (('no', 35, 'good'), 'less'),
 (('no', 41, 'good'), 'less'),
 (('yes', 30, 'poor'), 'more'),
 (('no', 39, 'poor'), 'more'),
 (('no', 20, 'good'), 'less'),
 (('yes', 18, 'poor'), 'less'),
 (('yes', 55, 'good'), 'more')]
```

1.5 Nearest neighbor classifier (25 P)

We consider the nearest neighbor algorithm that classifies test points following the label of the nearest neighbor in the training data. For this, we need to define a distance function between data points. We define it to be

`d(a,b) = (a[0]!=b[0])+((a[1]-b[1])/50.0)**2+(a[2]!=b[2])`

where `a` and `b` are two tuples corrsponding to the attributes of two data points.

- Write a function that retrieves for a test point the nearest neighbor in the training set, and classifies the test point accordingly.
- Test your function on the tuple `('yes',31,'good')`

```
In [10]: def d(a, b):
             """Calculate distance metric."""
             return (a[0]!=b[0]) + ((float(a[1])-float(b[1]))/50)**2 + (a[2]!=b[2])

         def get_nearest_neighbor(target, train_set):
             dist = [d(target, x[0]) for x in train_set]
             return train_set[dist.index(min(dist))]


         neighborino = get_nearest_neighbor(test_tuple, health_train)
         ((test_tuple, neighborino[1]))

Out[10]: (('yes', 31, 'good'), 'more')
```

- Apply both the decision tree and nearest neighbor classifiers on the test set, and find the data point(s) for which the two classifiers disagree, and with which probability it happens.

```
In [12]: health_neighbor = []

         for subject in health_test:
             extention = get_nearest_neighbor(subject, health_train)
             health_neighbor.append(extention[1])

         print("Predictions by Decision Tree:\n%s" % health_tree)
         print("Predictions by Nearest Neighbor Algorithm:\n%s" % health_neighbor)

         indices = []
         different = []
```

```
        for i in range(len(health_tree)):
            if health_tree[i] != health_neighbor[i]:
                indices.append(i)
                different.append(health_test[i])

        difference_prob = float(len(different)) / len(health_tree)

        print("Index: %s\nDatapoint: %s" % (indices, different))
        print("Probability: %s" % difference_prob)
        print('--------------------------------------------------------------------------')
        ((different, difference_prob))
```

```
Predictions by Decision Tree:
['less', 'less', 'less', 'more', 'more', 'less', 'more', 'less']
Predictions by Nearest Neighbor Algorithm:
['less', 'less', 'less', 'more', 'less', 'less', 'more', 'less']
Index: [4]
Datapoint: [('yes', '51', 'good')]
Probability: 0.125
--------------------------------------------------------------------------
```

```
Out[12]: ([('yes', '51', 'good')], 0.125)
```

One problem of simple nearest neighbors is that one needs to compare the point to predict to all data points in the training set. This can be slow for datasets of thousands of points or more. Alternatively, some classifiers train a model first, and then use it to classify the data.

1.6 Nearest mean classifier (25 P)

We consider one such trainable model, which operates in two steps:

(1) Compute the average point for each class, (2) classify new points to be of the class whose average point is nearest to the point to predict.

For this classifier, we convert the attributes smoker and diet to real values (for smoker: yes=1.0 and no=0.0, and for diet: good=0.0 and poor=1.0), and use the modified distance function:
`d(a,b) = (a[0]-b[0])**2+((a[1]-b[1])/50.0)**2+(a[2]-b[2])**2`
We adopt an object-oriented approach for building this classifier.

- Implement the methods `train` and `predict` of the class `NearestMeanClassifier`.

```
In [15]: class NearestMeanClassifier(object):
             """Training Method that takes a dataset as input and produces two internal
             vectors corresponding to the mean of each class."""
             def train(self, dataset):
                 # Data preparation
                 self.classes = []
                 data = []
```

```
        for line in dataset:
            if line[1] not in self.classes:
                self.classes.append(line[1])

            X, y = line
            smoker, age, diet = X
            numeric_tuple = ((int(smoker == 'yes'),
                              int(age),
                              int(diet == 'poor')), y)
            data.append(numeric_tuple)

        self.class_averages = []

        for classes in self.classes:
            dict_content = [tuples for tuples in data if tuples[1] == classes]
            smoker_list = [tuples[0][0] for tuples in data if tuples[1] == classes]
            age_list = [tuples[0][1] for tuples in data if tuples[1] == classes]
            diet_list = [tuples[0][2] for tuples in data if tuples[1] == classes]

            mean_tuple = ((float(sum(smoker_list)) / len(smoker_list),
                           float(sum(age_list)) / len(age_list),
                           float(sum(diet_list)) / len(diet_list)), classes)
            self.class_averages.append(mean_tuple)

    def predict(self, dataset):
        # Define Distance-Metric
        def d(a, b):
            return (a[0] - b[0])**2 + ((a[1] - b[1])/50)**2 + (a[2] - b[2])**2

        def get_nearest_neighbor(target, train_set):
            dist = [d(target, x[0]) for x in train_set]
            return train_set[dist.index(min(dist))][1]


        self.predictions = []
        for line in dataset:
            smoker, age, diet = line
            line_numeric = (int(smoker == 'yes'),
                            int(age),
                            int(diet == 'poor'))

            prediction = get_nearest_neighbor(line_numeric, self.class_averages)
            self.predictions.append(prediction)
```

- Build an object of class `NearestMeanClassifier`, train it on the training data, and print the mean vector for each class.

In [17]:
```
avg_neighbor = NearestMeanClassifier()
avg_neighbor.train(health_train)
```

```
avg_neighbor.predict(health_test)

avg_neighbor.class_averages
```

```
('less', (0.3333333333333333, 32.111111111111114, 0.2222222222222222))
('more', (0.5714285714285714, 37.142857142857146, 0.5714285714285714))
```

```
Out[17]: [((0.3333333333333333, 32.111111111111114, 0.2222222222222222), 'less'),
          ((0.5714285714285714, 37.142857142857146, 0.5714285714285714), 'more')]
```

- Predict the test data using the nearest mean classifier and print all test examples for which all three classifiers (decision tree, nearest neighbor and nearest mean) agree.

```
In [23]: same_pred = []
         for i in range(len(health_test)):
             if health_tree[i] == health_neighbor[i] == health_avg_neighbor[i]:
                 same_pred.append(health_test[i])

         print("Coinciding Predictions:")
         same_pred
```

```
Coinciding Predictions:
```

```
Out[23]: [('no', '50', 'good'),
          ('no', '23', 'good'),
          ('yes', '45', 'poor'),
          ('no', '60', 'good'),
          ('no', '15', 'poor'),
          ('no', '18', 'good')]
```