sheet1 - solution group SSASCD

Karel Smejkal
Darya Smirnova
Mohammad Abdelkhalek
Michael Samuels
Julián Cantor
Marta Domagalska

October 20, 2017

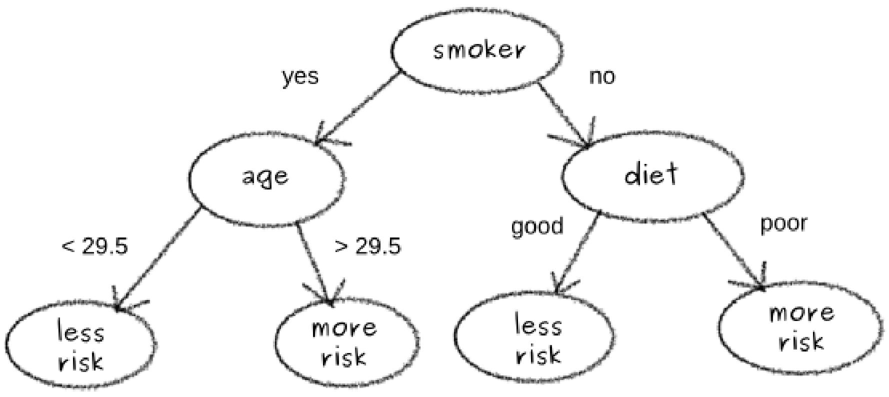
1 Exercise Sheet 1: Python Basics

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 exercise1(x):
            result = []
            if x[0] == 'yes':
                if x[1] > 29.5:
                    decision = 'more'
                else:
                    decision = 'less'
            else:
                if x[2] == 'poor':
                    decision = 'more'
                else:
                    decision = 'less'
            result.append(((x[0],x[1],x[2]),decision))
            return result
```



```
        exercise1(('yes',31,'good'))
```

```
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 [2]: def exercise2():
            f = open('health-test.txt','r')
            L = []
            for line in f:
                elem = line[:-1].split(',')
                #L += [(elem[0],float(elem[1]),elem[2])]
                if float(elem[1])%1 == 0:
                    age = int(elem[1])
                else:
                    age = float(elem[1])
                L.append((elem[0],age,elem[2]))
            f.close()
            return L

        exercise2()
```

```
Out[2]: [('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 [3]: def exercise3(dataSet):
            avr = 0
            percentage = 0
            for j in dataSet:
                if exercise1(j)[0][1] == 'more':
                    avr += 1
            return (avr/(len(dataSet)))

        exercise3(exercise2())

Out[3]: 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 [4]: def exercise4():
            f = open('health-train.txt','r')
            Dtriplet = []
            Dlabel = []
            for line in f:
                elem = line[:-1].split(',')
                if float(elem[1])%1 == 0:
                    age = int(elem[1])
                else:
                    age = float(elem[1])
                Dtriplet.append((elem[0],age,elem[2]))
                Dlabel.append((elem[3]))
            f.close()
            return(list(zip(Dtriplet, Dlabel)))

        exercise4()
```

```
Out[4]: [(('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 [5]: def exercise5a():
            #distance fuction based on the assigment
            def d(a,b):
                return (a[0]!=b[0])+((a[1]-b[1])/50.0)**2+(a[2]!=b[2])

            def getNeighbors(testPoint):
                f = open('health-train.txt','r')

                Dtriplet = []
                Dlabel = []
                for line in f:
                    elem = line[:-1].split(',')
                    Dtriplet.append((elem[0],float(elem[1]),elem[2]))
                    Dlabel.append((elem[3]))

                testSet = list(zip(Dtriplet, Dlabel))
                distances = []
                for x in range(len(testSet)):
                    dist = d(testPoint, testSet[x][0])
                    distances.append((testSet[x], dist))
                #sort the list appended by distance and than take only the lowest distance and s
                distances.sort(key=lambda tup: tup[1])
                neighbors = []
                result = []
                neighbors.append(distances[0][0])
                result.append((testPoint,neighbors[0][1]))
                return ((result))

            return(getNeighbors(('yes',31,'good')))

        exercise5a()
```

```
Out[5]: [(('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 [6]: def exercise5b():
            def decisionTree(x):
                    result = []
                    if x[0] == 'yes':
                        if x[1] > 29.5:
                            decision = 'more'
                        else:
                            decision = 'less'
                    else:
                        if x[2] == 'poor':
                            decision = 'more'
                        else:
                            decision = 'less'
                    result.append((x,decision))
                    return result

            #distance fuction based on the assigment
            def d(a,b):
                return (a[0]!=b[0])+((a[1]-b[1])/50.0)**2+(a[2]!=b[2])

            def getNeighbors(testPoint):
                f = open('health-train.txt','r')

                Dtriplet = []
                Dlabel = []
                for line in f:
                    elem = line[:-1].split(',')
                    Dtriplet.append((elem[0],float(elem[1]),elem[2]))
                    Dlabel.append((elem[3]))
                testSet = list(zip(Dtriplet, Dlabel))
                distances = []
                for x in range(len(testSet)):
                    dist = d(testPoint, testSet[x][0])
                    distances.append((testSet[x], dist))
                #sort the list appended by distance and than take only the lowest distance and s
                distances.sort(key=lambda tup: tup[1])
                neighbors = []
                result = []
                neighbors.append(distances[0][0])
                result.append((testPoint,neighbors[0][1]))
                f.close()
                return ((result))
            #testSet
            f = open('health-test.txt','r')
            L = []
            for line in f:
                elem = line[:-1].split(',')
                L.append((elem[0],float(elem[1]),elem[2]))

            L_notMatching = []
            for line in L:
                if decisionTree(line) != getNeighbors(line):
                    L_notMatching.append(line)
                    probability_notMatching = len(L_notMatching)/len(L)

            return((L_notMatching, probability_notMatching))

        exercise5b()
```

```
Out[6]: ([('yes', 51.0, '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 [7]: class NearestMeanClassifier:

            # Training method that takes as input a dataset
            # and produces two internal vectors corresponding
            # to the mean of each class.
            def train(self,dataset):
                def valuate(trainPoint):
                        if trainPoint[0][0] == 'yes':
                            smoke = 1.0
                        else:
                            smoke = 0.0
                        age = trainPoint[0][1]
                        if trainPoint[0][2] == 'poor':
                            diet = 1.0
                        else:
                            diet = 0.0
                        return((smoke, age, diet), trainPoint[1])

                L_valuated = []
                for trainPoint in dataset:
                    L_valuated.append(valuate(trainPoint))

                more_risk_data = []
                less_risk_data = []
                for line in L_valuated:
                    if line[1] == 'more':
                        more_risk_data.append(line[0])
                    else:
                        less_risk_data.append(line[0])

                more_risk_avg = tuple(sum(y) / len(y) for y in zip(*more_risk_data))
                less_risk_avg = tuple(sum(y) / len(y) for y in zip(*less_risk_data))
                self.means = ([(more_risk_avg, 'more'), (less_risk_avg, 'less')])

            # Prediction method that takes as input a new data
            # point and predicts it to belong to the class with
            # nearest mean.
            def predict(self,x):
                def valuate(trainPoint):
                    if trainPoint[0] == 'yes':
                        smoke = 1.0
                    else:
                        smoke = 0.0
                    age = trainPoint[1]
                    if trainPoint[2] == 'poor':
                        diet = 1.0
                    else:
                        diet = 0.0
                    return((smoke, age, diet))
                def mean_dist(mean_tuple, new_tuple):
                    new_tuple = valuate(new_tuple)
                    d = (mean_tuple[0][0] - new_tuple[0]) ** 2 + ((mean_tuple[0][1] - new_tuple[
                    return (d)

                j = ''
                L_predict = []
                for new_tuple in x:
                    if mean_dist(self.means[1], new_tuple) < mean_dist(self.means[0], new_tuple)
                        j = 'less'
                    else:
                        j = 'more'
                    L_predict.append((new_tuple,j))
                return(L_predict)
```

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

```
In [8]: def exercise6a():
            trainData = exercise4()
            c = NearestMeanClassifier()
            c.train(trainData)
            return c

        exercise6a().means

Out[8]: [((0.5714285714285714, 37.142857142857146, 0.5714285714285714), 'more'),
         ((0.3333333333333333, 32.111111111111114, 0.2222222222222222), 'less')]
```

- 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 [9]: def exercise6b():
            def decisionTree(x):
                result = []
                if x[0] == 'yes':
                    if x[1] > 29.5:
                        decision = 'more'
                    else:
                        decision = 'less'
                else:
                    if x[2] == 'poor':
                        decision = 'more'
                    else:
                        decision = 'less'
                result.append((x,decision))
                return result
            def d(a,b):
```

```
            return (a[0]!=b[0])+((a[1]-b[1])/50.0)**2+(a[2]!=b[2])
        def getNeighbors(testPoint):
            f = open('health-train.txt','r')

            Dtriplet = []
            Dlabel = []
            for line in f:
                elem = line[:-1].split(',')
                Dtriplet.append((elem[0],float(elem[1]),elem[2]))
                Dlabel.append((elem[3]))

            testSet = list(zip(Dtriplet, Dlabel))
            distances = []
            for x in range(len(testSet)):
                dist = d(testPoint, testSet[x][0])
                distances.append((testSet[x], dist))
            #sort the list appended by distance and than take only the lowest distance and s
            distances.sort(key=lambda tup: tup[1])
            neighbors = []
            result = []
            neighbors.append(distances[0][0])
            result.append((testPoint,neighbors[0][1]))
            return ((result))

        L = exercise2()
        c = exercise6a()

        for x in range(len(L)):
            nm = c.predict(L)[x][1]
            nn = getNeighbors(L[x])
            dt = decisionTree(L[x])

            if nn[0][1] == dt[0][1] == nm:
                print(dt)




    exercise6b()
```

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