MTL782
Assignment

Data Mining - Multiclass Classification

Submitted By:

Shruti Jain: 2019MT10726

Pragya Dhakar: 2019MT60756

Anurag Mittal: 2019MT10677

Data Set Used: Maternal Health Risk Data Set

(https://archive.ics.uci.edu/ml/datasets/Maternal+Health+Risk+Data+Set)

1) A Data description

The data is about the health risk of females during pregnancy based on their blood pressure, heart rate, and blood glucose level. Data has been collected from different hospitals, community clinics, maternal health cares in the rural areas of Bangladesh through the IoT-based risk monitoring system.

2) Benefits of using Data Mining

The data can help in pre-diagnosing the pregnant females that their health condition is good enough for giving birth or they need any kind of premedication to make their vitals normal before going into labor. This can help the patient in giving safe birth and reassure the doctor that the health of the patient will be alright after giving birth.

3) Further information about data

i) There were some data that did not align with our predictions this could be because of some noise present that is an error in collecting the data. The other reason is that every human body is different and it is very difficult to accurately predict the risk level by using a limited amount of dataset.

ii) We can try to fix the error by taking the data from different parts in India and different hospitals (this will improve our test and training sets), we can also introduce new attributes using the current attributes or introduce some new attributes that might be useful (this will improve our algorithm).

Code for Decision Tree

```
import pandas as pd
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn import metrics

data = pd.read_csv("Maternal Health Risk Data Set.csv")
X = pd.read_csv("Maternal Health Risk Data Set.csv", usecols=[i for i in
data.columns if i != 'RiskLevel' ])
y = data['RiskLevel'].values.reshape((-1, 1))
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=1)
clf = DecisionTreeClassifier()
clf = clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
print("Accuracy:", metrics.accuracy_score(y_test, y_pred))
```

Code for Random Forest

```
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn import metrics

data = pd.read_csv("Maternal Health Risk Data Set.csv")
X = pd.read_csv("Maternal Health Risk Data Set.csv", usecols=[i for i in
data.columns if i != 'RiskLevel' ])
y = data['RiskLevel'].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=1)
clf = RandomForestClassifier()
clf = clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
print("Accuracy:", metrics.accuracy_score(y_test, y_pred))
```

Code for KNN Classifier

```
import pandas as pd
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn import metrics

data = pd.read_csv("Maternal Health Risk Data Set.csv")
X = pd.read_csv("Maternal Health Risk Data Set.csv", usecols=[i for i in
data.columns if i != 'RiskLevel' ])
y = data['RiskLevel'].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=1)
clf = KNeighborsClassifier()
clf = clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
print("Accuracy:", metrics.accuracy_score(y_test, y_pred))
```

Code for Naïve Bayes Classifier

```
import pandas as pd
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import train_test_split
from sklearn import metrics

data = pd.read_csv("Maternal Health Risk Data Set.csv")
X = pd.read_csv("Maternal Health Risk Data Set.csv", usecols=[i for i in
data.columns if i != 'RiskLevel' ])
y = data['RiskLevel'].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=1)
clf = GaussianNB()
clf = clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
print("Accuracy:", metrics.accuracy_score(y_test, y_pred))
```

Code for comparing performances using K-fold Cross Validation

```
import numpy as np
import pandas as pd
from random import randrange
from sklearn import preprocessing
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB

import warnings

from sklearn.tree import DecisionTreeClassifier

warnings.filterwarnings('ignore')


def printMetrics(actual, predictions):
    assert len(actual) == len(predictions)
    correct = 0
    for i in range(len(actual)):
        if actual[i] == predictions[i]:
            correct += 1
    return correct / float(len(actual)) * 100.0


class kFoldCV:

    def __init__(self):
        pass

    def crossValSplit(self, dataset, numFolds):
        dataSplit = list()
        dataCopy = list(dataset)
        foldSize = int(len(dataset) / numFolds)
        for _ in range(numFolds):
            fold = list()
            while len(fold) < foldSize:
                index = randrange(len(dataCopy))
                fold.append(dataCopy.pop(index))
            dataSplit.append(fold)
        return dataSplit

    def kFCVEvaluate(self, dataset, numFolds):
        print('*' * 20)
        print("Using KNN Classifier")
        knn = KNeighborsClassifier()
        folds = self.crossValSplit(dataset, numFolds)
        scores = list()
        for fold in folds:
            trainSet = list(folds)
            trainSet.remove(fold)
            trainSet = sum(trainSet, [])
            testSet = list()
            for row in fold:
                rowCopy = list(row)
```

```
                testSet.append(rowCopy)

            trainLabels = [row[-1] for row in trainSet]
            trainSet = [train[:-1] for train in trainSet]
            knn.fit(trainSet, trainLabels)

            actual = [row[-1] for row in testSet]
            testSet = [test[:-1] for test in testSet]

            predicted = knn.predict(testSet)

            accuracy = printMetrics(actual, predicted)
            scores.append(accuracy)

        print('Scores: %s' % scores)
        print('\nMaximum Accuracy: %3f%%' % max(scores))
        print('\nMean Accuracy: %.3f%%' % (sum(scores) /
float(len(scores))))
        print('*' * 20)
        print('\n')

        print('*' * 20)
        print("Using Decision Tree Classifier")
        DT = DecisionTreeClassifier()
        folds = self.crossValSplit(dataset, numFolds)
        scores = list()
        for fold in folds:
            trainSet = list(folds)
            trainSet.remove(fold)
            trainSet = sum(trainSet, [])
            testSet = list()
            for row in fold:
                rowCopy = list(row)
                testSet.append(rowCopy)

            trainLabels = [row[-1] for row in trainSet]
            trainSet = [train[:-1] for train in trainSet]
            DT.fit(trainSet, trainLabels)

            actual = [row[-1] for row in testSet]
            testSet = [test[:-1] for test in testSet]

            predicted = DT.predict(testSet)

            accuracy = printMetrics(actual, predicted)
            scores.append(accuracy)

        print('Scores: %s' % scores)

        print('\nMaximum Accuracy: %3f%%' % max(scores))
        print('\nMean Accuracy: %.3f%%' % (sum(scores) /
float(len(scores))))
        print('*' * 20)
        print('\n')

        print('*' * 20)
        print("Using Random Forest Classifier")
        RF = RandomForestClassifier()
        folds = self.crossValSplit(dataset, numFolds)
        scores = list()
        for fold in folds:
```

```
            trainSet = list(folds)
            trainSet.remove(fold)
            trainSet = sum(trainSet, [])
            testSet = list()
            for row in fold:
                rowCopy = list(row)
                testSet.append(rowCopy)

            trainLabels = [row[-1] for row in trainSet]
            trainSet = [train[:-1] for train in trainSet]
            RF.fit(trainSet, trainLabels)

            actual = [row[-1] for row in testSet]
            testSet = [test[:-1] for test in testSet]

            predicted = RF.predict(testSet)

            accuracy = printMetrics(actual, predicted)
            scores.append(accuracy)

        print('Scores: %s' % scores)

        print('\nMaximum Accuracy: %3f%%' % max(scores))
        print('\nMean Accuracy: %.3f%%' % (sum(scores) /
float(len(scores))))
        print('*' * 20)
        print('\n')

        print('*' * 20)
        print("Using Naive Bayes Classifier")
        NB = GaussianNB()
        folds = self.crossValSplit(dataset, numFolds)
        scores = list()
        for fold in folds:
            trainSet = list(folds)
            trainSet.remove(fold)
            trainSet = sum(trainSet, [])
            testSet = list()
            for row in fold:
                rowCopy = list(row)
                testSet.append(rowCopy)

            trainLabels = [row[-1] for row in trainSet]
            trainSet = [train[:-1] for train in trainSet]
            NB.fit(trainSet, trainLabels)

            actual = [row[-1] for row in testSet]
            testSet = [test[:-1] for test in testSet]

            predicted = NB.predict(testSet)

            accuracy = printMetrics(actual, predicted)
            scores.append(accuracy)

        print('Scores: %s' % scores)

        print('\nMaximum Accuracy: %3f%%' % max(scores))
        print('\nMean Accuracy: %.3f%%' % (sum(scores) /
float(len(scores))))
        print('*' * 20)
        print('\n')
```

```
def readData(fileName):
    data = []
    labels = []

    with open(fileName, "r") as file:
        lines = file.readlines()
    for line in lines:
        splitline = line.strip().split(',')
        data.append(splitline)
        labels.append(splitline[-1])
    return data, labels


File = 'Maternal Health Risk Data Set.csv'

Data, Label = readData(File)
df = pd.DataFrame(Data)
df = df.apply(preprocessing.LabelEncoder().fit_transform)
Features = df.values.tolist()
Labels = [car[-1] for car in Features]
kfcv = kFoldCV()
print('*' * 20)
print('Maternal Health Risk Data Set')
print('\n')

kfcv.kFCVEvaluate(Features, 10)
```

Code for Apriori Algorithm

```
import pandas as pd


def DataSet(dataSet):
    C = []
    for tid in dataSet:
        for item in tid:
            if not [item] in C:
                C.append([item])
    C.sort()
    return list(map(frozenset, C))


def DataScan(dataSet, CK, minSupport):  # Accept candidate k itemsets and
output frequent k itemsets
    ssCnt = {}
    for tid in dataSet:
        for can in CK:
            if can.issubset(tid):
                if not can in ssCnt:
                    ssCnt[can] = 1
                else:
                    ssCnt[can] += 1
    numItem = float(len(dataSet))
    retList = []
    supportData = {}
```

```
    for key in ssCnt:
        support = float(ssCnt[key] / numItem)
        if support >= minSupport:
            retList.insert(0, key)
        supportData[key] = support  # Update support dictionary
    return retList, supportData
    retList = []
    lenLK = len(LK)
    for i in range(lenLK):
        for j in range(i + 1, lenLK):
            L1 = list(LK[i])[:k - 2]
            L2 = list(LK[j])[:k - 2]
            L1.sort()
            L2.sort()
            if L1 == L2:
                retList.append(LK[i] | LK[j])
    return retList


def aprioriGen(LK, k):  # Create candidate item set CK, where k is the
number of elements in the output set

    retList = []
    lenLK = len(LK)
    for i in range(lenLK):
        for j in range(i + 1, lenLK):
            L1 = list(LK[i])[:k - 2]
            L2 = list(LK[j])[:k - 2]
            L1.sort()
            L2.sort()
            if L1 == L2:
                retList.append(LK[i] | LK[j])
    return retList


def apriori(dataSet, minSupport):
    C1 = DataSet(dataSet)
    D = set()
    for tid in dataSet:
        tid = frozenset(tid)
        D.add(tid)
    L1, supportData = DataScan(D, C1, minSupport)
    L = [L1]
    k = 2
    while (len(L[k - 2]) > 0):
        CK = aprioriGen(L[k - 2], k,)
        LK, supK = DataScan(D, CK, minSupport)
        supportData.update(supK)
        L.append(LK)
        k += 1
    L = [i for i in L if i]  # Delete empty list
    return L, supportData


name = 'menu_orders.txt'
minSupport = 0.2
minconfidence = 0.5


def createData(name):  # Preprocess the data and output the dataset
```

```
    D = pd.read_csv(name, header=None, index_col=False, names=['1', '2',
'3'])

    D = D.fillna(0)
    D = D.values.tolist()
    for i in range(len(D)):
        D[i] = [j for j in D[i] if j != 0]
    return D


def calculate(dataset):  # Calculation of support and confidence by
algorithm
    dataset, dic = apriori(dataset, minSupport)

    Rname = []
    Rsupport = []
    Rconfidence = []
    emptylist = []
    for i in range(len(dataset)):
        for AB in dataset[i]:
            for A in emptylist:
                if A.issubset(AB):
                    conf = dic.get(AB) / dic.get(AB - A)
                    if conf >= minconfidence:
                        Rname.append(str(AB - A) + '-->' + str(A))
                        Rconfidence.append(conf)
                        Rsupport.append(dic.get(AB))

            emptylist.append(AB)
    return Rname, Rsupport, Rconfidence


def outputdata(Rname, Rsupport, Rconfidence):
    data = {

        "Association rules": Rname,
        "Support": Rsupport,
        "Confidence": Rconfidence
    }
    df = pd.DataFrame(data,
                      columns=['Association rules', 'Support',
'Confidence'])

    return df


dataset = createData(name)
R1, R2, R3 = calculate(dataset)
df = outputdata(R1, R2, R3)
df.to_csv('Report.txt')
```

TXT file used above: menu_orders.txt

```
a,c,e
b,d
b,c
a,b,c,d
a,b
b,c
a,b
```

```
a,b,c,e
a,b,c
a,c,e
```

Code for FP-Growth

```
# import pandas as pd

class node:
    def _init_(self, word, word_count=0, parent=None, link=None):
        self.word = word
        self.word_count = word_count
        self.parent = parent
        self.link = link
        self.children = {}

    # tree traversal
    def visittree(self):
        #        if self is None:
        #            return None
        output = []
        output.append(str(vocabdic[self.word]) + " " +
str(self.word_count))
        if len(list(self.children.keys())) > 0:
            for i in (list(self.children.keys())):
                output.append(self.children[i].visittree())
        return output


'''      Build FPTREE class and method       '''


class fptree:
    def _init_(self, data, minsup):
        # raw data and minminual support
        self.data = data
        self.minsup = minsup

        # null root
        self.root = node(word="Null", word_count=1)

        # each line of transaction with new order from the most frequent
items to less
        self.wordlinesort = []
        # node table containing link of all nodes of same word
        self.nodetable = []
        # dictionary contaiing word more than the minsupport count with des
order
        self.wordsortdic = []

        # dictionaly containing word and the support count
        self.worddic = {}
        # dictionary with word and it's postion of the support count rank
        self.wordorderdic = {}
        #
        #        self.preprocess(data)
        #        #first scan to build all the necessay dictionary
```

```
        self.construct(data)
        # second scan and build fp tree line  by line

    def construct(self, data):
        # get support count for all word
        for tran in data:
            for words in tran:
                if words in self.worddic.keys():
                    self.worddic[words] += 1
                else:
                    self.worddic[words] = 1
        wordlist = list(self.worddic.keys())
        # prune all the world with < min support count
        for word in wordlist:
            if (self.worddic[word] < self.minsup):
                del self.worddic[word]
        # sort the remaing items des, with first word count than work#id
        self.wordsortdic = sorted(self.worddic.items(), key=lambda x: (-
x[1], x[0]))
        # create a table containing word, wordcount and all link node of
that word
        t = 0
        for i in self.wordsortdic:
            word = i[0]
            wordc = i[1]
            self.wordorderdic[word] = t
            t += 1
            wordinfo = {'wordn': word, 'wordcc': wordc, 'linknode': None}
            self.nodetable.append(wordinfo)
        # construct fptree line by line

        for line in data:
            supword = []
            for word in line:
                # only keep words with support count higher than minsupport
                if word in self.worddic.keys():
                    supword.append(word)
            # insert words to the fp tree
            if len(supword) > 0:
                # reorder the words
                sortsupword = sorted(supword, key=lambda k:
self.wordorderdic[k])
                self.wordlinesort.append(sortsupword)
                # enter the word one by one from begining
                R = self.root
                #                 print(sortsupword)
                for i in sortsupword:
                    if i in R.children.keys():
                        R.children[i].word_count += 1
                        R = R.children[i]
                    else:

                        R.children[i] = node(word=i, word_count=1,
parent=R, link=None)
                        R = R.children[i]
                        # link this node to nodetable
                        for wordinfo in self.nodetable:
                            if wordinfo["wordn"] == R.word:
                                # find the last node of the  node linklist
                                if wordinfo["linknode"] is None:
                                    wordinfo["linknode"] = R
```

```
                                else:
                                    iter_node = wordinfo["linknode"]
                                    while (iter_node.link is not None):
                                        iter_node = iter_node.link
                                    iter_node.link = R

    # create transactions for conditinal tree
    def condtreetran(self, N):
        if N.parent is None:
            return None

        condtreeline = []
        # starting from the leaf node reverse add word till hit root
        while N is not None:
            line = []
            PN = N.parent
            while PN.parent is not None:
                line.append(PN.word)
                PN = PN.parent
            # reverse order the transaction
            line = line[::-1]
            for i in range(N.word_count):
                condtreeline.append(line)
                # move on to next linknode
            N = N.link
        return condtreeline

    # Find frequent word list by creating conditional tree
    def findfqt(self, parentnode=None):
        if len(list(self.root.children.keys())) == 0:
            return None
        result = []
        sup = self.minsup
        # starting from the end of nodetable
        revtable = self.nodetable[::-1]
        for n in revtable:
            fqset = [set(), 0]
            if (parentnode == None):
                fqset[0] = {n['wordn'], }
            else:
                fqset[0] = {n['wordn']}.union(parentnode[0])
            fqset[1] = n['wordcc']
            result.append(fqset)
            condtran = self.condtreetran(n['linknode'])
            # recursively build the conditinal fp tree
            contree = fptree(condtran, sup)
            conwords = contree.findfqt(fqset)
            if conwords is not None:
                for words in conwords:
                    result.append(words)
        return result

    # check if tree hight is larger than 1
    def checkheight(self):
        if len(list(self.root.children.keys())) == 0:
            return False
        else:
            return True


min_sup = 2
```

```
import numpy as np
import pandas as pd

dataset = pd.read_csv('menu_orders.csv', sep='delimeter', header=None)
test_data = []
for sublist in dataset.values.tolist():
    clean_sublist = [item for item in sublist if item is not np.nan]
    test_data.append(clean_sublist)

fp_tree = fptree(test_data, min_sup)  # create FP tree on data

# print ("\n========== Printing Frequent Word Set on " + i +" ==========")
frequentwordset = fp_tree.findfqt()  # mining frequent patt
frequentwordset = sorted(frequentwordset, key=lambda k: -k[1])

# print frequent patt
for word in frequentwordset:
    count = (str(word[1]) + "\t")
    words = ''
    for val in word[0]:
        words += (str([val]) + " ")
    print(count + words)

for i in fp_tree.nodetable[::-1]:
    lines = fp_tree.condtreetran(i['linknode'])
    condtree = fptree(lines, min_sup)
    if (condtree.checkheight()):
        print('Condtional FPTree Root on ' + (vocabdic[i['wordn']]))
        print(condtree.root.visittree())
```

Modified Algorithm for Apriori

Optimization in Apriori algorithm

1.It is possible to delete infrequent itemsets directly in the transaction database to avoid multiple scans and reduce I/O overhead

2.For frequent K itemsets, if a single item i appears less than k times, the itemset containing i cannot appear in the frequent k+1 itemset.In the frequent K itemset, items containing one i should be deleted and linked together

```
import pandas as pd


def DataSet(dataSet):
    C = []
    for tid in dataSet:
        for item in tid:
            if not [item] in C:
                C.append([item])
    C.sort()
    return list(map(frozenset, C))


def DataScan(dataSet, CK, min_support, numItem, k=0):  # Accept candidate k
```

```
itemsets and output frequent k itemsets
    ssCnt = {}
    for tid in dataSet:
        for can in CK:
            if can.issubset(tid):
                if can not in ssCnt:
                    ssCnt[can] = 1
                else:
                    ssCnt[can] += 1
    retList = []
    supportData = {}
    for key in ssCnt:
        support = float(ssCnt[key] / numItem)
        if support >= min_support:
            retList.insert(0, key)

        # Scan D again and delete the infrequent k itemset. This
improvement is to compress the transaction database and reduce the scanned
data
        else:

            for tid in dataSet:
                if key == tid:
                    dataSet.remove(tid)

        supportData[key] = support

    R_List = []
    # if a single item i appears less than k times, i cannot appear in the
frequent k+1 itemset. Items containing a single i should be deleted from
the frequent K itemset and then linked
    # Improvement direction: compression candidate set CK
    if k > 1:
        ssCnt = {}
        for tid in retList:
            for key in tid:
                if key not in ssCnt:
                    ssCnt[key] = 1
                else:
                    ssCnt[key] += 1
        tids = []
        for tid in retList:
            for item in tid:
                if item in ssCnt.keys():
                    if ssCnt[item] < k:
                        tids.append(tid)
        R_List = list(set(retList) - set(tids))

    print(
        'Frequent itemsets before optimization' + str(retList) + '
' + 'Optimized frequent itemsets' + str(
            R_List))
    return retList, supportData, R_List


def aprioriGen(LK, k, RK):  # Create candidate item set CK, where k is the
number of elements in the output set

    if RK:
        LK = RK
    else:
```

```
        pass

    retList = []
    lenLK = len(LK)
    for i in range(lenLK):
        for j in range(i + 1, lenLK):
            L1 = list(LK[i])[:k - 2]
            L2 = list(LK[j])[:k - 2]
            L1.sort()
            L2.sort()
            if L1 == L2:
                retList.append(LK[i] | LK[j])
    return retList


def apriori(dataSet, min_support_1):
    C1 = DataSet(dataSet)
    D = set()
    for tid in dataSet:
        tid = frozenset(tid)
        D.add(tid)
    numItem = float(
        len(D))  # This is the only numItem to be calculated. Otherwise,
the element of data list D will be deleted by scanD method, resulting in
the change of numItem
    L1, supportData, R1 = DataScan(D, C1, min_support_1, numItem)
    L = [L1]
    R = [R1]
    k = 2
    while len(L[k - 2]) > 0:
        CK = aprioriGen(L[k - 2], k, R[k - 2])
        LK, supK, RK = DataScan(D, CK, min_support_1, numItem, k)
        supportData.update(supK)
        L.append(LK)
        R.append(RK)
        k += 1
    L = [i for i in L if i]  # Delete empty list
    return L, supportData


name = 'menu_orders.txt'
minSupport = 0.2
minconfidence = 0.5


def createData(name):  # Preprocess the data and output the dataset

    D = pd.read_csv(name, header=None, index_col=False, names=['1', '2',
'3'])

    D = D.fillna(0)
    D = D.values.tolist()
    for i in range(len(D)):
        D[i] = [j for j in D[i] if j != 0]
    return D


def calculate(dataset):  # Calculation of support and confidence by
algorithm
    dataset, dic = apriori(dataset, minSupport)
```

```
    Rname = []
    Rsupport = []
    Rconfidence = []
    emptylist = []
    for i in range(len(dataset)):
        for AB in dataset[i]:
            for A in emptylist:
                if A.issubset(AB):
                    conf = dic.get(AB) / dic.get(AB - A)
                    if conf >= minconfidence:
                        Rname.append(str(AB - A) + '-->' + str(A))
                        Rconfidence.append(conf)
                        Rsupport.append(dic.get(AB))

            emptylist.append(AB)
    return Rname, Rsupport, Rconfidence


def outputdata(Rname, Rsupport, Rconfidence):
    data = {

        "Association rules": Rname,
        "Support": Rsupport,
        "Confidence": Rconfidence
    }
    df = pd.DataFrame(data,
                      columns=['Association rules', 'Support',
'Confidence'])

    return df


dataset = createData(name)
R1, R2, R3 = calculate(dataset)
df = outputdata(R1, R2, R3)
df.to_csv('Report.txt')
```