Правительство Российской Федерации

Федеральное государственное автономное образовательное учреждение
высшего профессионального образования

**«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»**

Факультет компьютерных наук

Большое домашнее задание по курсу

«Упорядоченные множества в анализе данных»

Выполнил студент группы мНоД16_ИССА
Смирнов Денис

Москва 2016

Исходный код этого iPython-блокнота, а также файл с данными и отчёт доступны на GitHub по ссылке https://github.com/dextravaganz/hse-fca-benchmark (https://github.com/dextravaganz/hse-fca-benchmark)

Описание данных

В данной работе использовался Mushroom Data Set из UCI Machine Learning Repository. Исходные данные доступны по адресу https://archive.ics.uci.edu/ml/datasets/Mushroom (https://archive.ics.uci.edu/ml/datasets/Mushroom)

Эти данные представляют с собой описание 8124 грибов, для каждого из которых указаны 22 категориальных атрибута, описывающих физические характеристики гриба и метку, показывающую, съедобен данный гриб или нет.

Для выполнения работы из исходного массива данных были взяты все наблюдения с меткой съедобности и следующими атрибутами:

1. cap-shape
2. cap-surface
3. cap-color
4. bruises
5. odor
6. gill-attachment
7. gill-spacing
8. gill-size

Рассматриваемые в работе алгоритмы будут сравниваться по результатам решения задачи бинарной классификации на этих данных - предсказании значения метки съедобности гриба.

Поскольку все признаки категориальные и количество категорий небольшое, то сразу же преобразуем их в бинарные с помощью дамми-переменных.

In [1]:
```
%matplotlib inline
import pandas as pd
import numpy as np

src_file = './agaricus-lepiota.data'

columns = [
    'edible',
    'cap-shape',
    'cap-surface',
    'cap-color',
    'bruises',
    'odor',
    'gill-attachment',
    'gill-spacing',
    'gill-size',
]

raw_data = pd.read_csv(src_file,
                       header=None,
                       usecols=[i for i in range(len(columns))],
                       names=columns)

data = pd.get_dummies(raw_data, prefix=columns)\
    .astype(int)\
    .rename(columns={'edible_e' : 'is_edible'})\
    .drop('edible_p', axis=1)

target = 'is_edible'

data[target].value_counts()
```

Out[1]:
```
1    4208
0    3916
Name: is_edible, dtype: int64
```

Как видно, соотношение положительных и отрицательных примеров примерно одинаковое.

Метод оценки алгоритмов

В следующей ячейке реализован метод, с помощью которого будет происходить оценка всех алгоритмов в данной работе.

В нём алгоритм тестируется на кросс-валидации, посчитанные средние метрики возвращаются в виде DataFrame с одной строкой.

In [2]:
```
from sklearn.model_selection import KFold
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score

def evaluate_method(trial_name, classifier, data, target,
                    np_convert=False, kf_splits=5):
```

```
    """Common evaluation method for algoritms in this research

    Parameters
    ----------
    trial_name : str
        Label of current experiment
     classifier
        Model to be evaluated.
        This object must implement 'fit' method
    data : DataFrame
    target : str
        Target feature name
    np_convert : bool, default False
        Indicates if data should be converted to NumPy
        arrays before passing to 'fit' method
    kf_splits : int, default 5
        KFold number of splits
    Returns
    -------
    DataFrame
        with one row labeled as 'trial_name'
        and containig scores in cells
    """
    X = data.drop(target, 1).fillna(0)
    Y = data[target]
    if np_convert:
        X = np.array(X)
        Y = np.array(Y)

    acc, pr, rc, tpr, tnr, fpr, npv, fdr = \
        [np.array([]) for i in xrange(8)]
    kf = KFold(n_splits=kf_splits)
    for train_ind, test_ind in kf.split(X):

        X_tr = X[train_ind] if np_convert else X.loc[train_ind]
        Y_tr = Y[train_ind] if np_convert else Y.loc[train_ind]

        X_test = X[test_ind] if np_convert else X.loc[test_ind]
        y_test = Y[test_ind] if np_convert else\
                                    np.array(Y.loc[test_ind])

        current_fold_model = \
            classifier.fit(X_tr, Y_tr)
        y_pred = current_fold_model.predict(X_test)

        acc = np.append(acc,accuracy_score(y_test,y_pred))
        pr = np.append(pr,precision_score(y_test,y_pred))
        rc = np.append(rc,recall_score(y_test,y_pred))
        TP = np.sum(y_test * y_pred)
        TN = np.sum(y_test + y_pred == 0)
        FP = np.sum((y_test  == 0) * (y_pred == 1))
        FN = np.sum((y_test  == 1) * (y_pred == 0))
        tpr = np.append(tpr,float(TP) / np.sum(y_test == 1))
        tnr = np.append(tnr,float(TN) / np.sum(y_test == 0))
        fpr = np.append(fpr,float(FP) / ((TP + FN)\
                                    if (TP + FN) !=0 else 1.))
```

```
        npv = np.append(npv,float(TN) / ((TN + FN)\
                                         if (TN + FN) !=0 else 1.))
        fdr = np.append(fdr,float(FP) / ((TP + FP)\
                                         if (TP + FP)!=0 else 1.))

    return pd.DataFrame(
        np.round(np.array([
             [acc.mean(), pr.mean(), rc.mean(), tpr.mean(),
              tnr.mean(), fpr.mean(), npv.mean(), fdr.mean()]]),
          decimals=4
        ),
        index = [trial_name],
        columns = ['Accuracy', 'Precision', 'Recall', 'TP', 'TN',
                   'FP', 'NP', 'FD'])
```

1. Базовое решение

В качестве базовой модели, будем использовать GradientBoostingClassifier из scikit-learn с параметрами по умолчанию

In [3]:
```
from sklearn.ensemble import GradientBoostingClassifier

metrics = evaluate_method('Baseline',
                          GradientBoostingClassifier(),
                          data, target,
                          np_convert=True)
metrics
```

Out[3]:

	Accuracy	Precision	Recall	TP	TN	FP	NP	FD
Baseline	0.9892	0.9593	0.9868	0.9868	0.9897	0.0472	0.9968	0.0407

По всем метрикам, результат градиентного бустинга почти идеален, однако недостатком этого алгоритма можно считать плохую интерпретируемость.

Посмотрим, как с этой же задачей справятся методы упорядоченных множеств и анализа формальных понятий, которые этого недостатка лишены.

2. ДСМ-Метод

Далее представлена моя реализация алгортима поиска формальных понятий Close by One с возможностью ограничить поиск формальных понятий по размеру содержания или по количеству найденных формальных понятий.

Этот алгоритм будет использоваться для порождения формальных понятий в контекстах положительных и отрицательных примеров. Представленный далее ДСМ-классификатор будет использовать полученные формальные понятия как гипотезы для классификации.

In [4]:
```
class FormalContext:
    """:param ctx - Pandas DataFrame representing formal context"""
    def __init__(self, ctx):
        self.ctx = ctx
        self.M = list(ctx.columns)
        self.G = list(ctx.index)

    def get_g(self, attributes):
        ind = self.ctx[attributes].all(1)
        return list(self.ctx[ind].index)

    def get_m(self, objects):
        ind = self.ctx.loc[objects].all()
        return list(self.ctx.columns[ind])

def close_by_one(ctx, max_fc=-1, max_intent=-1):
    M = sorted(ctx.M)
    if max_intent < 0:
        max_intent = len(M)
    if max_fc < 0:
        max_fc = 2**min(len(M), len(ctx.G))
    formal_concepts = []
    if ctx.get_m(ctx.get_g([])) == [] and max_fc > 0:
        formal_concepts.append(([], ctx.get_g([])))
    def close_by_one_inner(candidate_prefix):
        last_ind = M.index(candidate_prefix[-1]) \
            if len(candidate_prefix) > 0 else 1
        for i in xrange(last_ind + 1, len(M)):
            next = candidate_prefix + [M[i]]
            while not set(ctx.get_m(ctx.get_g(next)))==\
                      set(next) and\
                  i < len(M) - 1:
                i += 1
                next.append(M[i])
            if set(ctx.get_m(ctx.get_g(next))) == set(next) and\
                len(next) <= max_intent and\
                len(formal_concepts) < max_fc:
                formal_concepts.append((next, ctx.get_g(next)))
                close_by_one_inner(next)
            else:
                pass
    close_by_one_inner([])

    return formal_concepts
```

Ниже представлена моя совбственная реализация ДСМ классификатора. Он предоставляет возможность задать результат классификации по умолчанию, который будет использоваться в спорных ситуациях, использовать ли нормировку голосов на количество гипотез, а также максимальное допустимое количество контр-примеров, при которых гипотеза классификации принимается.

In [5]:
```
class JSMClassifier:

    def __init__(self,
```

```
                 unclassified_label=0,
                 vote_adjustment='None',
                 max_positive_counter_exmpls=0,
                 max_negative_counter_exmpls=0,
                 max_positive_fc=-1,
                 max_negative_fc=-1,
                 max_positive_intent=-1,
                 max_negative_intent=-1,
                 debug_output=False):
        self.unclassified_label = unclassified_label
        self.max_positive_counter_exmpls = max_positive_counter_exm
pls
        self.max_negative_counter_exmpls = max_negative_counter_exm
pls
        if not vote_adjustment in ('None', 'HypothesisCount'):
            raise ValueError('Unsupperted vote_adjustment')
        self.vote_adjustment = vote_adjustment
        self._max_positive_fc=max_positive_fc
        self._max_negative_fc=max_negative_fc
        self._max_positive_intent=max_positive_intent
        self._max_negative_intent=max_negative_intent
        self._debug_output = debug_output


    def fit(self, X, Y):
        if self._debug_output:
            print('-- fit(X,Y) start --')

        self._positive_ctx = FormalContext(X[Y == 1.])
        self._negative_ctx = FormalContext(X[Y == 0.])

        positive_fc = \
            close_by_one(self._positive_ctx,
                         max_fc=self._max_positive_fc,
                         max_intent=self._max_positive_intent)
        if self._debug_output:
            print('Positive concepts count:' +\
                  str(len(positive_fc)))
        self._positive_hypothesis = []
        for (attributes, _) in positive_fc:
            if len(self._negative_ctx.get_g(attributes)) <=\
                   self.max_negative_counter_exmpls:
                self._positive_hypothesis.append(attributes)

        negative_fc = \
            close_by_one(self._negative_ctx,
                         max_fc=self._max_negative_fc,
                         max_intent=self._max_negative_intent)
        if self._debug_output:
            print('Negative concepts count:' +\
                  str(len(negative_fc)))
        self._negative_hypothesis = []
        for (attributes, _) in negative_fc:
            if len(self._positive_ctx.get_g(attributes)) <= \
                   self.max_positive_counter_exmpls:
                self._negative_hypothesis.append(attributes)
```

```
        if self._debug_output:
            print('Total positive hypothesis:' + \
                  str(len(self._positive_hypothesis)))
            print('Total neagative hypothesis:' + \
                  str(len(self._negative_hypothesis)))
        if self._debug_output:
            print('-- fit(X,Y) end --')
        return self


    def predict(self, g):
        positive_votes = \
            pd.DataFrame(np.zeros(len(g.index)),
                         index=g.index,columns=['sum'])
        for hypo in self._positive_hypothesis:
            positive_votes['sum'] += \
                (g[hypo].sum(1)==len(hypo)).astype(int)

        positive_adj = len(self._positive_hypothesis) \
            if self.vote_adjustment=='HypothesisCount' and\
                len(self._positive_hypothesis) > 0 else 1.
        positive_votes['sum'] = \
            positive_votes['sum'].astype(float) / positive_adj


        negative_votes = \
            pd.DataFrame(np.zeros(len(g.index)),
                         index=g.index, columns=['sum'])
        for hypo in self._negative_hypothesis:
            negative_votes['sum'] += \
                (g[hypo].sum(1)==len(hypo)).astype(int)

        negative_adj = len(self._negative_hypothesis) \
            if self.vote_adjustment=='HypothesisCount' and\
                len(self._negative_hypothesis) > 0 else 1.
        negative_votes['sum'] = \
            negative_votes['sum'].astype(float) / negative_adj

        if self.unclassified_label == 1:
            pred = positive_votes['sum'] >= negative_votes['sum']
        else:
            pred = positive_votes['sum'] > negative_votes['sum']

        return pred.values.T.astype(int)
```

Проверим, как ДСМ классификатор решает нашу задачу с параметрами по умаолчанию. Параметры по умолчанию предполагают, что в качестве гипотез будт проверяться все существующие в контекстах формальные понятия, результат голосования никак не будет нормироваться, а все принимаемые гипотезы должны быть строгими, то есть не допускать контрпримеров.

In [6]:
```
metrics = metrics.append(
    evaluate_method('JSM (strict)',
                    JSMClassifier(debug_output=True),
                    data, target,
                    np_convert=False))
metrics
```

```
-- fit(X,Y) start --
Positive concepts count:115
Negative concepts count:55
Total positive hypothesis:17
Total neagative hypothesis:3
-- fit(X,Y) end --
-- fit(X,Y) start --
Positive concepts count:130
Negative concepts count:55
Total positive hypothesis:21
Total neagative hypothesis:3
-- fit(X,Y) end --
-- fit(X,Y) start --
Positive concepts count:130
Negative concepts count:55
Total positive hypothesis:21
Total neagative hypothesis:3
-- fit(X,Y) end --
-- fit(X,Y) start --
Positive concepts count:129
Negative concepts count:40
Total positive hypothesis:33
Total neagative hypothesis:3
-- fit(X,Y) end --
-- fit(X,Y) start --
Positive concepts count:80
Negative concepts count:9
Total positive hypothesis:18
Total neagative hypothesis:1
-- fit(X,Y) end --
```

Out[6]:

	Accuracy	Precision	Recall	TP	TN	FP	NP	FD
Baseline	0.9892	0.9593	0.9868	0.9868	0.9897	0.0472	0.9968	0.0407
JSM (strict)	0.7596	0.9764	0.5176	0.5176	0.9961	0.0134	0.6111	0.0236

Результаты неплохие. ДСМ метод с параметрами по умолчанию справляется с задачей явно лучше, чем случайное угадывание, однако до качества градиентного бустинга пока не дотягивает.

Можно заметить, что при высоком True Negative rate, очень низкий показатель True Positive, то есть данный классификатор часто определяет положительные как отрицательные.

В контексте выбора грибов это, может быть, и хорошо: лучше не съесть ядовитый гриб, чем выкинуть съедобный, однако от этого смещения страдает общая точность классификации,

Попробуем улучшить показатели. Судя по выводу отладки, в большинстве случаев, отрицательных гипотез больше, чем положительных, от этого и смещение. Попробуем внести следующие изменения в настройки:

1. Включим нормирование на количество гипотез, чтобы решение принималось процентным большиством, а не абсолютным
2. Попробуем получить большее колчиство положительных гипотез, разрешив принимать положительную гипотезу с небольшим количеством контрпримеров, попробуем со значениями 3, 5 и 10

In [7]:
```
metrics = metrics.append(
    evaluate_method('JSM (strict, adj)',
                    JSMClassifier(
                        vote_adjustment='HypothesisCount'),
                    data, target,
                    np_convert=False))
metrics = metrics.append(
    evaluate_method('JSM (soft+, 3)',
                    JSMClassifier(
                        max_negative_counter_exmpls=3),
                    data, target,
                    np_convert=False))
metrics = metrics.append(
    evaluate_method('JSM (soft+, 5)',
                    JSMClassifier(
                        max_negative_counter_exmpls=5),
                    data, target,
                    np_convert=False))
metrics = metrics.append(
    evaluate_method('JSM (soft+, 10)',
                    JSMClassifier(
                        max_negative_counter_exmpls=10),
                    data, target,
                    np_convert=False))

metrics
```

Out[7]:

	Accuracy	Precision	Recall	TP	TN	FP	NP	FD
Baseline	0.9892	0.9593	0.9868	0.9868	0.9897	0.0472	0.9968	0.0407
JSM (strict)	0.7596	0.9764	0.5176	0.5176	0.9961	0.0134	0.6111	0.0236
JSM (strict, adj)	0.7596	0.9764	0.5176	0.5176	0.9961	0.0134	0.6111	0.0236
JSM (soft+, 3)	0.7752	0.9503	0.5861	0.5861	0.9894	0.0460	0.6242	0.0497
JSM (soft+, 5)	0.8010	0.9424	0.6531	0.6531	0.9845	0.0700	0.6458	0.0576
JSM (soft+, 10)	0.8027	0.9395	0.6571	0.6571	0.9835	0.0750	0.6473	0.0605

Нормирование по колчиству гипотез не изменило ситуацию, а вот смягчение положительных гипотез дало прирост точности.

Однако, как можно заметить, наибольший прирост дают гипотезы, у которых не больше 5 контрпримеров в отрицательном контексте, а гипотезы, у которых от 5 до 10 контрпримеров, оказывают несущественное влияние.

3. Ленивый алгоритм

Анализ ошибок классификации и настройка ДСМ метода позволяет приблизиться к качеству градиентного бустинга, однако у такого метода есть существенный недостаток - при обучении, для построения гипотез нужно найти формальные понятия, что в худшем случае делается за экспоненциальное время.

Поэтому далее мы рассмотрим алгоритм ленивой классификации, в котором не требуется находить формальные понятия. (Из корня репозитория lazyfca15 с незначительными доработками).

In [8]:
```
class LazyClassifier:

    def __init__(self,
                 unclassified_label=0,
                 debug_output=False):
        self.unclassified_label = unclassified_label
        self._debug_output = debug_output

    def fit(self, X, Y):
        self._positive = X[Y == 1]
        self._negative = X[Y == 0]
        return self

    def predict(self, X_test):
        y_pred = []
        unclassified = 0
        for test_obj in X_test:
            pos = np.sum(test_obj == self._positive) / float(len(se
lf._positive))
            neg = np.sum(test_obj == self._negative) / float(len(se
lf._negative))
            if pos > neg:
                y_pred.append(1)
            elif pos < neg:
                y_pred.append(0)
            else:
                unclassified+=1
                y_pred.append(self.unclassified_label)
        if self._debug_output:
            print(str(unclassified) + ' objects were unclassified')
        return np.array(y_pred)
```

In [9]:
```
metrics = metrics.append(
    evaluate_method('Lazy',
                    LazyClassifier(),
                    data, target,
                    np_convert=True))
metrics
```

Out[9]:

	Accuracy	Precision	Recall	TP	TN	FP	NP	FD
Baseline	0.9892	0.9593	0.9868	0.9868	0.9897	0.0472	0.9968	0.0407
JSM (strict)	0.7596	0.9764	0.5176	0.5176	0.9961	0.0134	0.6111	0.0236
JSM (strict, adj)	0.7596	0.9764	0.5176	0.5176	0.9961	0.0134	0.6111	0.0236
JSM (soft+, 3)	0.7752	0.9503	0.5861	0.5861	0.9894	0.0460	0.6242	0.0497
JSM (soft+, 5)	0.8010	0.9424	0.6531	0.6531	0.9845	0.0700	0.6458	0.0576
JSM (soft+, 10)	0.8027	0.9395	0.6571	0.6571	0.9835	0.0750	0.6473	0.0605
Lazy	0.9189	0.8377	0.9290	0.9290	0.9268	0.2855	0.9074	0.1623

Простой метод ленивой классификации смог значительно превзойти ДСМ без всякой настройки.

В заключение можно сказать, что построение предсказетельных моделей - долгий и кропотливый процесс, требующий много итераций. Всегда лучше начинать с простого алгоритма, с интерпретируемыми результатами. Высока вероятность, что при минимальной настройке он сможет выполнять поставленую задачу с адекватным качеством.