Illustrate the different kinds of classification including "multi-class-multi-label" using a simple problem.

The classification is less the point here than understanding the different ways that input data can be structured and the capabilities and output formats of various classifiers. Also, in practice, multiclass and multilabel are often used interchangeably while they have specific, different meanings in sklearn. Multiclass-multilabel is not defined by sklearn. It is generally used to mean what sklearn calls multioutput-multilabel.

- The scenario:

We have an store that sells multiple items. Our customers are either male or female and grouped into age groups, 'adult', 'senior' and 'youth'.

For a subset of customers, we have this information and we would like to predict the sex and age group of new customers by examining their purchases.

So, there are 2 targets (aka 'classes') to be predicted. Gender has 2 possible labels, 'female' and 'male'. Age group has 3 possible labels, 'adult', 'senior' and 'youth'.

There are several ways to structure this problem.

The terminology:

The following definitions are from http://scikit-learn.org/stable/modules/multiclass.html.

> Multiclass classification means a classification task with more than two classes; e.g., classify a set of images of fruits which may be oranges, apples, or pears. Multiclass classification makes the assumption that each sample is assigned to one and only one label: a fruit can be either an apple or a pear but not both at the same time.
>
> Multilabel classification assigns to each sample a set of target labels. This can be thought as predicting properties of a data-point that are not mutually exclusive.

In our toy problem, gender and age_group are the two items to be predicted. This most closely aligns with the second definition below, i.e. multioutput-multiclass.

> Multioutput regression assigns each sample a set of target values. This can be thought of as predicting several properties for each data-point, such as wind direction and magnitude at a certain location.
>
> Multioutput-multiclass classification and multi-task classification means that a single estimator has to handle several joint classification tasks. This is both a generalization of the multi-label classification task, which only considers binary classification, as well as a generalization of the multi-class classification task. The output format is a 2d numpy array or sparse matrix.

Unfortunately, in practice, the terms multi-label and multi-class are often used interchangeably. In the following we have chosen to use multi-target, multi-class to mean predicting several outputs that are independent of each other, where each prediction is one of a set of mutually exclusive and exhaustive labels specific to that target (the output must be one and only one of the labels for that target). In sklearn parlance, this should probably be 'multioutput-multiclass'.

Note: in the following, for the sake of brevity, we will just fit and predict on the whole dataset because we are only interested in the input and output structure. In the normal case the dataset would be split into train and test sets.

In [1]:
```
import pandas as pd
import numpy as np
import random
from sklearn.linear_model import LogisticRegression
from sklearn.multiclass import OneVsRestClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, f1_score, log_loss

from sklearn.preprocessing import LabelEncoder
```

In [2]:
```
# a function to build a toy dataset
def make_data(size):
    # build target gender
    genders = [random.choice(['male','female' ]) for _ in range(size)]
    # build target ages
    ages = [random.choice(['youth','adult', 'senior' ]) for _ in range(siz
e)]
    # build purchases columns
    y1 = [random.choice([1, 0, 0, 0 ]) for _ in range(size)]
    y2 = [random.choice([1, 1, 1, 0, 0, 0 ]) for _ in range(size)]
    a1 = [random.choice([1, 1, 0, 0, 0]) for _ in range(size)];
    a2 = [random.choice([1, 1, 1, 0, 0, 0, 0]) for _ in range(size)]
    s1 = [random.choice([1, 1, 0, 0 ]) for _ in range(size)];
    s2 = [random.choice([1, 1, 0, 0, 0, 0 ]) for _ in range(size)]
    # make a df
    the_data = pd.DataFrame([genders, ages, s1, s2, a1, a2, y1, y2]).T
    # name the cols
    the_data.columns = ['gender', 'age', 'p1', 'p2', 'p3', 'p4', 'p5', 'p6
']
    return the_data
```

In [3]:
```
the_data = make_data(12); the_data
```

Out[3]:

	gender	age	p1	p2	p3	p4	p5	p6
0	female	adult	1	0	1	0	0	0
1	male	adult	1	0	0	0	0	0
2	female	senior	1	0	0	0	0	0
3	female	youth	1	0	0	1	1	1
4	male	adult	0	0	0	1	0	1

5	female	adult	0	0	1	1	0	1
6	male	adult	1	0	0	1	1	0
7	female	adult	0	0	1	1	0	1
8	male	adult	1	0	1	1	0	1
9	male	youth	1	0	1	0	0	1
10	male	youth	1	1	0	0	0	0
11	female	senior	1	0	0	0	0	1

Binary classification

For the first classification, we predict gender only. Since there are two possible outcomes of a prediction this is binary classification. The string labels 'female' and 'male' could be translated to 0s and 1s before classification but almost all (if not all) classifiers will do this automatically. If string labels are provided in the input, they are used for predictions. If integer mappings are used, these integers will used for the predictions.

In [4]:
```
# instantiate a classifier
lr_clf = LogisticRegression()
```

In [5]:
```
# define the inputs
X = the_data[['p1', 'p2', 'p3', 'p4','p5', 'p6']]; y = the_data['gender']
```

In [6]:
```
# fit the classifier on training set
lr_clf.fit(X, y)
```

Out[6]:
```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=Tru
e,
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
          verbose=0, warm_start=False)
```

In [7]:
```
# generate predictions for the training set
gender_yhat = lr_clf.predict(X)
```

In [8]:
```
# examine
gender_yhat
```

Out[8]:
```
array(['female', 'male', 'male', 'male', 'female', 'female', 'male',
       'female', 'female', 'female', 'male', 'female'], dtype=object)
```

Some metrics (accuracy_score, f1_score, etc.) use predicted values and y .

In [9]:
```
# the accuracy
accuracy_score(y, gender_yhat)
```

Out[9]:
```
0.5833333333333334
```

In [10]:
```
# the f1_score
f1_score(y, gender_yhat, labels=['female', 'male'], pos_label='male')
```

Out[10]:
```
0.5454545454545454
```

Some metrics (log_loss, ROC_AUC, etc.) use predicted probabilties and y .

In [11]:
```
# get the probability of each target given the predictors
gender_probas = lr_clf.predict_proba(X)
```

The probability output for a target is an array of probilities for each possible output value (female and male, in this case). Note that for a target, the sum of the probabilities is 1.0, because the true value must be exactly one of the possible labels.

In [12]:
```
gender_probas
```

Out[12]:
```
array([[0.50442471, 0.49557529],
       [0.44558056, 0.55441944],
       [0.44558056, 0.55441944],
       [0.48599851, 0.51400149],
       [0.52934213, 0.47065787],
       [0.58752575, 0.41247425],
       [0.43045563, 0.56954437],
       [0.58752575, 0.41247425],
       [0.52414981, 0.47585019],
       [0.56012544, 0.43987456],
       [0.35940499, 0.64059501],
       [0.50135709, 0.49864291]])
```

Multi-class classification (sklearn terminology, in practice often called multi-label)

Now consider, the second target, age group. This target has 3 possible values, adult, senior and youth. The predictors (features) are the same as previously. We could transform the string labels to a set of integer values, but again, the classifier will do that automatically. All sklearn classifiers are able to do multiclass problems.

Again, labels may be mapped to integer values or used directly.

In [13]:
```
# rebind y; X stays the same
y = the_data['age']
```

In [14]:
```
# make a new classifier
clf_age_group = LogisticRegression()
```

In [15]:
```
# fit
clf_age_group.fit(X, y)
```

Out[15]:
```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=Tru
e,
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
```

```
                   verbose=0, warm_start=False)
```

In [16]:
```
# predict
yhat = clf_age_group.predict(X)
```

In [17]:
```
# examine predictions
yhat
```

Out[17]:
```
array(['adult', 'adult', 'adult', 'adult', 'adult', 'adult', 'adult',
       'adult', 'adult', 'adult', 'youth', 'youth'], dtype=object)
```

Notice that the classifier has predicted a 1-d structure, the same format as the input.

In [18]:
```
# accuracy
accuracy_score(y,yhat)
```

Out[18]:
```
0.6666666666666666
```

Now we predict probabilities

In [19]:
```
yhat_probas =  clf_age_group.predict_proba(X)
```

In [20]:
```
# examine the probabilities
yhat_probas
```

Out[20]:
```
array([[0.5375623 , 0.19424327, 0.26819444],
       [0.38339763, 0.31185813, 0.30474424],
       [0.38339763, 0.31185813, 0.30474424],
       [0.5301731 , 0.13813737, 0.33168953],
       [0.6044801 , 0.17052148, 0.22499842],
       [0.71661278, 0.09425259, 0.18913463],
       [0.56991352, 0.14421579, 0.28587069],
       [0.71661278, 0.09425259, 0.18913463],
       [0.6956917 , 0.09425035, 0.21005796],
       [0.49784294, 0.18861058, 0.31354648],
       [0.31515821, 0.26059266, 0.42424913],
       [0.33659556, 0.30890597, 0.35449847]])
```

In [21]:
```
#  log loss
log_loss(y, yhat_probas,  labels=['adult', 'senior', 'youth'])
```

Out[21]:
```
0.7612551368765109
```

In [22]:
```
# check row sums
yhat_probas.sum(axis=1)
```

Out[22]:
```
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

Notice that we now have 3 columns of probability output, one for each possible label. Again, each row's probabilities sum to 1 for the same reason as in binary classification, the true value must be exactly one of the possible labels.

Multi-label (sklearn terminology) classification

In this kind of classification, we have multiple targets. One possible reason that multi-class and multi-label are often conflated is that multi-class classification can readily be transformed into multi-label by encoding the single target (with n labels) as n columns of binary, as in the following example.

This kind of classification is best suited to problems in which a set of target labels may or may not be assigned to a sample. For example, a movie might be classified as 'action', 'romance', 'drama', 'comedy'. The appropriate subset of possible labels should be assigned to an unlabeled sample.

In [23]:
```
# here are the counts of the labels for 'age' in the data.
the_data['age'].value_counts()
```

Out[23]:
```
adult     7
youth     3
senior    2
Name: age, dtype: int64
```

We can use the pandas function get_dummies() to encode a single target column with multiple classes (labels) as n columns of binary data. A one in the column means that label is present in original input; a zero indicates its absence.

In [24]:
```
the_data.head()
```

Out[24]:

	gender	age	p1	p2	p3	p4	p5	p6
0	female	adult	1	0	1	0	0	0
1	male	adult	1	0	0	0	0	0
2	female	senior	1	0	0	0	0	0
3	female	youth	1	0	0	1	1	1
4	male	adult	0	0	0	1	0	1

In [25]:
```
dummies = pd.get_dummies(the_data[['age']]); dummies.head()
```

Out[25]:

	age_adult	age_senior	age_youth
0	1	0	0
1	1	0	0
2	0	1	0
3	0	0	1
4	1	0	0

In [26]:
```
# look at the number of 1s in each dummy column; note correspondence with
previous format
dummies.sum()
```

Out[26]:
```
age_adult     7
```

```
age_senior    2
age_youth     3
dtype: int64
```

We see above that the single target column has been encoded as explained above. The multi-class problem has been transformed into a multi-label problem, requiring n binary predictions.

We cannot use logistic regression to directly classify with this input (other classifiers will accept input in this form), but the classification can be accomplished by wrapping logistic regression in sklearn.multiclass.OneVsRest().

OneVsRest creates one binary classifier per ouput. For each output, if the input column is positive (i.e. 1) that is a positive sample, if the input column is negative (i.e. 0), that row is a negative sample.

We note that by the sklearn definition, all of the binary classifications are independent, unlike the multi-class example above where exactly one of the classes is true. In this case, the predictions proceed as if the other columns did not exist, thus any number of columns can be true or false.

In order to produce a classification equivalent to the one above, the probabilities must be predicted, then normalized such that their sum is one, then the most probable column is selected as 1 and the others are set to 0.

In [27]:
```
# make a new classifier
multilabel_clf = OneVsRestClassifier(LogisticRegression())
```

In [28]:
```
# fit the classifier
multilabel_clf.fit(X, dummies)
```

Out[28]:
```
OneVsRestClassifier(estimator=LogisticRegression(C=1.0, class_weight=None,
 dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
          verbose=0, warm_start=False),
          n_jobs=1)
```

In [29]:
```
# look at predictions: some rows have no ones;
yhat = multilabel_clf.predict(X); yhat
```

Out[29]:
```
array([[1, 0, 0],
       [0, 0, 0],
       [0, 0, 0],
       [1, 0, 0],
       [1, 0, 0],
       [1, 0, 0],
       [1, 0, 0],
       [1, 0, 0],
       [1, 0, 0],
       [0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
```

You can see that the predictions don't match the multi-class example above. This is because

each column is independent of the others (i.e. sklearn's 'multi-label'). Accuracy doesn't care that these predictions don't make sense in this problem; it just gives the ratio of matches to total input size.

In [30]:
```
# train accuracy - not necessarily to be believed...
accuracy_score(dummies, yhat)
```

Out[30]:
```
0.5
```

In [31]:
```
# predict probabilities and examine
yhat_probas = multilabel_clf.predict_proba(X); yhat_probas
```

Out[31]:
```
array([[0.5860069 , 0.21174829, 0.29236386],
       [0.43318958, 0.35235922, 0.34432145],
       [0.43318958, 0.35235922, 0.34432145],
       [0.52805913, 0.13758658, 0.33036698],
       [0.6796947 , 0.19173922, 0.25299465],
       [0.79717017, 0.10484792, 0.21039603],
       [0.61462369, 0.15552963, 0.30829747],
       [0.79717017, 0.10484792, 0.21039603],
       [0.70501611, 0.0955136 , 0.21287338],
       [0.49826173, 0.18876924, 0.31381023],
       [0.35004123, 0.28943614, 0.47120679],
       [0.34903474, 0.32032186, 0.36759927]])
```

When we look at the probabilities we can see why not all rows get a 1 prediction in one of the 3 columns representing the different classes (age_adult, age_senior, age_youth). Each prediction is for a binary classification of one column. If the probability is greater than 0.5 (the threshold for binary logistic regression), the classifier predicts a 1, otherwise 0.

In [32]:
```
# check row sums
yhat_probas.sum(axis=1)
```

Out[32]:
```
array([1.09011906, 1.12987025, 1.12987025, 0.99601269, 1.12442856,
       1.11241411, 1.0784508 , 1.11241411, 1.01340309, 1.00084121,
       1.11068417, 1.03695588])
```

Note that the rows sums do not exactly equal 1. To make this approach equivalent to the multi-class scheme above requires 3 steps after fitting.

1. predict probabilities
2. normalize such that the sum of the probabilities is 1
3. to predict which column should be true, select the column with highest probability

Note that the normalized probabilites are precisely equal to the multiclass probabilities above.

In [102]:
```
normed_probas = yhat_probas/yhat_probas.sum(axis=1, keepdims=True); normed
_probas
```

Out[102]:
```
array([[0.5375623 , 0.19424327, 0.26819444],
       [0.38339763, 0.31185813, 0.30474424],
       [0.38339763, 0.31185813, 0.30474424],
       [0.5301731 , 0.13813737, 0.33168953],
```

```
       [0.6044801 , 0.17052148, 0.22499842],
       [0.71661278, 0.09425259, 0.18913463],
       [0.56991352, 0.14421579, 0.28587069],
       [0.71661278, 0.09425259, 0.18913463],
       [0.6956917 , 0.09425035, 0.21005796],
       [0.49784294, 0.18861058, 0.31354648],
       [0.31515821, 0.26059266, 0.42424913],
       [0.33659556, 0.30890597, 0.35449847]])
```

In [34]:
```
# verify row sums
normed_probas.sum(axis=1)
```

Out[34]:
```
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

Now the predictions can be corrected as well.

In [35]:
```
# build predictions array
predictions = np.zeros(normed_probas.shape)
# this can be done directly with numpy, but this code exposes the correct
logic.
selected = np.argmax(normed_probas, axis=1)
for i in range(predictions.shape[0]):
    predictions[i, selected[i]] = 1
```

In [36]:
```
# the predictions now match the multi-class example.
predictions
```

Out[36]:
```
array([[1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [0., 0., 1.],
       [0., 0., 1.]])
```

The correct prediction requires that the highest probability column be selected. The code above enforces the constraint that all the probabilities of the dummy variables must sum to one and the prediction must have one and only one of these columns (the most probable) set to 1 with all the other columns equal to 0.

This is because in transforming the problem to n binary classifications (one per label or class), we have lost this constraint which is enforced by the multi-class version above.

Multioutput-multiclass (often known as multiclass-multilabel).

This is the natural formulation for our toy problem: we want to predict two output, gender and age, each of which must be one of the appropriate labels. In sklearn, only DecisionTreeClassifier, ExtraTreeClassifier, ExtraTreesClassifier, KNeighborsClassifier, RadiusNeighborsClassifier and

RandomForestClassifier support this approach.

We'll do this example with RandomForestClassifier and then look at another approach.

RandomForestClassifier wants the different classes for each output expressed as numbers not strings. We can do that with pandas or LabelEncoder.

In [37]:
```
targets = the_data[['gender', 'age']]
```

In [38]:
```
the_data['coded_gender'] = the_data.gender.map({'female' : 0, 'male' : 1})
```

In [39]:
```
the_data['coded_age'] = the_data.age.map({'adult' : 0, 'senior' : 1, 'youth' : 2})
```

In [40]:
```
# we tack this into the data to make it easy to keep track
```

In [41]:
```
the_data
```

Out[41]:

	gender	age	p1	p2	p3	p4	p5	p6	coded_gender	coded_age
0	female	adult	1	0	1	0	0	0	0	0
1	male	adult	1	0	0	0	0	0	1	0
2	female	senior	1	0	0	0	0	0	0	1
3	female	youth	1	0	0	1	1	1	0	2
4	male	adult	0	0	0	1	0	1	1	0
5	female	adult	0	0	1	1	0	1	0	0
6	male	adult	1	0	0	1	1	0	1	0
7	female	adult	0	0	1	1	0	1	0	0
8	male	adult	1	0	1	1	0	1	1	0
9	male	youth	1	0	1	0	0	1	1	2
10	male	youth	1	1	0	0	0	0	1	2
11	female	senior	1	0	0	0	0	1	0	1

In [42]:
```
type(the_data.gender)
```

Out[42]:
```
pandas.core.series.Series
```

Now we can use RandomForest in a multioutput-multiclass fashion

In [43]:
```
rf = RandomForestClassifier()
```

In [44]:
```
rf.fit(the_data[['p1', 'p2', 'p3', 'p4', 'p5', 'p6']], the_data[['coded_ge
```

```
nder', 'coded_age']])
```

Out[44]:
```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini'
,
            max_depth=None, max_features='auto', max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators=10, n_jobs=1,
            oob_score=False, random_state=None, verbose=0,
            warm_start=False)
```

In [45]:
```
# keeps track of the classes for each output
rf.n_classes_
```

Out[45]:
```
[2, 3]
```

In [46]:
```
rf.classes_
```

Out[46]:
```
[array([0, 1], dtype=int64), array([0, 1, 2], dtype=int64)]
```

In [47]:
```
# now predict with the fitted model
rf_prediction = rf.predict(the_data[['p1', 'p2', 'p3', 'p4', 'p5', 'p6']])
; rf_prediction
```

Out[47]:
```
array([[0., 0.],
       [0., 1.],
       [0., 1.],
       [0., 2.],
       [1., 0.],
       [0., 0.],
       [1., 0.],
       [0., 0.],
       [1., 0.],
       [1., 2.],
       [1., 2.],
       [0., 1.]])
```

The labels can be recovered with the reverse operation.

In [48]:
```
rf_predict = pd.DataFrame(data=rf_prediction, columns = ['p_gen', 'p_age']
)
```

In [49]:
```
rf_predict['p_gen_s'] = rf_predict.p_gen.map({0:'female',1:'male'})
```

In [50]:
```
rf_predict['p_age_s'] = rf_predict.p_age.map({0:'adult',1:'senior',2:'yout
h'})
```

In [51]:
```
the_data = pd.concat([the_data, rf_predict], axis=1); the_data
```

Out[51]:

	gender	age	p1	p2	p3	p4	p5	p6	coded_gender	coded_age	p_gen	p_age	p_gen_s
0	female	adult	1	0	1	0	0	0	0	0	0.0	0.0	female
1	male	adult	1	0	0	0	0	0	1	0	0.0	1.0	female

2	female	senior	1	0	0	0	0	0	0	1	0.0	1.0	female
3	female	youth	1	0	0	1	1	1	0	2	0.0	2.0	female
4	male	adult	0	0	0	1	0	1	1	0	1.0	0.0	male
5	female	adult	0	0	1	1	0	1	0	0	0.0	0.0	female
6	male	adult	1	0	0	1	1	0	1	0	1.0	0.0	male
7	female	adult	0	0	1	1	0	1	0	0	0.0	0.0	female
8	male	adult	1	0	1	1	0	1	1	0	1.0	0.0	male
9	male	youth	1	0	1	0	0	1	1	2	1.0	2.0	male
10	male	youth	1	1	0	0	0	0	1	2	1.0	2.0	male
11	female	senior	1	0	0	0	0	1	0	1	0.0	1.0	female

The probability outputs

In [52]:
```
rf.predict_proba(the_data[['p1', 'p2', 'p3', 'p4', 'p5', 'p6']])
```

Out[52]:
```
[array([[0.78333333, 0.21666667],
        [0.57333333, 0.42666667],
        [0.57333333, 0.42666667],
        [0.7       , 0.3       ],
        [0.4       , 0.6       ],
        [0.9       , 0.1       ],
        [0.4       , 0.6       ],
        [0.9       , 0.1       ],
        [0.2       , 0.8       ],
        [0.2       , 0.8       ],
        [0.25      , 0.75      ],
        [0.6       , 0.4       ]]),
 array([[0.91666667, 0.08333333, 0.        ],
        [0.42666667, 0.57333333, 0.        ],
        [0.42666667, 0.57333333, 0.        ],
        [0.3       , 0.        , 0.7       ],
        [0.9       , 0.1       , 0.        ],
        [1.        , 0.        , 0.        ],
        [0.6       , 0.        , 0.4       ],
        [1.        , 0.        , 0.        ],
        [0.8       , 0.        , 0.2       ],
        [0.2       , 0.        , 0.8       ],
        [0.05      , 0.25      , 0.7       ],
        [0.        , 0.6       , 0.4       ]])]
```

Note that we get a list of 2 elements, one for each output. Each list element is an array of probabilities, n_samples by n_classes, properly normalized. This is the equivalent result to breaking the problem into two pieces, one for each output, and doing a multiclass classification separately for each and joining the result in a list. There is one list element per output.

Another method: OneVsRestClassifier(LogisticRegressionClassifier)

This is the method presented in the DrivenData tutorial. This is a generalization of the transformation from multiclass to multilabel for predicting multiple outputs. DrivenData omits the normalization step, but provides a metric that normalizes within targets for the competition.

Here we encode each output-class pair as a binary columns and use OneVsRestClassifier to produce the necessary binary classifiers.

First we clean up the data.

In [53]:
```
the_data__ = the_data; the_data = the_data__[['gender', 'age', 'p1', 'p2',
 'p3', 'p4', 'p5', 'p6']]
```

In [54]:
```
the_data
```

Out[54]:

	gender	age	p1	p2	p3	p4	p5	p6
0	female	adult	1	0	1	0	0	0
1	male	adult	1	0	0	0	0	0
2	female	senior	1	0	0	0	0	0
3	female	youth	1	0	0	1	1	1
4	male	adult	0	0	0	1	0	1
5	female	adult	0	0	1	1	0	1
6	male	adult	1	0	0	1	1	0
7	female	adult	0	0	1	1	0	1
8	male	adult	1	0	1	1	0	1
9	male	youth	1	0	1	0	0	1
10	male	youth	1	1	0	0	0	0
11	female	senior	1	0	0	0	0	1

In [60]:
```
# build a classifier
momc_clf = OneVsRestClassifier(LogisticRegression())
```

In [55]:
```
dummies = pd.get_dummies(the_data[['gender', 'age']]); dummies.head(3)
```

Out[55]:

	gender_female	gender_male	age_adult	age_senior	age_youth
0	1	0	1	0	0
1	0	1	1	0	0
2	1	0	0	1	0

Now we have one binary column for each possible target/label pair. We proceed with the classification as in the multilabel version of multiclass above.

In [62]:
```
momc_clf.fit(the_data[[ 'p1', 'p2', 'p3', 'p4', 'p5', 'p6']], dummies)
```

Out[62]:
```
OneVsRestClassifier(estimator=LogisticRegression(C=1.0, class_weight=None,
 dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
          verbose=0, warm_start=False),
          n_jobs=1)
```

We now have the same issue as before: because the binary classification are now independent, we have lost the contraint that within a target we must have exactly one True column in a row. Also the probabilities for each class within a target must sum to 1 and the most probable class should be predicted.

Here are the predictions and probabilities without normalization.

In [63]:
```
momc_clf.predict(the_data[[ 'p1', 'p2', 'p3', 'p4', 'p5', 'p6']])
```

Out[63]:
```
array([[1, 0, 1, 0, 0],
       [0, 1, 0, 0, 0],
       [0, 1, 0, 0, 0],
       [0, 1, 1, 0, 0],
       [1, 0, 1, 0, 0],
       [1, 0, 1, 0, 0],
       [0, 1, 1, 0, 0],
       [1, 0, 1, 0, 0],
       [1, 0, 1, 0, 0],
       [1, 0, 0, 0, 0],
       [0, 1, 0, 0, 0],
       [1, 0, 0, 0, 0]])
```

Notice that while we get 1 true prediction for gender for each row, for age, there are several predictions with no age label selected. Here are the uncorrected probabilities.

In [67]:
```
probas = momc_clf.predict_proba(the_data[[ 'p1', 'p2', 'p3', 'p4', 'p5', '
p6']]); probas
```

Out[67]:
```
array([[0.50442471, 0.49557529, 0.5860069 , 0.21174829, 0.29236386],
       [0.44558056, 0.55441944, 0.43318958, 0.35235922, 0.34432145],
       [0.44558056, 0.55441944, 0.43318958, 0.35235922, 0.34432145],
       [0.48599851, 0.51400149, 0.52805913, 0.13758658, 0.33036698],
       [0.52934213, 0.47065787, 0.6796947 , 0.19173922, 0.25299465],
       [0.58752575, 0.41247425, 0.79717017, 0.10484792, 0.21039603],
       [0.43045563, 0.56954437, 0.61462369, 0.15552963, 0.30829747],
       [0.58752575, 0.41247425, 0.79717017, 0.10484792, 0.21039603],
       [0.52414981, 0.47585019, 0.70501611, 0.0955136 , 0.21287338],
       [0.56012544, 0.43987456, 0.49826173, 0.18876924, 0.31381023],
       [0.35940499, 0.64059501, 0.35004123, 0.28943614, 0.47120679],
       [0.50135709, 0.49864291, 0.34903474, 0.32032186, 0.36759927]])
```

We take a similar approach to above, but now have to consider that there are 2 targets. We normalize so that within a target (gender is columns 0 and 1, age is columns 3, 4 and 5).

In [103]:
```
def norm_probas(array):
    '''normalize so that row sums equal 1'''
    return array/array.sum(axis=1, keepdims=True)
```

In [76]:
```
normed_probas = np.hstack([norm_probas(probas[:, 0:2]), norm_probas(probas
[:, 2:])]); normed_probas
```

Out[76]:
```
array([[0.50442471, 0.49557529, 0.5375623 , 0.19424327, 0.26819444],
       [0.44558056, 0.55441944, 0.38339763, 0.31185813, 0.30474424],
       [0.44558056, 0.55441944, 0.38339763, 0.31185813, 0.30474424],
       [0.48599851, 0.51400149, 0.5301731 , 0.13813737, 0.33168953],
       [0.52934213, 0.47065787, 0.6044801 , 0.17052148, 0.22499842],
       [0.58752575, 0.41247425, 0.71661278, 0.09425259, 0.18913463],
       [0.43045563, 0.56954437, 0.56991352, 0.14421579, 0.28587069],
       [0.58752575, 0.41247425, 0.71661278, 0.09425259, 0.18913463],
       [0.52414981, 0.47585019, 0.6956917 , 0.09425035, 0.21005796],
       [0.56012544, 0.43987456, 0.49784294, 0.18861058, 0.31354648],
       [0.35940499, 0.64059501, 0.31515821, 0.26059266, 0.42424913],
       [0.50135709, 0.49864291, 0.33659556, 0.30890597, 0.35449847]])
```

In [77]:
```
# check row sums; this time we should have exactly 2 for each row because
there are 2 targets.
normed_probas.sum(axis=1)
```

Out[77]:
```
array([2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.])
```

Now we do the prediction in a similar fashion to the above, selecting the most probable column, resulting in exactly 2 positive predictions per row.

In [ ]:
```
# # build predictions array
# predictions = np.zeros(normed_probas.shape)
# # this can be done directly with numpy, but this code exposes the correc
t logic.
# selected = np.argmax(normed_probas, axis=1)
# for i in range(predictions.shape[0]):
#     predictions[i, selected[i]] = 1
```

In [85]:
```
def prediction_array(probas):
    '''Build an indicator array, selecting most probable output'''
    out = np.zeros(probas.shape)
    maxes = np.argmax(probas, axis=1)
    for i in range(probas.shape[0]):
        out[i, maxes[i]] = 1.0
    return out
```

In [89]:
```
predictions = np.hstack([prediction_array(normed_probas[:, 0:2]), predicti
on_array(normed_probas[:, 2:])]); predictions
```

Out[89]:
```
array([[1., 0., 1., 0., 0.],
       [0., 1., 1., 0., 0.],
       [0., 1., 1., 0., 0.],
       [0., 1., 1., 0., 0.],
```

```
       [1., 0., 1., 0., 0.],
       [1., 0., 1., 0., 0.],
       [0., 1., 1., 0., 0.],
       [1., 0., 1., 0., 0.],
       [1., 0., 1., 0., 0.],
       [1., 0., 1., 0., 0.],
       [0., 1., 0., 0., 1.],
       [1., 0., 0., 0., 1.]])
```

Conclusion

We've seen the similarities and differences between various forms of classifications predicting single and multiple targets. The various methods can produce equivalent results but attention must be paid to the input requirements and output formats of each method in order to produce correct predictions and probabilites.

It should be noted that most sklearn metrics work best with a single target. In some instances they will not complain if given a malformed input but produce misleading output. All produce good results if care is taken to present them with a single target (like age in the above example). Here's an example of F1 and log loss for age.

In [99]:
```
f1_score(dummies.values[:, 2:], predictions[:, 2:], average='micro')
```

Out[99]:
```
0.6666666666666666
```

In [101]:
```
log_loss(dummies.values[:, 2:], predictions[:, 2:])
```

Out[101]:
```
11.512925464970229
```

Final thoughts:

Instead of working with the single array represent all targets and predictors, often people chose to decompose this kind of problem into several classifications then compose the results at the end of the process. There are advantages and disadvantage to both approaches.

As the number of targets increases though, it may become more convenient to use what amounts to a single classifier for the whole problem.