Preprocessing Datasets for Machine Learning

Introduction

```
"If you have extracted the wrong features, no classifier will
work; if you have extracted the right features, any
classifier will work." King-Sun Fu
```

In practice, the data acquired for real-world problems are often incomplete, noisy, and inconsistent. A few percentages of non-clean data points may affect the final performance by a few percentage drops. Better results would be easily achievable if a few preprocessing steps were taken in the right direction. Good data preprocessing is necessary for good machine learning performance, and it is widely accepted that preprocessing takes the bulk of the overall machine learning effort.

In addition to data "cleaning", certain algorithms require data feature properties in certain ways, such as **normalized** and **standardized** to make the method work better. For example, clustering approaches by distance measures require data features to be normalized. The following procedures are common steps in preprocessing:

- Data formatting, cleaning
- Discretization, one-hot encoding
- Data integration and transformation
- Data reduction

Data Formatting and Cleaning

Machine learning frameworks, such as `pandas`, `scikit-learn`, `Weka`, expect dataset files to be in certain formats to be able to process them. The Comma Separated Values **CSV** is one of the most common file formats. Such as the file `breast_cancer_raw.csv` and the first 4 rows,

"age"	"menopause"	"tumor-size"	"inv-nodes"	"node-caps"	"deg-malig"	"breast"	"breast-quad"	"irradiat
44	"premeno"	21	2	"no"	2	"right"	"left_up"	"no"

"age"	"menopause"	"tumor-size"	"inv-nodes"	"node-caps"	"deg-malig"	"breast"	"breast-quad"	"irradiat
46	"premeno"	22	3	"yes"	3	"right"	"left_up"	"no"
46	"premeno"	22	3	"yes"	3	"right"	"left_up"	"no"

When examining datasets, sometimes we see the files might contain artifacts:

- single quotes in double quotes, i.e., `"Cote d'Azor"` or reversed? e.g. `'Cote d'Azor'`
- single quotes to differentiate between strings and values. i.e. `'1'` or `1`
- use of semicolons instead of commas e.g., `1;50;red;` in a row

In addition to the data formats artifacts, we might also see:

- duplicates of data lines (why is this undesired?)
- missing values (marked as `'?'` in Weka or `'NaN'` in pandas for numerical variables)
- incorrect entries (e.g., clerical errors)

Note that framework programs such as Weka learners are mature and strong enough to work with these problems without necessitating us cleaning them by a preprocessing stage. However, if we do the preprocessing ourselves, then we always increase the **quality of the dataset** and this helps the following stages of machine learning pipeline.

Worked Example

Consider the breast cancer dataset file located on the module page. Load it with the `pandas` library and check for (1.) duplicates, (2.) missing values, and (3.) incorrect entries. In the following cells, for each problem that the dataset has, a correction is provided once the situation is determined.

In [1]:
```
# Following avoids a warning for KMeans
%env OMP_NUM_THREADS=2

# Standard libraries we always include
%matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams["figure.dpi"] = 72
from IPython.display import display
import numpy as np
import pandas as pd
import seaborn as sns; sns.set(style="ticks", color_codes=True)
```

```python
# Locate and load the data file
df = pd.read_csv('/EP_datasets/breast_cancer_raw.csv')
print(f'N rows={len(df)} M columns={len(df.columns)}')

# Print some info and plots to get a feeling about the dataset
print(df.dtypes)
```

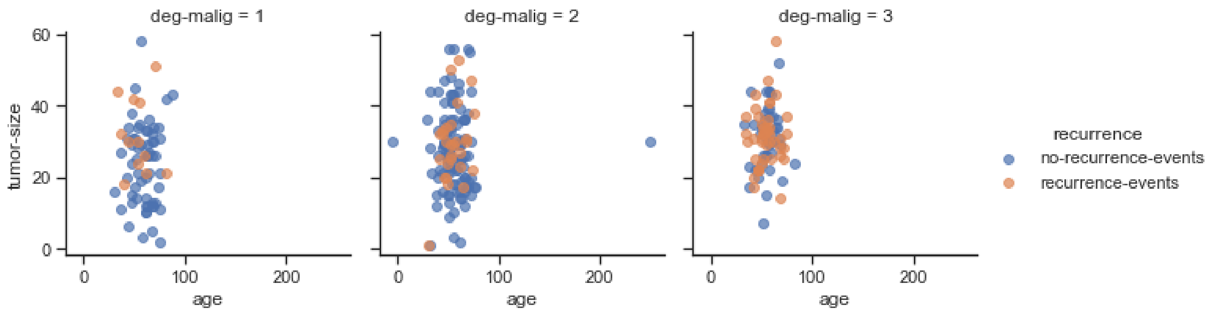
```
env: OMP_NUM_THREADS=2
N rows=298 M columns=10
age            float64
menopause       object
tumor-size     float64
inv-nodes      float64
node-caps       object
deg-malig        int64
breast          object
breast-quad     object
irradiat        object
recurrence      object
dtype: object
```

In [2]:
```python
df.head()
```

Out[2]:

	age	menopause	tumor-size	inv-nodes	node-caps	deg-malig	breast	breast-quad	irradiat	recurren
0	44.0	premeno	21.0	2.0	no	2	right	left_up	no	r recurrenc eve
1	46.0	premeno	22.0	3.0	yes	3	right	left_up	no	recurrenc eve
2	46.0	premeno	22.0	3.0	yes	3	right	left_up	no	recurrenc eve
3	46.0	premeno	22.0	3.0	yes	3	right	left_up	no	recurrenc eve
4	46.0	premeno	22.0	3.0	yes	3	right	left_up	no	recurrenc eve

In [3]:
```python
# Make sure use a '_variable' name to avoid shadowing variable names in ot
def plot_bc(_df, xyscale=None):  # xyscale to use on the plots
    g = sns.FacetGrid(_df, col='deg-malig', hue='recurrence')
    g.fig.set_dpi(72)
    g.map(plt.scatter, 'age', 'tumor-size', alpha=.7)
    g.add_legend()
    if xyscale is not None:
        plt.xlim(xyscale[0], xyscale[1])
        plt.ylim(xyscale[0], xyscale[1])
    plt.show()

plot_bc(df)
```



Observe: In the second plot what is that data point at age 250?? ...Hmmm.

Duplicates

Let's check duplicate values in our dataset.

In [4]:
```
# Check for duplicates, this adds a new column to the dataset
df["is_duplicate"]= df.duplicated()

# Note that when using f-strings, the internal quote character must be dif
print(f"#total= {len(df)}")
print(f"#duplicated= {len(df[df['is_duplicate']==True])}")
```

```
#total= 298
#duplicated= 5
```

In [5]:
```
# Print rows which have True in column 'is_duplicate'
df[df['is_duplicate']==True]
```

Out[5]:

	age	menopause	tumor-size	inv-nodes	node-caps	deg-malig	breast	breast-quad	irradiat	recurre
2	46.0	premeno	22.0	3.0	yes	3	right	left_up	no	recurrer eve
3	46.0	premeno	22.0	3.0	yes	3	right	left_up	no	recurrer eve
4	46.0	premeno	22.0	3.0	yes	3	right	left_up	no	recurrer eve
12	61.0	premeno	29.0	5.0	no	2	right	left_up	yes	recurrer eve
13	61.0	premeno	29.0	5.0	no	2	right	left_up	yes	recurrer eve

In [6]:
```
# Drop the duplicate rows using index - best way to drop in pandas
index_to_drop = df[df['is_duplicate']==True].index
df.drop(index_to_drop, inplace=True)

# Remove the duplicate marker column
```

```
df.drop(columns='is_duplicate', inplace=True)
print(f'#total= {len(df)}')
```

```
#total= 293
```

Observe: Total number of rows (data points) reduced to 293

Missing Values

Let's impute missing values. If we do not handle missing values, then very often the ML algorithms will handle them internally.

The safest and most common approach is to use **mean** (or equally acceptable **median**) for numerical values and **mode** for nominal values to **impute** missing values. Note that a variable is the entire feature or column of data.

Mean: $\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i$

Median: $\tilde{x} = \dfrac{x[|x|/2] + x[|x|/2+1]}{2}$

Mode: $\hat{x} = \underset{x}{\mathbf{argmax}} f(x)$

Mean vs Mode

- Mean is the **average value** of the feature, mode is the **most frequent level** in the feature
- Mean is proper for numerical, mode is proper for nominal features
  - e.g., Mode might end up being 1 in a large column of real numbers when all levels are expressed just once
- Mode is not sensitive to noise or outliers
- Mean value might not exist in the column; mode value is the most frequent level

```
In [7]: # Do we have NaN in our dataset?
        df.isnull().any()
```

Out[7]:
```
age             True
menopause      False
tumor-size      True
inv-nodes       True
node-caps      False
deg-malig      False
breast         False
breast-quad    False
irradiat       False
recurrence     False
dtype: bool
```

In [8]:
```
# We do have NaN - three numerical variables - check first cell, it says f
display(df[df['age'].isnull()])
display(df[df['tumor-size'].isnull()])
display(df[df['inv-nodes'].isnull()])
```

	age	menopause	tumor-size	inv-nodes	node-caps	deg-malig	breast	breast-quad	irradiat	recurrenc
25	NaN	ge40	34.0	1.0	no	1	right	central	no	no recurrenc even
26	NaN	ge40	28.0	1.0	no	2	right	left_up	no	no recurrenc even

	age	menopause	tumor-size	inv-nodes	node-caps	deg-malig	breast	breast-quad	irradiat	recurrenc
27	52.0	premeno	NaN	3.0	no	2	left	left_low	yes	recurrenc even
28	37.0	premeno	NaN	2.0	no	3	left	central	no	no recurrenc even

	age	menopause	tumor-size	inv-nodes	node-caps	deg-malig	breast	breast-quad	irradiat	recurrenc
29	62.0	premeno	10.0	NaN	no	1	right	left_up	no	no recurrenc even

In [9]:
```
# Mean values of numerical columns
means = {c:df[c].mean() for c in df.columns if df[c].dtype != object}

print(f"mean-age= {means['age']}")
print(f"mean-tumor-size= {means['tumor-size']}")
print(f"mean-inv-nodes= {means['inv-nodes']}")

# Impute
df['age'] = df['age'].fillna(means['age'])
df['tumor-size'] = df['tumor-size'].fillna(means['tumor-size'])
```

```
df['inv-nodes'] = df['inv-nodes'].fillna(means['inv-nodes'])

# Check with the previous cell results
display(df.loc[[24,25,26,27,28]])
```

```
mean-age= 56.261168384879724
mean-tumor-size= 28.343642611683848
mean-inv-nodes= 3.5753424657534247
```

	age	menopause	tumor-size	inv-nodes	node-caps	deg-malig	breast	breast-quad	irradiat
24	62.000000	premeno	10.000000	6.0	no	1	right	left_up	no
25	56.261168	ge40	34.000000	1.0	no	1	right	central	no
26	56.261168	ge40	28.000000	1.0	no	2	right	left_up	no
27	52.000000	premeno	28.343643	3.0	no	2	left	left_low	yes
28	37.000000	premeno	28.343643	2.0	no	3	left	central	no

Missing Nominal Values

Finding missing values in nominal variables is more tricky. First, let's look at the nominal variables and then see what kind of unique values these nominal variables have. i.e., this is the **level** of the nominal variable drawn from a finite alphabet. Unless a numerical type (`int64` , `float64` , etc.) `df.dtype` will correspond to an **object**, which is an `object` class after being read into a `DataFrame` from a CSV file.

It is generally accepted to impute the **mode** of the feature when a level missing. Such as `no` for the missing `node-caps` levels as in below.

In [10]:
```
# What are the column types?
df.dtypes
```

Out[10]:
```
age              float64
menopause         object
tumor-size       float64
inv-nodes        float64
node-caps         object
deg-malig          int64
breast            object
breast-quad       object
irradiat          object
recurrence        object
dtype: object
```

In [11]:
```
# Check unique levels and see any marker is used for a missing level
for col in df.columns:
    if df[col].dtype == object:
        print(col, df[col].unique())
```

```
menopause ['premeno' 'ge40' 'lt40']
node-caps ['no' 'yes' '?']
breast ['right' 'left']
breast-quad ['left_up' 'central' 'left_low' 'right_up' 'right_low' '?']
irradiat ['no' 'yes']
recurrence ['no-recurrence-events' 'recurrence-events']
```

The variables `node-caps` and `breast-quad` has `'?'` levels which need to be **imputed** with values to help the preprocessing. Note that some classifiers in `sklearn` do not accept data points with NaN values.

In [12]:
```
# Check the next feature
display(df['node-caps'].value_counts())

print('mode-node-caps', df['node-caps'].value_counts().index[0])
```

```
node-caps
no     227
yes     56
?       10
Name: count, dtype: int64
mode-node-caps no
```

In [13]:
```
# Check the next feature
display(df['breast-quad'].value_counts())

print('mode-breast-quad', df['breast-quad'].value_counts().index[0])
```

```
breast-quad
left_low     111
left_up       99
right_up      33
right_low     26
central       23
?              1
Name: count, dtype: int64
mode-breast-quad left_low
```

In [14]:
```
# Replace '?' with mode - value/level with the highest frequency in the fe
df['node-caps'] = df['node-caps'].replace({'?':'no'})
df['breast-quad'] = df['breast-quad'].replace({'?':'left_low'})
```

In [15]:
```
# Again, check unique levels and see any marker is used or left out for a
for col in df.columns:
    if df[col].dtype == object:
        print (col, df[col].unique())
```

```
menopause ['premeno' 'ge40' 'lt40']
node-caps ['no' 'yes']
breast ['right' 'left']
breast-quad ['left_up' 'central' 'left_low' 'right_up' 'right_low']
irradiat ['no' 'yes']
recurrence ['no-recurrence-events' 'recurrence-events']
```

Incorrect Entries

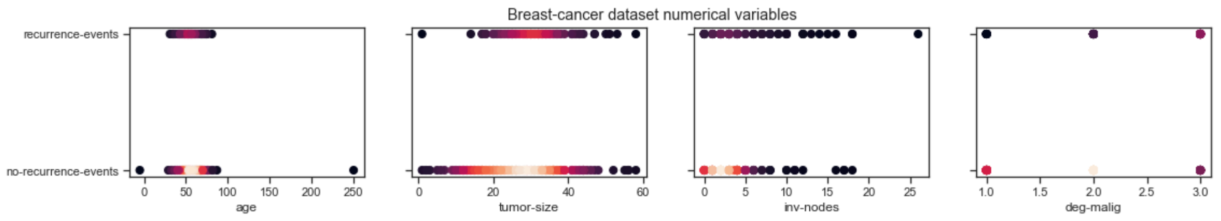
Remember the age value `250` from previous cells?

Finding incorrect entries is more difficult than the previous steps as they truly depend on the data column and **domain knowledge**. For this step, we will look at the plots of numerical columns and figure out possible incorrect entries, such as outliers. Also, subject-matter experts (SME) would help greatly in real-world projects about incorrect entries.

It may not be easy (or possible at all) to correct the incorrect entries, and sometimes the best is dropping that data point.

In [16]:
```
# Let's use kernel density estimation to color the density
from scipy.stats import gaussian_kde

# We will reuse this plotting function later
def plot_bc_numericals(_df):
    fig, axs = plt.subplots(1, 4, figsize=(18, 2.5), sharey=True, dpi=72)
    y = _df['recurrence'].astype('category').cat.codes.ravel()
    xy = np.vstack([_df['age'],y]); z = gaussian_kde(xy)(xy)
    axs[0].scatter(_df['age'], _df['recurrence'], c=z, s=50, edgecolor=Non
    axs[0].set_xlabel('age')
    xy = np.vstack([_df['tumor-size'],y]); z = gaussian_kde(xy)(xy)
    axs[1].scatter(_df['tumor-size'], _df['recurrence'], c=z, s=50, edgeco
    axs[1].set_xlabel('tumor-size')
    xy = np.vstack([_df['inv-nodes'],y]); z = gaussian_kde(xy)(xy)
    axs[2].scatter(_df['inv-nodes'], _df['recurrence'], c=z, s=50, edgecol
    axs[2].set_xlabel('inv-nodes')
    xy = np.vstack([_df['deg-malig'],y]); z = gaussian_kde(xy)(xy)
    axs[3].scatter(_df['deg-malig'], _df['recurrence'], c=z, s=50, edgecol
    axs[3].set_xlabel('deg-malig')
    fig.suptitle('Breast-cancer dataset numerical variables')
    plt.show()
```

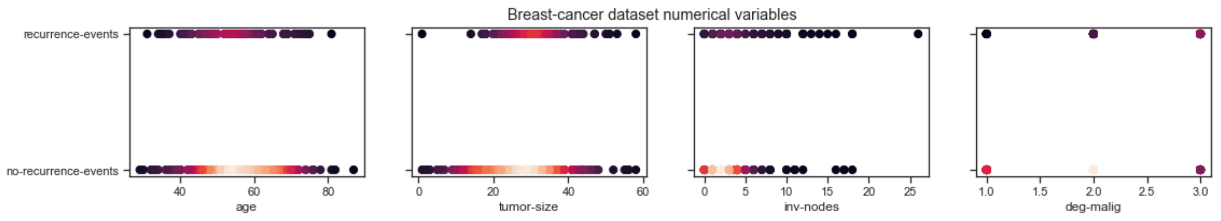
```
plot_bc_numericals(df)
```



In [17]:
```
# Remove that line with the incorrect age=250 and age=-5
dftemp = df.copy()  # use a temporary DataFrame
display(dftemp[dftemp['age']==250])
index_to_drop = dftemp[dftemp['age']==250].index
dftemp.drop(index_to_drop, inplace=True)
index_to_drop = dftemp[dftemp['age']==-5].index
dftemp.drop(index_to_drop, inplace=True)

# Check results
print(f'#total= {len(dftemp)}')
plot_bc_numericals(dftemp)
```

	age	menopause	tumor-size	inv-nodes	node-caps	deg-malig	breast	breast-quad	irradiat	recurre
10	250.0	premeno	30.0	3.0	no	2	left	right_low	yes	recurrei ev

```
#total= 291
```



In [18]:
```
# Let's reset the indices to the dataframe after dropping a few rows
dftemp = dftemp.reset_index(drop=True)
```

Question: What if we don't reset the index?

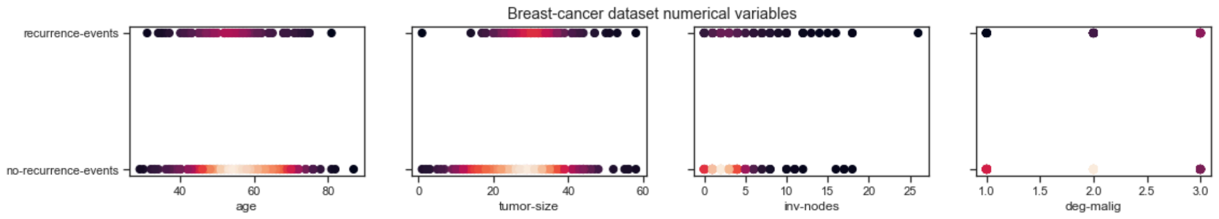
Alternative Data Manipulation

- Use of `apply` method via passing a `lambda`
- Use age as an integer variable, possibly using integer for all 'age' values

In [19]:
```
# Replace anomalous ages with mean when age is less than 0 or greater than
mean_age = int(df['age'].mean())
df['age'] = df['age'].apply(lambda x: mean_age if x<0 or x>120 else x)

# Check results
```

```
print(f'#total= {len(df)}')
plot_bc_numericals(df)
```

```
#total= 293
```



Cleaning Complete

Compare the previous two cells to see the effect of removing the incorrect age entry.

At this point, we are ready to apply a few learners to our data, such as the Random Forest classifier.

Discretization

Discretization is the process where a numerical variable is mapped to some levels by binning. This step is a big research/engineering area in machine learning. Recall that an example was provided in the past modules where the target (dependent) variable was discretized into three levels.

For our purposes, in this step, we will do the post-discretization and apply one hot encoding to a nominal/discretized variable. Note that the variable might be a nominal variable naturally, such as the `'breast'` variable, which takes values from the alphabet {`'left'`, `'right'`}.

Generally, we keep the dependent variable as an integer even if the cardinality is more than 2.

Now, we would like to continue preparing (preprocessing) the dataset further to meet the requirements of the classifier that we would like to use—the Random Forest classifier from the scikit-learn library. This classifier works only on numerical data. Thus, as explained in previous modules, we will convert the nominal variables into one hot-encoded numerical variable.

In [20]:
```
# pandas get_dummies function is the one-hot-encoder
def encode_onehot(_df, _f):
    _df2 = pd.get_dummies(_df[_f], prefix=_f, prefix_sep=' - ', dtype=int)
    _df3 = pd.concat([_df, _df2], axis=1)
    _df3 = _df3.drop([_f], axis=1)
    return _df3

# Print nominal variables
for f in list(df.columns.values):
    if df[f].dtype == object:
        print(f)
```

```
menopause
node-caps
breast
breast-quad
irradiat
recurrence
```

Question: Will we one-hot-encode the dependent variable `'recurrence'`?

In [21]:

```
# Display the original
display(df['menopause'][:10])

# Apply the onehot-encoding method
df_o = encode_onehot(df, 'menopause')

# Check the onehot-encoded version of this feature
cols = []
for f in list(df_o.columns.values):
    if 'menopause' in f:
        cols += [f]
```

```
0     premeno
1     premeno
5        ge40
6        ge40
7     premeno
8     premeno
9     premeno
10    premeno
11    premeno
14       ge40
Name: menopause, dtype: object
```

In [22]:

```
# Display the onehot-encoded
display(df_o[cols][:10])
```

	menopause - ge40	menopause - lt40	menopause - premeno
0	0	0	1
1	0	0	1
5	1	0	0
6	1	0	0
7	0	0	1
8	0	0	1
9	0	0	1
10	0	0	1
11	0	0	1
14	1	0	0

In [23]:
```
# Apply the rest of the nominal features too
df_o = encode_onehot(df_o, 'node-caps')
df_o = encode_onehot(df_o, 'breast')
df_o = encode_onehot(df_o, 'breast-quad')
df_o = encode_onehot(df_o, 'irradiat')
```

In [24]:
```
# Let's check how many features we have
print(f'before={len(df.columns)}, after={len(df_o.columns)}')
```

```
before=10, after=19
```

In [25]:
```
df_o.head()
```

Out[25]:

	age	tumor-size	inv-nodes	deg-malig	recurrence	menopause - ge40	menopause - lt40	menopause - premeno	nod ca -
0	44.0	21.0	2.0	2	no-recurrence-events	0	0	1	
1	46.0	22.0	3.0	3	recurrence-events	0	0	1	
5	56.0	19.0	4.0	1	no-recurrence-events	1	0	0	
6	58.0	41.0	0.0	2	recurrence-events	1	0	0	
7	53.0	36.0	0.0	3	no-recurrence-events	0	0	1	

Evaluation

Next, let's classify the preprocessed dataset using the following strategies:

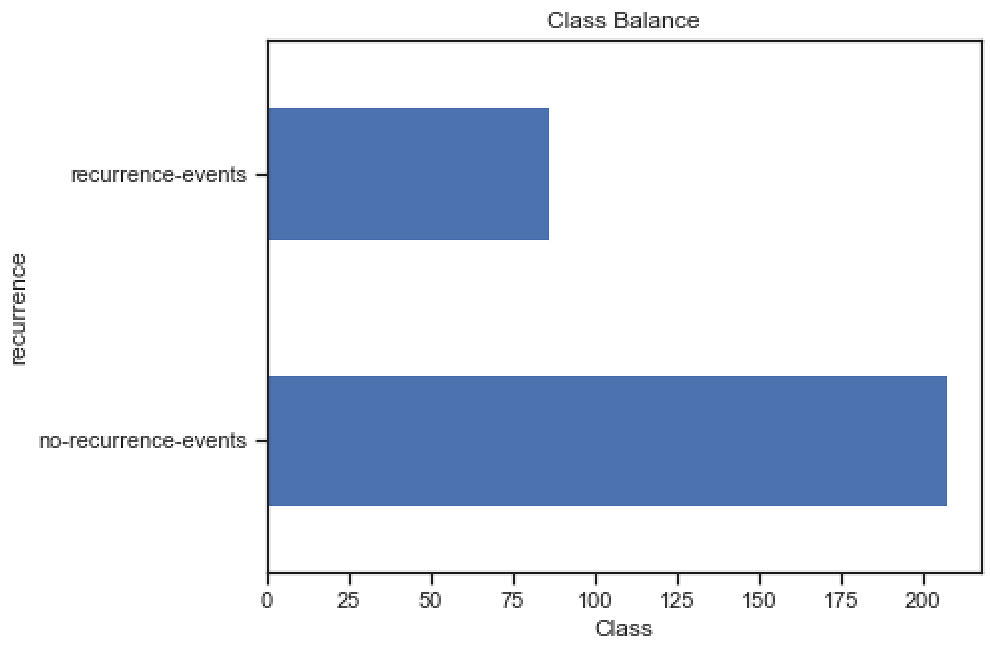
1. 80% random train-test split
2. Leave-one-out
3. 10-fold cross-validation
4. Stratified 10-fold cross-validation

Note that the target variable is binary, predicting whether the cancer will recur or not. Clearly, this dataset has ground truth captured from the data source, or in other words, dataset is pre-labeled, or carry the ground truth. Thus, we will employ **supervised learning**.

Important: Do not forget to remove the target (predicted, dependent) variable from `X`. Remember that the `Dataframe` we are working on already has the target variable, and we will move it to the `y` vector.

In [26]:
```
# Show that the dependent variable is unbalanced
display(df['recurrence'].value_counts())

df['recurrence'].value_counts().plot(kind='barh', xlabel='Class', title='C

# The semicolon above causes hiding the result of the last expression in t
```

```
recurrence
no-recurrence-events    207
recurrence-events        86
Name: count, dtype: int64
```



In [27]:
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score  # f1_score can be used too
from sklearn.model_selection import KFold, StratifiedKFold, train_test_spl

# Converting from class labels integers
# df_o['recurrence'] = df_o['recurrence'].replace({'recurrence-events':1,

# We will reuse the classifier function below
def rf_train_test(_X_tr, _X_ts, _y_tr, _y_ts):
    # Create a new random forest classifier, with working 4 parallel cores
    rf = RandomForestClassifier(n_estimators=200, max_depth=5, random_stat
    # Train on training data
    rf.fit(_X_tr, _y_tr)
    # Test on training data
    y_pred = rf.predict(_X_ts)
    # Return more proper evaluation metric
    # return f1_score(_y_ts, y_pred, pos_label='recurrence-events', zero_d
    # Return accuracy
    return accuracy_score(_y_ts, y_pred)
```

In [28]:
```
# Prepare the input X matrix and target y vector
X = df_o.loc[:, df_o.columns != 'recurrence'].values
y = df_o.loc[:, df_o.columns == 'recurrence'].values.ravel()
```

In [29]:
```
# Sanity check
print(y[:10])
```

```
['no-recurrence-events' 'recurrence-events' 'no-recurrence-events'
 'recurrence-events' 'no-recurrence-events' 'recurrence-events'
 'no-recurrence-events' 'no-recurrence-events' 'no-recurrence-events'
 'no-recurrence-events']
```

80% Random Train-test Split Evaluation

In [30]:
```
# 80% split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20,
rf_train_test(X_train, X_test, y_train, y_test)
```

Out[30]:
```
0.7966101694915254
```

Question: What will be the performance (i.e., accuracy) when we run the above cell again? Will you see any variations?

In [31]:
```
# Run 10 times
for i in range(10):
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.
    print(rf_train_test(X_train, X_test, y_train, y_test))
```

```
0.7966101694915254
0.7796610169491526
0.7966101694915254
0.7627118644067796
0.6610169491525424
0.7627118644067796
0.7457627118644068
0.7288135593220338
0.7627118644067796
0.7457627118644068
```

Important: As the training and testing partition changes, the performance follows respectively.

Question: How can we measure the performance so that we can be sure of reporting it right?

In [32]:
```
# Run 100 times and collect statistics
Accuracies = []
for _ in range(100):
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.
    Accuracies += [rf_train_test(X_train, X_test, y_train, y_test)]

print(f'80% train-test split accuracy is {np.mean(Accuracies):.3f} {chr(17
```

```
80% train-test split accuracy is 0.797 ±0.0000
```

Leave-one-out Evaluation

Leave-one-out evaluation keeps a single data point and label for the test and uses all except the test vector for training. Then, the evaluation process repeats this for each of the remaining data points, having a total number of N accuracies.
The `sklearn` API says `train` and `test` require a 2D `X` and 1D `y` even when there is only one data point. The below code generates the test vectors properly.

This evaluation is helpful when data is scarce, such as in the Bioinformatics and Medical fields.

In [33]:
```
%%time

# Leave one out testing - this takes relatively longer
N = X.shape[0]
Accuracies = []
for i in range (0,N):
    # Keep the 2D vector for the single test data point X
    X_test = X[i].reshape(1, -1)
    X_train = np.delete(np.array(X, copy=True), i, axis=0)

    # Keep the 1D vector for the single test label y
    y_test = [y[i]]
    y_train = np.delete(np.array(y, copy=True), i, axis=0)
    Accuracies += [rf_train_test(X_train, X_test, y_train, y_test)]

# Sanity
print(f'Leave-one-out accuracy N= {N}, #accuracies= {len(Accuracies)}')

# Score
print(f'Leave-one-out accuracy is {np.mean(Accuracies):.3f} {chr(177)}{np.
```

```
Leave-one-out accuracy N= 293, #accuracies= 293
Leave-one-out accuracy is 0.737 ±0.4402
CPU times: total: 59.7 s
Wall time: 50.6 s
```

10-fold Cross-validation Evaluation

In [34]:

```
# 10-fold cross-validation
Accuracies = []
kfold = KFold(n_splits=10,shuffle=False)
for train_index, test_index in kfold.split(X, y):
    acc = rf_train_test(X[train_index], X[test_index], y[train_index], y[t
    Accuracies += [acc]

print(f'10-fold cross-validation accuracy is {np.mean(Accuracies):.3f} {ch
```

```
10-fold cross-validation accuracy is 0.747 ±0.0595
```

Stratified 10-fold Cross-validation Evaluation

In [35]:

```
def eval_classifier(_X, _y, _niter):
    accs = []
    for _ in range(_niter):
        kf = StratifiedKFold(n_splits=10, shuffle=True, random_state=_)
        for tr_ix, ts_ix in kf.split(_X, _y):
            accuracy = rf_train_test(_X[tr_ix], _X[ts_ix], _y[tr_ix], _y[t
            accs += [accuracy]

    print(f'Stratified 10-fold CV acc={np.mean(accs):.3f} {chr(177)}{np.st

eval_classifier(X, y, 1)
eval_classifier(X, y, 10)
```

```
Stratified 10-fold CV acc=0.734 ±0.0734 with 1 iterations
Stratified 10-fold CV acc=0.738 ±0.0656 with 10 iterations
```

Note the above performance results for discussion in the following cells.

Question: What are the differences between these four evaluation methods?

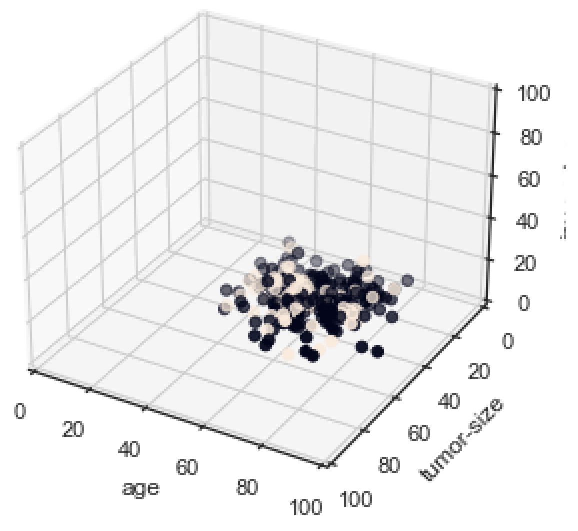
Data Transformation

Now that we have preprocessed and used the data for classification, we can move on to other interesting problems.

Imagine that we did not have the ground truth, so supervised learning was not possible. A natural approach in this case is clustering the data to see if there are some patterns or models we can develop that explain the cancer behavior. We will attempt to answer questions like *"Is there a direct relation between menopause and cancer?"*

First, let's draw some plots where the x, y, and z-dimensions are `'age'`, `'tumor-size'`, `'inv-nodes'` and color is `'recurrence'`.

In [36]:
```
from mpl_toolkits.mplot3d import Axes3D

# Deep copy original dataframe
df2 = df.copy()

# Convert every feature to numbers
df2['recurrence'] = df['recurrence'].astype("category").cat.codes

df2['menopause'] = df['menopause'].astype("category").cat.codes.astype('fl
df2['node-caps'] = df['node-caps'].astype("category").cat.codes.astype('fl
df2['breast'] = df['breast'].astype("category").cat.codes.astype('float')
df2['breast-quad'] = df['breast-quad'].astype("category").cat.codes.astype
df2['irradiat'] = df['irradiat'].astype("category").cat.codes.astype('floa

df2['deg-malig'] = df['deg-malig'].astype('float')

def draw3d(_df, _mn, _mx):
    fig = plt.figure(dpi=72)
    ax = fig.add_subplot(111, projection='3d')
    ax.set_xlim3d(_mn, _mx)
    ax.set_ylim3d(_mn, _mx)
    ax.set_zlim3d(_mn, _mx)
    ax.set_ylim(ax.get_ylim()[::-1])
    ax.scatter(_df['age'], _df['tumor-size'], _df['inv-nodes'], c=_df['rec
    ax.set_xlabel('age'); ax.set_ylabel('tumor-size'); ax.set_zlabel('inv-

draw3d(df2, 0, 100)
```



Question: Do the dimensions `'age'`, `'tumor-size'`, `'inv-nodes'` look fine in the above 3D plot?

Answer: The features are clumped and do not nicely occupy the $[0 - 100]$ range, i.e., we do not see a spherical cluster shape.

Let's cluster the cancer data without using the ground truth. We have to convert the nominal variables to numerical by using the category codes, as we applied to `'recurrence'` variable.

Important: Make sure every variable is the same type, e.g. `float32`.

Important: Note that the values `'recurrence'` took {`0`, `1`}, and by looking at the 3d plot above, can we easily find out which values (`0` or `1`) corresponds to `'recurrence-events'` levels?

In [37]:
```python
from sklearn.cluster import KMeans

def kmeans(_X, _y, niter):  # do it niter times to collect statistics
    accuracies = []
    for _ in range(niter):
        # We know that there are two levels in target variable - thus n_cl
        km = KMeans(n_clusters=2, random_state=0, n_init=10)
        clusters = km.fit_predict(_X)
        accuracies += [accuracy_score(_y, clusters)]
    return np.mean(accuracies)

X = df2.loc[:, df2.columns != 'recurrence'].values
y = df2.loc[:, df2.columns == 'recurrence'].values.ravel()

print(f'Clustering error= {kmeans(X, y, 100):.3f}')
```

```
Clustering error= 0.471
```

Above performance is not very good as the error is almost equivalent to random choice, which would be $\frac{1}{2}$ since we have 2 classes.

Normalization and Standardization

Mapping the values of a column to the $[0, 1]$ range is normalization:

$$\frac{x_i - \min(x)}{\max(x) - \min(x)}$$

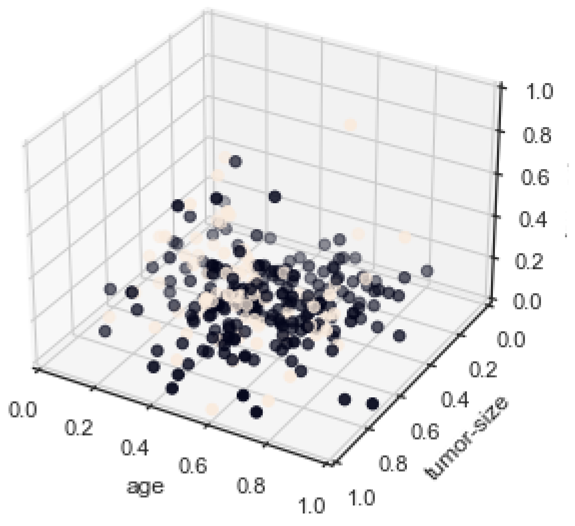
Standardization is mapping the values to a 0-mean 1-standard-deviation distribution: $\frac{x_i - \text{mean}(x)}{\text{stdev}(x)}$

Normalization makes the **optimization surface** more **spherical**, which helps the optimizer use each feature with equal importance. This is especially important and helps for **distance** based methods, such as neural networks, SVM, etc. Note that some probabilistic methods are Naive Bayes, decision trees, etc.

Let's try two scalers from `sklearn.preprocessing`

1. Normalization `MinMaxScaler()`
2. Standardization `scale()`

In [38]:
```
from sklearn import preprocessing

min_max_scaler = preprocessing.MinMaxScaler()
df2[['age', 'tumor-size', 'inv-nodes']] = min_max_scaler.fit_transform(df2

draw3d(df2, 0, 1)
```



By normalizing the values through expansion and contraction to $[0, 1]$ we achieve the **distance** between the data points are in the same "range" or unit. Thus, the distance metrics like Euclidean distance will weigh each **dimension** or feature **equally**.

Example: Imagine a dataset which has speed in miles $[0, 100]$ and time traveled in seconds $[0, 43200]$ (12 hours max). A proper approach would be mapping both features into $[0, 1]$ scale to treat the feature space spherically. For actual feature values, an inverse transformation can be used to map back to the original units (for example, to be presented to the user).

A distance metric d in M dimensions (`Dataframe` has M number of columns) such as Euclidean $d_{ik} = \sqrt{\sum_{j=0}^{M}(x_{ij} - x_{kj})^2}$

Clustering algorithms, for example, use some form of distance metric, such as Euclidean distance, between pairs of data points.

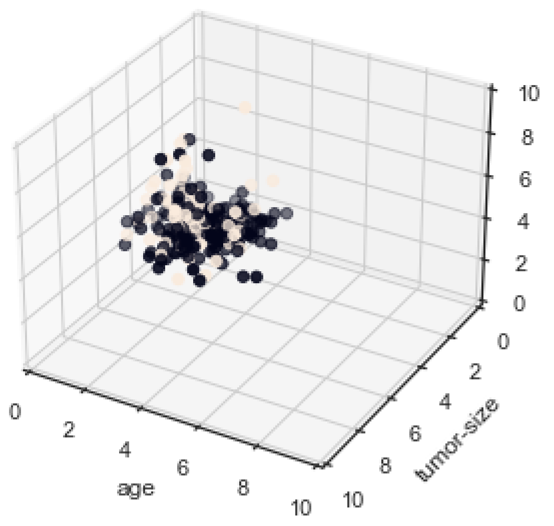
As seen from the above example, normalization of variables is necessary for clustering.

In [39]:
```
df2[['deg-malig', 'breast-quad']] = min_max_scaler.fit_transform(df2[['deg

X = df2.loc[:, df2.columns != 'recurrence'].values
y = df2.loc[:, df2.columns == 'recurrence'].values.ravel()

print(f'Clustering error= {kmeans(X, y, 100):.3f}')
```

```
Clustering error= 0.509
```

And now standardization.

In [40]:
```
df2[['age', 'tumor-size', 'inv-nodes']] = preprocessing.scale(df2[['age',

draw3d(df2, 0, 10)
```



In [41]:
```
df2.head()
```

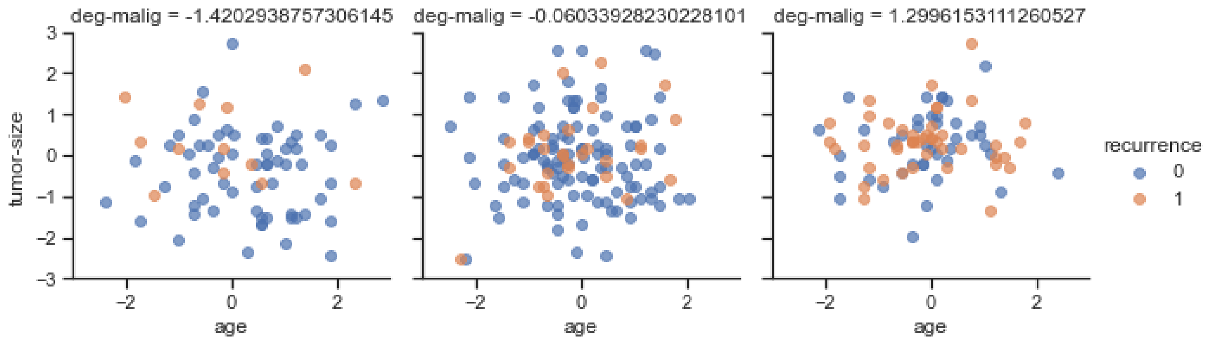
Out[41]:

	age	menopause	tumor-size	inv-nodes	node-caps	deg-malig	breast	breast-quad	irrad
0	-1.089465	2.0	-0.679846	-0.425865	0.0	0.5	1.0	0.50	(
1	-0.904923	2.0	-0.587270	-0.155533	1.0	1.0	1.0	0.50	(
5	0.017786	0.0	-0.864999	0.114798	0.0	0.0	1.0	0.00	(
6	0.202328	0.0	1.171677	-0.966529	0.0	0.5	0.0	0.25	(
7	-0.259027	2.0	0.708796	-0.966529	1.0	1.0	1.0	0.25	

In [42]:
```
df2[['deg-malig', 'breast-quad']] = preprocessing.scale(df2[['deg-malig',

X = df2.loc[:, df2.columns != 'recurrence'].values
y = df2.loc[:, df2.columns == 'recurrence'].values.ravel()

print(f'Clustering error= {kmeans(X, y, 100):.3f}')
```

```
Clustering error= 0.512
```

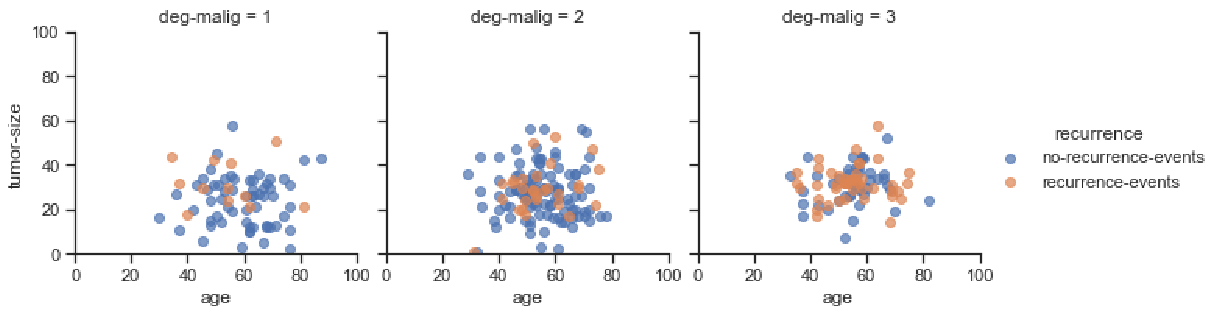
In [43]:
```
# Scaled
plot_bc(df2, xyscale=[-3,3])
```



Question: Do you see any difference/improvement in the variables compared to the first set of plots in cell 1, repeated below?

Answer: Shapes are the same, but axis scales are different.

In [44]:
```
# Original
plot_bc(df, xyscale=[0,100])
```



Note that after variable transformation, variables become more spherical or Gaussian-like. Still, the levels or data points do not correspond to any meaningful value in the domain knowledge the dataset originally belonged to. For example, `'deg–malign'` had three levels {1, 2, 3}, which probably meant something to doctors dealing with cancer patients. However, depending on the dataset, such transformations make a difference, albeit with a few percentage improvements in the performance.

Data Reduction

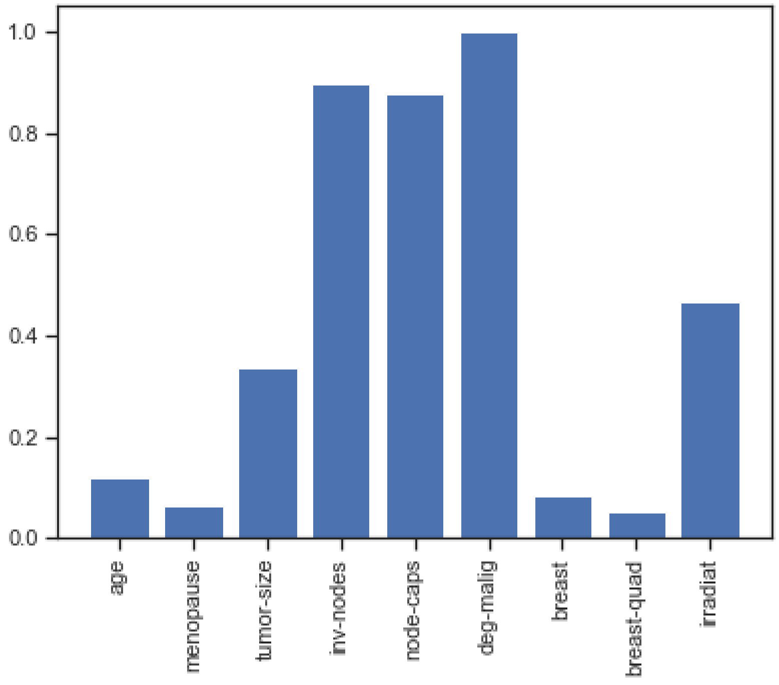
Reducing the data helps in a few ways:

- Faster method run-time, such as training
- More generalized models decrease overfitting
- Simpler models that make more sense to the domain expert or subject-matter expert (SME)
- In some cases, better accuracy performance - not necessarily always happens

Feature ranking and **feature selection** are common stages executed after data cleaning and preprocessing. In the following cells, we will examine the variable rankings using **Univariate Feature Selection**.

In [45]:
```
from sklearn.feature_selection import SelectPercentile, f_classif


selector = SelectPercentile(f_classif, percentile=10)
```

```
# Fit the data
selector.fit(X, y)
scores = -np.log10(selector.pvalues_)
scores /= scores.max()

# Display
cols = list(df2.loc[:, df2.columns != 'recurrence'].columns.values)
y_pos = np.arange(len(cols))
plt.bar(y_pos, scores)
plt.xticks(y_pos, cols, rotation=90)
plt.show()
```



Question: Can we drop `'age'`, `'menopause'`, `'breast'`, `'breast-quad'` variables and redo the classification evaluation without a performance loss?

In [46]:
```
df3 = df2.copy()
df3.drop(columns='age', inplace=True)
df3.drop(columns='menopause', inplace=True)
df3.drop(columns='breast', inplace=True)
df3.drop(columns='breast-quad', inplace=True)

X = df3.loc[:, df3.columns != 'recurrence'].values
y = df3.loc[:, df3.columns == 'recurrence'].values.ravel()
```

In [47]:
```
eval_classifier(X, y, 10)
```

```
Stratified 10-fold CV acc=0.755 ±0.0610 with 10 iterations
```

Wow!__ The performance accuracy did not drop, and we have fewer data columns now.

Note that we had standardized the data in the previous steps. Let's return to the original dataset after the cleaning was completed.

In [48]:
```
df4 = df_o.copy()
df4.drop(columns='age', inplace=True)

# 'menopause' was onehot-encoded
for col in df4.columns.values:
    if 'menopause' in col:
        df4.drop(columns=col, inplace=True)

# 'breast' was onehot-encoded
for col in df4.columns.values:
    if 'breast' in col:
        df4.drop(columns=col, inplace=True)

# 'breast-quad' was onehot-encoded
for col in df4.columns.values:
    if 'breast-quad' in col:
        df4.drop(columns=col, inplace=True)

X = df4.loc[:, df4.columns != 'recurrence'].values
y = df4.loc[:, df4.columns == 'recurrence'].values.ravel()
```

In [49]:
```
eval_classifier(X, y, 10)
```

```
Stratified 10-fold CV acc=0.755 ±0.0584 with 10 iterations
```

In [50]:
```
X = df_o.loc[:, df_o.columns.isin(['deg-malig', 'inv-nodes', 'node-caps -
X.shape
```

Out[50]:
```
(293, 4)
```

In [51]:
```
eval_classifier(X, y, 10)
```

```
Stratified 10-fold CV acc=0.758 ±0.0633 with 10 iterations
```

More success! The performance accuracy increased! Or did we bias it?

Harder Question: Do you accept the performance increase as valid? Or would you attribute it to the variance of error?

Question: What is the most important takeaway in this effort?

References

1. Raschka, Sebastian, et al. Machine Learning with PyTorch and Scikit-Learn: Develop machine learning and deep learning models with Python. Packt Publishing Ltd, 2022.

Exercises

Exercise 1. Change the cross-validation from 10 folds to 3 folds and report its evaluation performance. Do you think 3-fold CV is better than 10-fold CV?

Exercise 2. Use only one feature/column in your classifier model to predict cancer. Report the best 10-fold CV performance.

Exercise 3. Use only `'age'` feature in your classifier model to predict cancer. Report the best 10-fold CV performance.

Exercise 4. Change the `accuracy_score` to `f1_score` and repeat previous exercises. Report findings.

In [52]:
```
%%html
<style>
    table {margin-left: 0 !important;}
    p {font-family: verdana;}
    li {font-family: verdana;}
    div {font-size: 10pt;}
</style>
<!-- Display markdown tables left oriented in this notebook. -->
```
