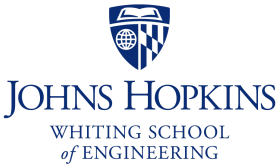
Regression



Introduction

When **numerical values** are **predicted** for a dependent variable then the process is called **regression**. Recall that the supervised learning is predicting class labels or categories of a dependent variable. Thus, regression and supervised learning are complementary approaches. The regression model output is continuous, non-nominal, or real-valued. The regression process can be considered as a function approximation or a curve fit.

- **Linear regression**:
  Fit the data with the *best* line which goes through the input values.
- **Multiple** linear regression:
  The input X is $\in \mathbb{R}^M$ where $M > 1$.
  For linear regression $M = 1$.
  Fit the data with the *best* **hyperplane** which goes through the input values.
- **Multivariate** linear regression:
  The dependent variable y has a dimension > 1, i.e. $y \in \mathbb{R}^K$ where $K > 1$.
  For multiple linear regression $K = 1$.

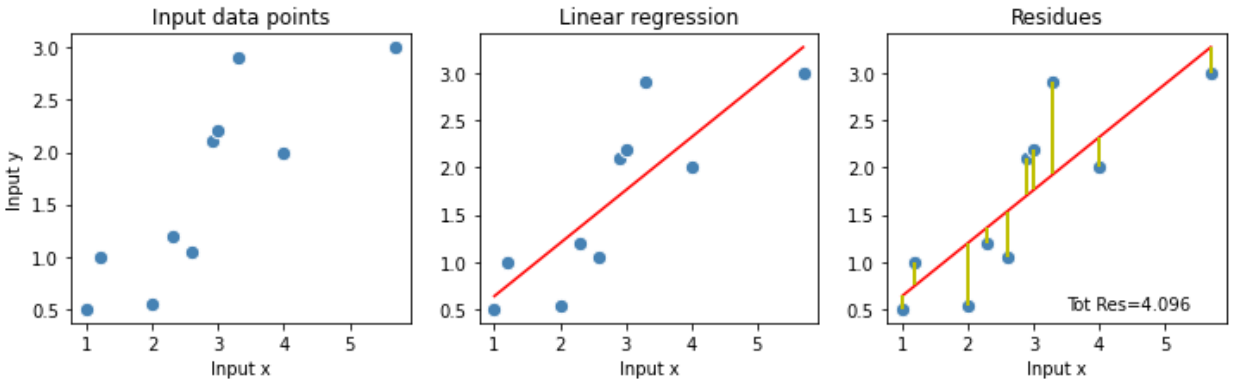
The numerical difference between the predicted point $\hat{y}$ (`y_pred`) and the actual observation y (input `y`) is the **residue**.
Finding the best line or hyperplane is an optimization problem, thus we need a cost function.
Such as $\text{cost} = \sum_i (\text{predicted}_i - \text{actual}_i)^2$ Or, $\text{cost} = \sum_i (\text{residue}_i)^2$.
Minimize the cost function to find the best line.

For the linear regression problem, where the line is given by $\boxed{y = b_0 + b_1 x}$, we can solve the optimization problem, minimize the cost C, where $C = \sum_i^N (\hat{y}_i - y_i)^2$ by taking the derivative, setting to zero and solving for b_0 and b_1.
Then, $b_1 = \frac{N\sum xy - \sum x \sum y}{N\sum x^2 - (\sum x)^2}$ and $b_0 = \frac{\sum y - b_1 \sum x}{N}$.

A perfect linear fit would have a total residue of 0. Normally the input data would not result a perfect line fit. The following is an example.

In [1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams["figure.dpi"] = 72
import numpy as np
```

```
# Sample input
x1 = np.array([1, 1.2, 2., 2.3, 2.6, 2.9, 3., 3.3, 4., 5.7])
y1 = np.array([0.5, 1., 0.55, 1.2, 1.05, 2.1, 2.2, 2.9, 2., 3.])

# Solve using above equations
def linearregression(_x:np.ndarray, _y:np.ndarray) -> np.ndarray:
    n = len(_x)
    b1 = (n*np.sum(_x*_y)-np.sum(_x)*np.sum(_y)) / (n*np.sum(_x*_x)-np.sum(_x)*
    b0 = (np.sum(_y)-b1*np.sum(_x)) / n
    return b0 + b1*_x

y1_pred = linearregression(x1, y1)

# Error
totresidue = np.sum(np.abs(y1-y1_pred))

# Plot
plt.figure(figsize=(12, 3), dpi=72)

plt.subplot(1, 3, 1)
plt.scatter(x1, y1, c='steelblue', edgecolor='white', s=70)
plt.xlabel('Input x'); plt.ylabel('Input y'); plt.title('Input data points')

plt.subplot(1, 3, 2)
plt.scatter(x1, y1, c='steelblue', edgecolor='white', s=70)
plt.plot(x1, y1_pred, c='red')
plt.xlabel('Input x'); plt.title('Linear regression')

plt.subplot(1, 3, 3)
plt.xlabel('Input x'); plt.title('Residues')
plt.scatter(x1, y1, c='steelblue', edgecolor='white', s=70)
plt.plot(x1, y1_pred, c='red')

# Residue lines
for i in range(len(x1)):
    plt.vlines(x=x1[i], ymin=min(y1[i],y1_pred[i]), ymax=max(y1[i],y1_pred[i]),

# Annotate residue amount
plt.text(3.5, 0.5, f"Tot Res={totresidue:.3f}", fontsize=10)

plt.show()
```



For the multiple linear regression problem the hyperplane is defined by

$$y = b_0 + b_1x_1 + b_2x_2 + \ldots + b_Mx_M \quad \ldots\ldots\ldots\ldots\ldots\ldots (Eq.1)$$

Explicit solutions for the coefficients (weights) $\{b_i\}$ exists for the cost function C above. There are also iterative techniques like **Gradient Descent** to find the optimum solution. Regression in this form is a **convex optimization** problem and a global minimum exists.

General form of linear regression problem: $y = \sum_{i=0}^{N} w_i \phi_i(X)$, where $\phi_i(X)$ are non-linear **basis functions**.

Question: Can you see similarities between the (Eq.1), i.e. $y = b_0 + b^\top x$, and the mathematical model of a Perceptron?

Advantages of Regression

Regression outputs a real-valued variable, and sometimes it may be desirable to have a numerical value as the predictor. Also, the important independent variables (features) x_i would have the largest b_i compared to other features. Thus, linear regression can be used to find the best **indicator** or used for feature ranking. Best indicator variables have the highest correlation with the target variable. The values can be either positive or negative and the magnitude shows the correlation index.

Important: Correlation does not imply or show **causality**.

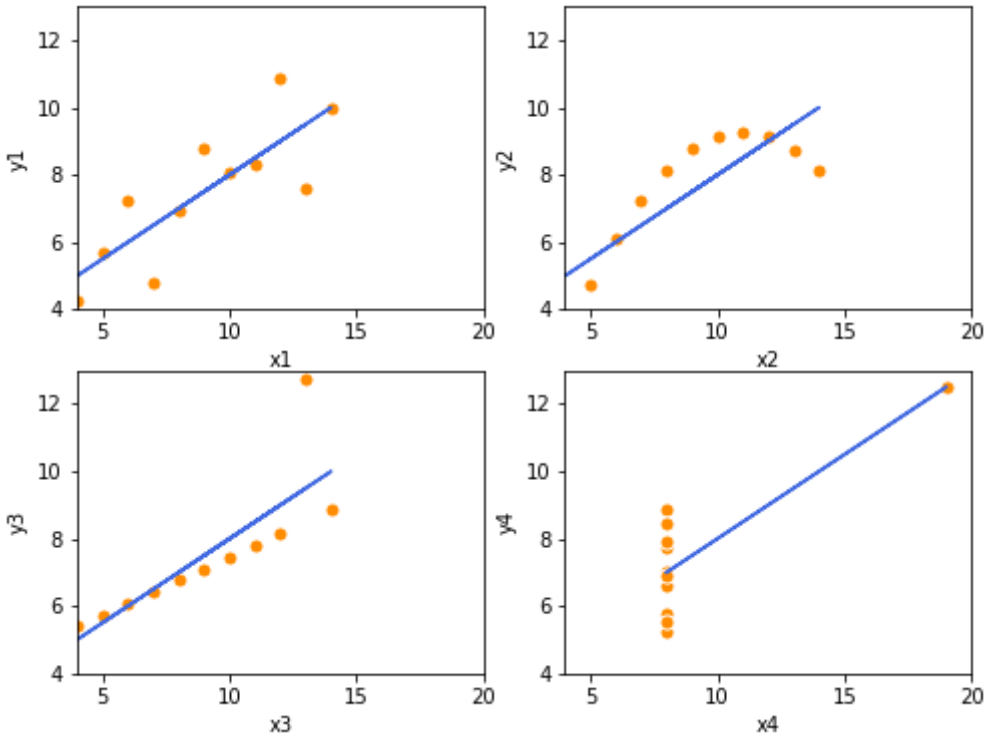
Disadvantages of Regression

The "best" line might be hard to determine sometimes. Consider the following **Anscombe's Quartet** examples:

(ref: Rousselet, G. A., et al. "Beyond differences in means: robust graphical methods to compare two groups in neuroscience." Eur. J. of Neuroscience 46.2 (2017): 1738-1748.)

In [2]:
```
# Anscombes Quartet
x1=np.array([10,8,13,9,11,14,6,4,12,7,5])
y1=np.array([8.04,6.95,7.58,8.81,8.33,9.96,7.24,4.26,10.84,4.82,5.68])

x2=np.array([10,8,13,9,11,14,6,4,12,7,5])
y2=np.array([9.14,8.14,8.74,8.77,9.26,8.1,6.13,3.1,9.13,7.26,4.74])

x3=np.array([10,8,13,9,11,14,6,4,12,7,5])
y3=np.array([7.46,6.77,12.74,7.11,7.81,8.84,6.08,5.39,8.15,6.42,5.73])

x4=np.array([8,8,8,8,8,8,8,19,8,8,8])
y4=np.array([6.58,5.76,7.71,8.84,8.47,7.04,5.25,12.5,5.56,7.91,6.89])

# Plot
plt.figure(figsize=(8, 6), dpi=72)

plt.subplot(2, 2, 1)
plt.scatter(x1, y1, c='darkorange', edgecolor='white', s=50)
y1_pred=linearregression(x1, y1)
plt.plot(x1, y1_pred, c='royalblue')
plt.xlim(4,20); plt.ylim(4,13)
plt.xlabel('x1'); plt.ylabel('y1')
```

```
plt.subplot(2, 2, 2)
plt.scatter(x2, y2, c='darkorange', edgecolor='white', s=50)
y2_pred=linearregression(x2, y2)
plt.plot(x2, y2_pred, c='royalblue')
plt.xlim(4,20); plt.ylim(4,13)
plt.xlabel('x2'); plt.ylabel('y2')

plt.subplot(2, 2, 3)
plt.scatter(x3, y3, c='darkorange', edgecolor='white', s=50)
y3_pred=linearregression(x3, y3)
plt.plot(x3, y3_pred, c='royalblue')
plt.xlim(4,20); plt.ylim(4,13)
plt.xlabel('x3'); plt.ylabel('y3')

plt.subplot(2, 2, 4)
plt.scatter(x4, y4, c='darkorange', edgecolor='white', s=50)
y4_pred=linearregression(x4, y4)
plt.plot(x4, y4_pred, c='royalblue')
plt.xlim(4,20); plt.ylim(4,13)
plt.xlabel('x4'); plt.ylabel('y4')

plt.show()
```



Question: Do you think a linear line fit is working for the above datasets?

Perhaps we should apply **non-linear** regression (i.e. quadratic) for the second dataset and remove anomalies or **outliers** in third and fourth datasets?

Note that all these four datasets are created artificially, and their statistical measures such as mean, standard deviation, correlation and their linear regression lines are same. Thus, a line fit may not be the best model for a particular dataset.

Now, let's check the regression errors for evaluation:

- Mean Absolute Error, MAE = $\frac{1}{N}\sum_{i=1}^{N}|\hat{y}_i - y_i|$
- Mean Squared Error, MSE = $\frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2$
- Mean Absolute Percentage Error, MAPE = $\frac{1}{N}\sum_{i=1}^{N}|\frac{\hat{y}_i - y_i}{y_i}|$
- Mean Percentage Error, MPE = $\frac{1}{N}\sum_{i=1}^{N}\frac{\hat{y}_i - y_i}{y_i}$
- R-Squared = $\frac{SSR}{SST} = \frac{\sum_{i=1}^{N}(\hat{y}_i - \bar{y})^2}{\sum_{i=1}^{N}(y_i - \bar{y})^2}$

MAPE and MPE are more robust to outliers. R-squared, normalized MSE, measures the goodness of fit or best-fit line.

Question: Do you think MSE for the above 4 datasets will be different?

In [3]:
```
from sklearn.metrics import r2_score

def mae(_y, _y_pred):
    return (len(_y)**-1) * np.sum(np.abs(_y_pred-_y))

def mse(_y, _y_pred):
    return (len(_y)**-1) * np.sum((_y_pred-_y)**2)

def mape(_y, _y_pred):
    return (len(_y)**-1) * np.sum(np.abs((_y_pred-_y)/_y))

def mpe(_y, _y_pred):
    return (len(_y)**-1) * np.sum((_y_pred-_y)/_y)

# Present all 4 metrics
print('For each of the linear regression above,')
print ('  MAE    MSE   MAPE    MPE     R2')
r = map( lambda y,z : (f'{mae(y,z):.3f}  {mse(y,z):.3f}  {mape(y,z):.3f}  {mpe(
                       (y1,y2,y3,y4),
                       (y1_pred,y2_pred,y3_pred,y4_pred)
       )
print('\n'.join(r))
```

```
For each of the linear regression above,
  MAE    MSE   MAPE    MPE     R2
0.837  1.251  0.121  0.026  0.667
0.968  1.252  0.157  0.042  0.666
0.716  1.251  0.080  0.011  0.666
0.903  1.249  0.135  0.027  0.667
```

Logistic Regression

The logistic regression can be used to predict the probability of a **dichotomous** outcome that has two values, such as a binary dependent variable. The prediction can be based on the use of one or several predictors, such as in classification. A linear regression is not proper for predicting the value of a binary variable for two reasons:

- A linear regression will predict values outside the acceptable range of $[0, 1]$

- Dichotomous experiments can only have one of two possible values for each data point, so that the residuals will not be normally distributed about the predicted line

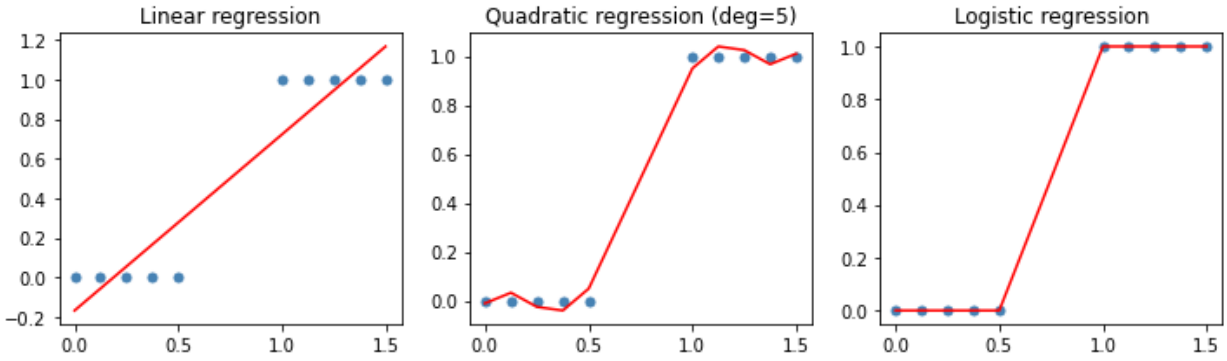
A logistic regression produces a logistic curve, which is limited to the values between 0 and 1. Logistic regression is similar to a linear regression, but the curve is constructed using the natural logarithm of the "odds" of the target variable, rather than the probability. Moreover, the predictors do not have to be normally distributed or have equal variance in each group.

Logistic regression involves fitting the data to a sigmoid function: $p = \frac{1}{1 + \exp^{-y}}$

The following demonstrates the regression for y-values that take 0 or 1.

In [4]:
```
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from scipy.optimize import curve_fit

# Above library methods need 2D arrays
x = np.concatenate((np.linspace(0, 0.5, 5),np.linspace(1, 1.5, 5)))[np.newaxis]
y = np.concatenate((np.zeros(5),np.ones(5)))[np.newaxis].T

# Linear
lin_reg = LinearRegression()
lin_reg.fit(x, y)
y_pred_lin = lin_reg.predict(x)

# Quadratic
degree = 5
poly = PolynomialFeatures(degree=degree)
x_poly = poly.fit_transform(x)
lin_reg2 = LinearRegression()
lin_reg2.fit(x_poly, y)
y_pred_pr = lin_reg2.predict(poly.fit_transform(x))

# Logistic function with scale 'a' and shift 'b' parameters
def fsigmoid(_x, a, b):
    return 1.0 / (1.0+np.exp(-a*(_x-b)))

# Curve fit needs 1D array
popt, pcov = curve_fit(fsigmoid, x.ravel(), y.ravel())
y_pred_sigmoid = fsigmoid(x, *popt)

# Plot
plt.figure(figsize=(12, 3), dpi=72)

plt.subplot(1, 3, 1)
plt.scatter(x, y, c='steelblue', edgecolor='white', s=50)
plt.plot(x, y_pred_lin, c='red')
plt.title('Linear regression')

plt.subplot(1, 3, 2)
plt.scatter(x, y, c='steelblue', edgecolor='white', s=50)
plt.plot(x, y_pred_pr, c='red')
plt.title(f'Quadratic regression (deg={degree})')

plt.subplot(1, 3, 3)
plt.scatter(x, y, c='steelblue', edgecolor='white', s=50)
```

```
plt.plot(x, y_pred_sigmoid, c='red')
plt.title('Logistic regression')

plt.show()
```



Note that `PolynomialFeatures(degree=3)` in the above cell creates a new feature of 4 columns, since degree=3, such that $x_{poly} = 1 + x + x^2 + x^3$ and a line is fitted to these new (x_{poly}, y) features. The net effect is fitting a quadratic curve to the input (x, y).

Logistic Regression as a Classifier

Logistic regression is a popular machine learning algorithm used for supervised learning where the target variable is binary, such as the cancer recurrence {'yes', 'no'}. The method involves a particular non-linear transform on the input data X and then a linear regression is performed on the transformed data. As a result, a sigmoidal/logistic curve is fit to the data. The output is an approximation of the probability of the output given the input y.

Learning

A.Transform initial input probabilities into log odds (**logit**)
B.Perform a standard linear regression on the logit values

Note that a linear regression on the plain probabilities would not extrapolate well. Thus, simply perform a linear regression on the transformed probabilities, i.e. on the logarithm of odds, logit, i.e. $ln(\texttt{Prob})$

Generalization:
A.Find the value for the new input on the logit line
B.Transform that logit value back into a probability

In statistics, the logit function is defined as,

$$\text{logit}(p) = \log \frac{p}{1-p} = -\log(\frac{1}{p} - 1)$$

Note on Optimization

As a greedy optimization the library function `LogisticRegression` uses a `solver`. The following table summarizes to help the selection of the `solver`.

Solver	Method	L1/L2	Very Large Dataset	Notes
`newton-cg`	Newton's method	L2	-	Computationally expensive
`lbfgs`	Limited-memory Broyden–Fletcher–Goldfarb–Shanno Algorithm	L2	-	Best for small datasets
`liblinear`	Solves a coordinated descent algorithm	L1	-	It may get stuck at a non-optimal point
`sag`	Stochastic Average Gradient	L2	yes	Θ(n) memory cost
`saga`	Stochastic Average Gradient variant	L1/L2	yes	Θ(n) memory cost

Example

The following is an example which uses the `breast_cancer_preprocessed.csv` which is similar to the module03 dataset without injected erroneous data rows.

In [5]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

# Load data
df = pd.read_csv('../../../EP_datasets/breast_cancer_preprocessed.csv')

# Sanity check
print(f'#rows= {len(df)} #columns= {len(df.columns)}')
df.head()
```

```
#rows= 286 #columns= 10
```

Out[5]:

	age	menopause	tumor-size	inv-nodes	node-caps	deg-malig	breast	breast-quad	irradiat	recurrence
0	44	premeno	18	0	yes	3	right	left_up	no	recurrence-events
1	58	ge40	18	1	no	1	right	central	no	no-recurrence-events
2	51	ge40	39	0	no	2	left	left_low	no	recurrence-events
3	49	premeno	35	1	yes	3	right	left_low	yes	no-recurrence-events
4	41	premeno	32	5	yes	2	left	right_up	no	recurrence-events

Let's check for missing values such as `?` and `NaN`. Then convert the nominal variables which are of dtype `object` to numerical by applying one-hot encoding. Then, check the number of rows are removed compared to the original dataset size.

In [6]:
```
# Check if we have any '?' in df values
print(df.columns[df.isin(['?']).any()])

# Check if we have any NaN in df values
print(df.columns[df.isnull().any()])

# Drop the rows with '?'
df = df[~df['node-caps'].isin(['?']) & ~df['breast-quad'].isin(['?'])]

# See how many rows we lost
print(f'#rows= {len(df)} #columns= {len(df.columns)}')
```

```
Index(['node-caps', 'breast-quad'], dtype='object')
Index([], dtype='object')
#rows= 277 #columns= 10
```

In [7]:
```
def encode_onehot (_df, _f_target):
    ___df = _df.copy()
    # Convert all features of type object to one-hot encoded with pandas dummie
    for f in list(_df.columns.values):
        if _df[f].dtype == object:
            if f == _f_target:
                # recurrence is the target variable and will be treated differe
                continue

            __df = pd.get_dummies(_df[f], prefix='',
                                  prefix_sep='').groupby(level=0, axis=1).max()
            _df = pd.concat([_df, __df], axis=1)
            _df = _df.drop([f], axis=1)

    return _df

# Convert the variable recurrence to numerical
df = encode_onehot(df, 'recurrence')

# Sanity check
df.head()
```

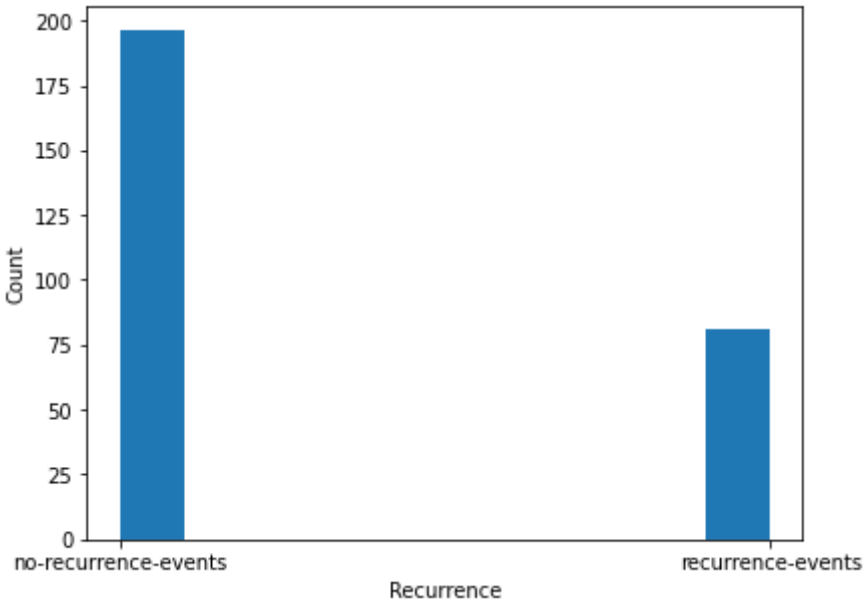
Out[7]:

	age	tumor-size	inv-nodes	deg-malig	recurrence	menopause - ge40	menopause - lt40	menopause - premeno	node-caps - no	node-caps - yes
0	44	18	0	3	recurrence-events	0	0	1	0	1
1	58	18	1	1	no-recurrence-events	1	0	0	1	0
2	51	39	0	2	recurrence-events	1	0	0	1	0
3	49	35	1	3	no-recurrence-events	0	0	1	0	1
4	41	32	5	2	recurrence-events	0	0	1	0	1

Let's use the `LogisticRegression` classifier from sklearn library and rank the important variables by observing the feature weight values. First check the class balance.

In [8]:

```
# Generally LabelEncoder is used supervised learning targets
from sklearn.preprocessing import LabelEncoder

# Set X and y
dfX = df.loc[:, df.columns != 'recurrence']

X = dfX.values

# Original labels from the datafile
y_org = df.loc[:, df.columns == 'recurrence'].values.ravel()

# Convert them to 0, 1
le = LabelEncoder()
y = le.fit_transform(y_org)

# Plot
plt.hist(y)
plt.xticks(np.unique(y), le.classes_)
plt.xlabel('Recurrence')
plt.ylabel('Count')
plt.show()
```



The class labels for this problem is a somewhat unbalanced, since around 30% of the data has 'recurrence', i.e. the cancer came back after the treatment. Now let's check the 10-fold CV with a pipeline of `MinMaxScaler` and `LogisticRegression`. It seems the model performance is fine.

In [9]:
```
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline

# Solve it with Stochastic Average Gradient
clf_lr = LogisticRegression(random_state=0, solver='sag', max_iter=10000)
clf_pipeline = Pipeline([('scaler', MinMaxScaler()), ('clf', clf_lr)])
scores = cross_val_score(estimator=clf_lr, X=X, y=y, cv=10, scoring='accuracy')

print(f'accuracy mean,stdev= {scores.mean():.2f},{scores.std():.2f}')
```

```
accuracy mean,stdev= 0.72,0.05
```

Let's rank the variables with respect to regression coefficients. Compared to the earlier module notebook the ranking is similar, as the first 3 top features were 'deg-malig', 'inv-nodes', and 'node-caps'. Remember that we have two features for 'node-caps' which are one-hot encoded so their weights will be lower as their predicting power are divided into two features.

In [10]:
```
np.warnings.filterwarnings('ignore')  # To ignore type conversion warning

# Uses Stochastic Average Gradient solver
clf = LogisticRegression(random_state=0, solver='sag', max_iter=10000)
clf.fit(MinMaxScaler().fit_transform(X), y)
coef = clf.coef_[0]

# Print the ranking
print('Order of Regression Weights:')
print('Weight   Feature')
```

```
for i in reversed(np.argsort(np.abs(coef))):
    print(f'{np.abs(coef[i]):.3f}    {dfX.columns[i]}')
```

```
Order of Regression Weights:
Weight   Feature
1.488    deg-malig
0.840    tumor-size
0.811    inv-nodes
0.550    menopause - lt40
0.457    age
0.388    breast-quad - right_up
0.367    menopause - premeno
0.312    node-caps - no
0.311    node-caps - yes
0.278    irradiat - no
0.276    irradiat - yes
0.189    breast-quad - right_low
0.181    menopause - ge40
0.139    breast-quad - central
0.131    breast-quad - left_up
0.101    breast - right
0.100    breast - left
0.069    breast-quad - left_low
```

Now let's check what would be predicted with our model for a new patient with the following two patient data:

In [11]:
```
# Create an example data point to test
test_df = pd.DataFrame(data={
    'age': [60, 70],
    'menopause': ['ge40', 'ge40'],
    'tumor-size': [20, 20],
    'inv-nodes': [0, 0],
    'node-caps': ['no', 'no'],
    'deg-malig': [4, 15],
    'breast': ['left', 'right'],
    'breast-quad': ['central', 'central'],
    'irradiat': ['no', 'yes']})
test_df_numerical = encode_onehot(test_df, '')  # Our current df does not conta
test_df_numerical.head()
```

Out[11]:

	age	tumor-size	inv-nodes	deg-malig	menopause - ge40	node-caps - no	breast - left	breast - right	breast-quad - central	irradiat - no	irradiat - yes
0	60	20	0	4	1	1	1	0	1	1	0
1	70	20	0	15	1	1	0	1	1	0	1

We have to match the `test_df` data point to the pre-processed data points which are all numerical (one hot encoded).

In [12]:
```
# Compare the test data point to original dataframe X
dfX.head()
```

Out[12]:

	age	tumor-size	inv-nodes	deg-malig	menopause - ge40	menopause - lt40	menopause - premeno	node-caps - no	node-caps - yes	breast - left	bre - ri
0	44	18	0	3	0	0	1	0	1	0	
1	58	18	1	1	1	0	0	1	0	0	
2	51	39	0	2	1	0	0	1	0	1	
3	49	35	1	3	0	0	1	0	1	0	
4	41	32	5	2	0	0	1	0	1	1	

In [13]:
```
# Add the missing one-hot encoded features
test_df_numerical['menopause - lt40'] = pd.Series([0,0])
test_df_numerical['menopause - premeno'] = pd.Series([0,0])
test_df_numerical['node-caps - yes'] = pd.Series([0,0])
test_df_numerical['breast-quad - left_low'] = pd.Series([0,0])
test_df_numerical['breast-quad - left_up'] = pd.Series([0,0])
test_df_numerical['breast-quad - right_low'] = pd.Series([0,0])
test_df_numerical['breast-quad - right_up'] = pd.Series([0,0])

# Reorder the columns to match the training dataset - the trained model
test_df_numerical = test_df_numerical[dfX.columns]  # dfX is the original trai
test_df_numerical.head()
```

Out[13]:

	age	tumor-size	inv-nodes	deg-malig	menopause - ge40	menopause - lt40	menopause - premeno	node-caps - no	node-caps - yes	breast - left	bre - ri
0	60	20	0	4	1	0	0	1	0	1	
1	70	20	0	15	1	0	0	1	0	0	

In [14]:
```
# Make the predictions
print(le.inverse_transform(clf.predict(test_df_numerical.values)))
```

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

As can be seen in the prediction the first patient outcome is predicted as `no-recurrence` but the second with `recurrence`.

References

1. Rousselet, Guillaume A., Cyril R. Pernet, and Rand R. Wilcox. "Beyond differences in means: robust graphical methods to compare two groups in neuroscience." European Journal of Neuroscience 46.2 (2017): 1738-1748.

Exercises

Exercise 1. Measure the mean and the standard deviation of the data vectors of Anscombe's Quartet, cell `2`. How about the third-moment? State your findings?

Exercise 2. Remove the feature `deg-malig` from the data and re-run the Logistic Regression classifier (Suggestion: Start working on it after cell `5`). What is the new value of the performance metric?

In [15]:
```
%%html
<style>
    table {margin-left: 0 !important;}
</style>
<!-- Display markdown tables left oriented in this notebook. -->
```
