Machine Learning with Python

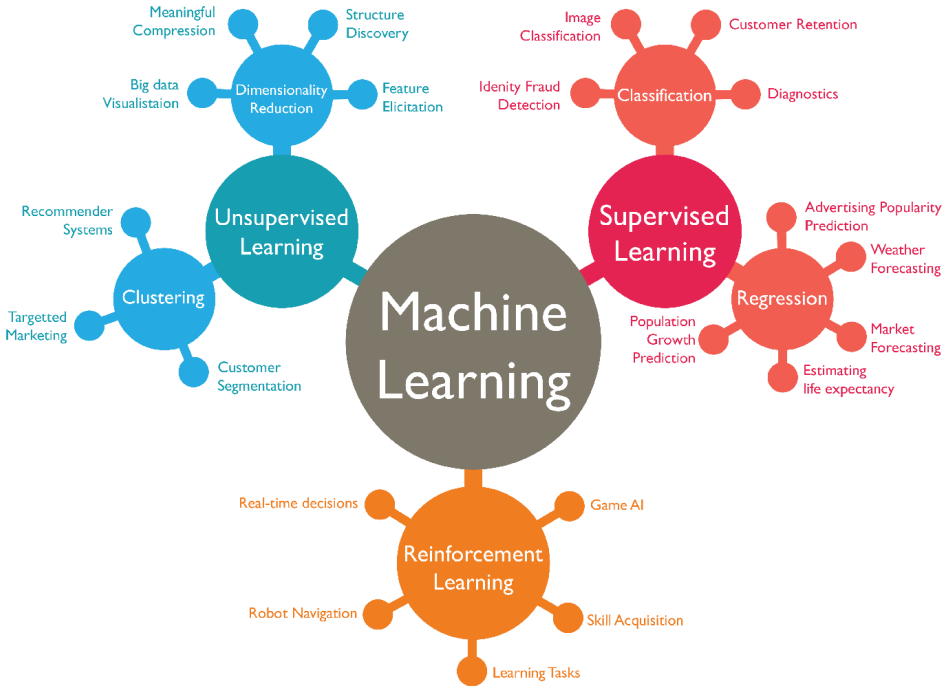
Machine Learning Workshop @ Universal COE

Instructor: Santosh Chapaneri

Machine Learning Concepts

In [1]:
```
from IPython.display import Image
Image('Images/MLTechniques.png', width=600)
```

Out[1]:

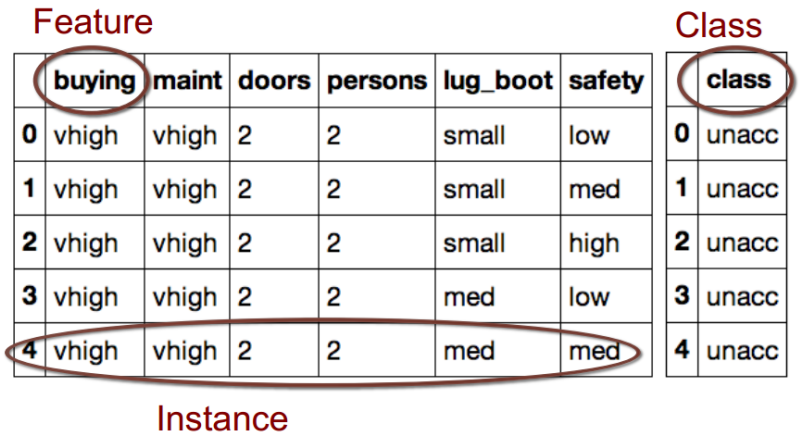


Learning from Data

- In machine learning, most datasets can be represented as tables containing numerical values.
- Every row is called an **instance**, a **sample**, or a **data point**.
- Every column is called a **feature** or a **variable**.

In [2]:
```
from IPython.display import Image
Image('Images/ML_Term1.png', width=500)
```

Out[2]:



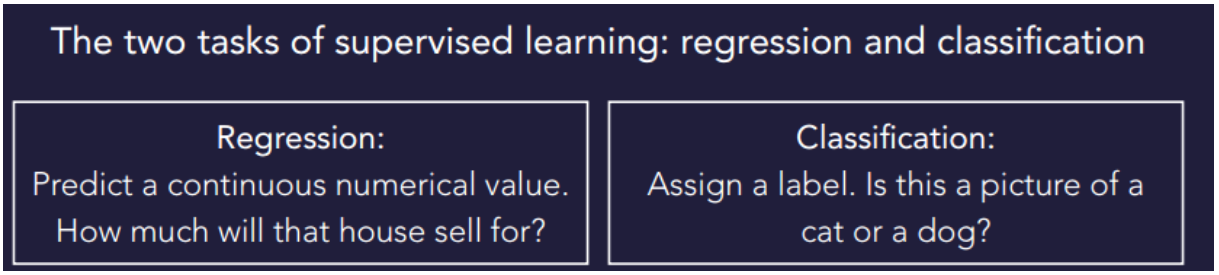
- N = the number of rows (or number of data samples)
- D = the number of columns (or number of features), also called **dimensionality** of data.
- A vector x contains D numbers $(x_1, \ldots, x_D)$, also called **features**.

- **Supervised learning** (http://en.wikipedia.org/wiki/Supervised_learning) is when we have a label y associated to every data point x. The goal is to **learn the mapping** from x to y from our data.
- **Unsupervised learning** (http://en.wikipedia.org/wiki/Unsupervised_learning) is when we don't have any labels. What we want to do is **discover some hidden structure** in the data.

Supervised learning

In [3]:
```
Image('Images/Class_Reg.png')
```

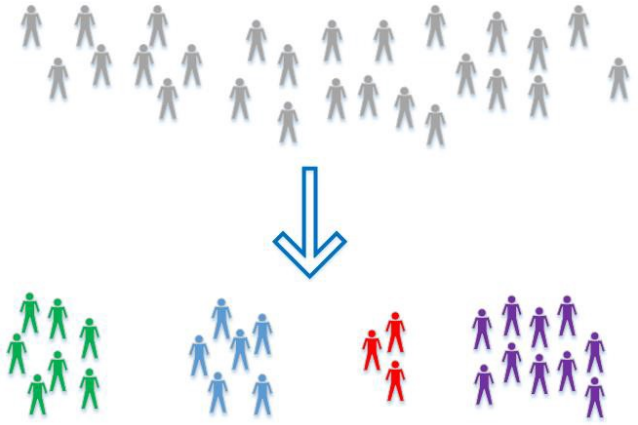
Out[3]:



Unsupervised learning

In [4]:
```
from IPython.display import Image
Image('Images/Unsupervised.png', width=400)
```

Out[4]:



Important terms related to unsupervised learning:

- **Clustering**: Grouping similar points together within clusters.
- **Dimension reduction**: Getting a simple representation of high-dimensional data points by projecting them onto a lower-dimensional space.
- **Manifold learning** (or nonlinear dimension reduction): Finding a low-dimensional manifold containing the data points.
- **Density estimation**: Estimating a probability density that can explain the distribution of the data points.

Feature Selection and Feature Extraction

- When our data contains many features, it is sometimes necessary to choose a subset of them. The features we want to keep are those that are most relevant to our question. This is the problem of **feature selection**.
- Additionally, we may want to extract new features by applying complex transformations on our original dataset. This is **feature extraction**.
- For example, in computer vision, training a classifier directly on pixels is not the most efficient method in general. We may want to extract the relevant points of interest or make appropriate mathematical transformations. These steps depend on our dataset and on the questions we want to answer.

Feature Scaling

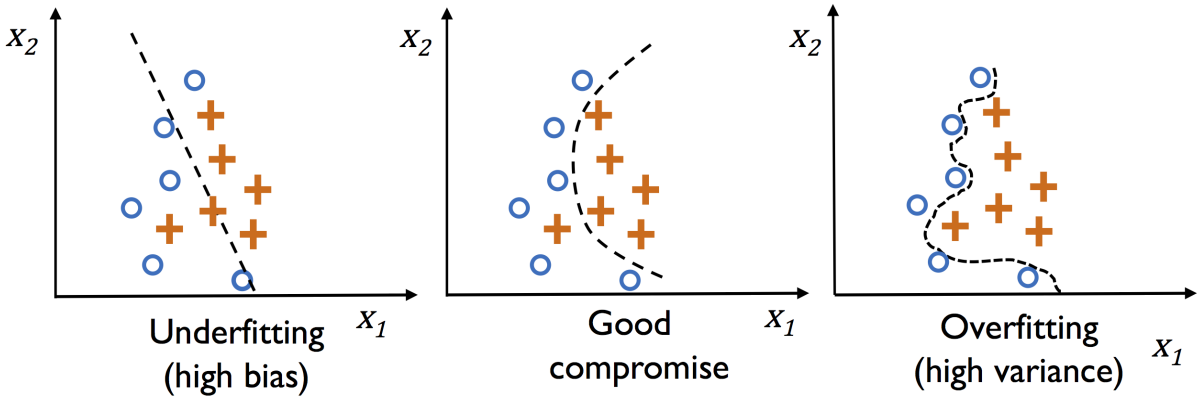
- Preprocess the data before learning models
- Standardization, Min-Max Scaling / Normalization

Overfitting, Underfitting, and the Bias-Variance Tradeoff

- A central notion in machine learning is the trade-off between **overfitting** (http://en.wikipedia.org/wiki/Overfitting) and **underfitting** (http://en.wikipedia.org/wiki/Underfitting).
- A model may be able to represent our data accurately. However, if it is **too accurate**, it may **not generalize well** to unobserved data.
- For example, in face recognition, a **too-accurate model** would be unable to identity someone who styled their hair differently that day. The reason is that our model may **learn irrelevant features** in the training data.
- On the contrary, an insufficiently trained model would not generalize well either. For example, it would be unable to correctly recognize twins.
- A popular solution to **reduce overfitting** consists of adding structure to the model with **regularization** (http://en.wikipedia.org/wiki/Regularization_%28mathematics%29).

In [5]:
```
from IPython.display import Image
Image('Images/Overfitting.png')
```

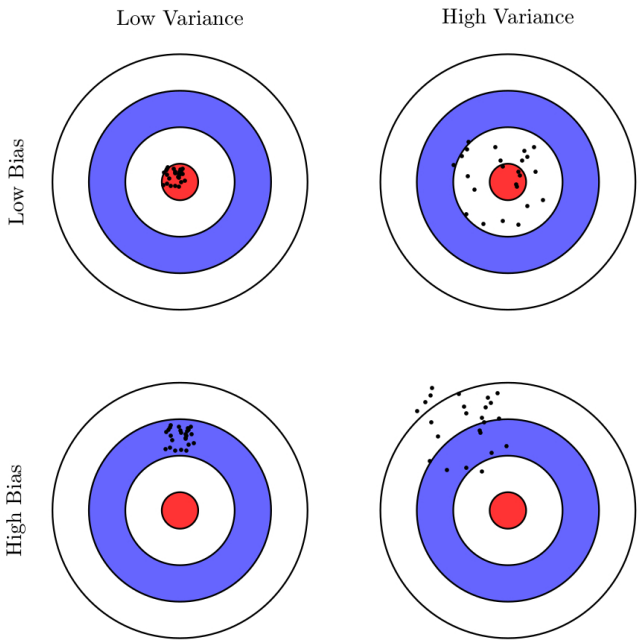
Out[5]:



- The **bias-variance dilemma** (http://en.wikipedia.org/wiki/Bias-variance_dilemma) is closely related.
- The **bias** of a model quantifies how precise a model is across training sets.
- The **variance** quantifies how sensitive the model is to small changes in the training set.
- A robust model is not overly sensitive to small changes. The dilemma involves minimizing both bias and variance; we want a precise and robust model. Simpler models tend to be less accurate but more robust. Complex models tend to be more accurate but less robust.

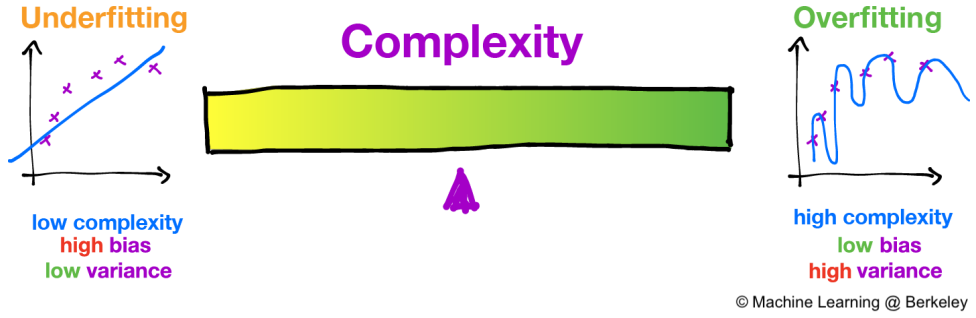
In [6]:
```
from IPython.display import Image
Image('Images/BiasVariance.jpg', width=400)
```

Out[6]:



In [7]:
```
Image('Images/BiasVariance2.png', width=600)
```

Out[7]:



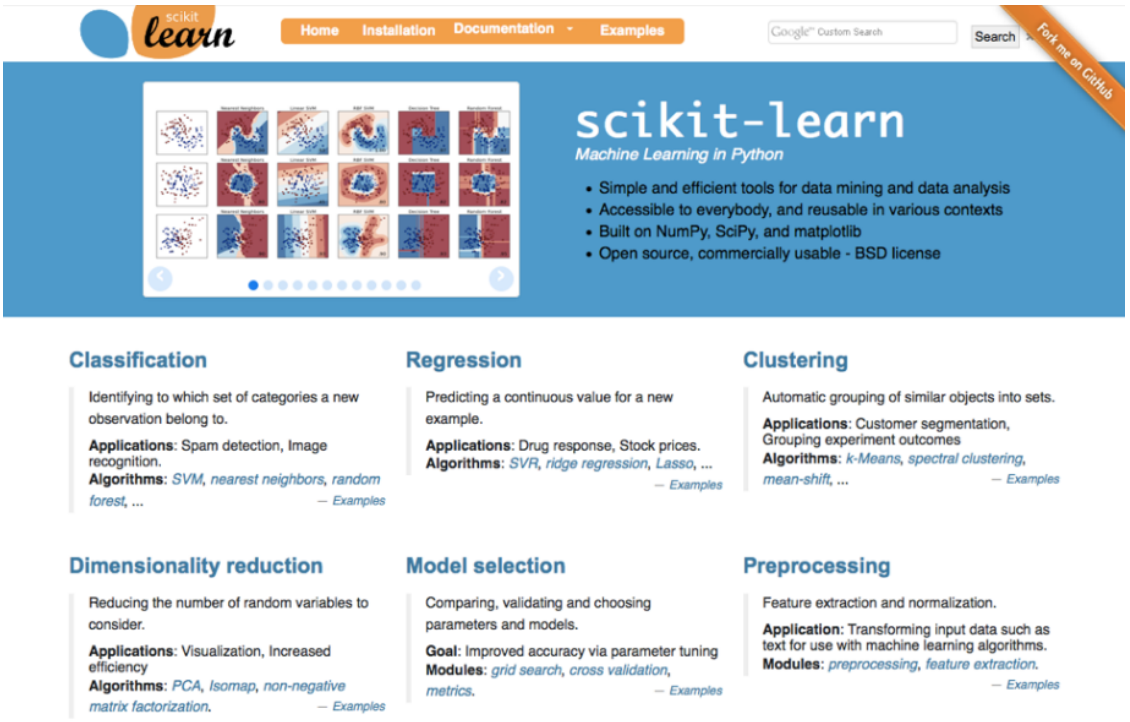
Model selection

- No model performs uniformly better than the others. One model may perform well on one dataset and badly on another. This is the question of **model selection** (http://en.wikipedia.org/wiki/Model_selection).

Getting started with Scikit-learn API

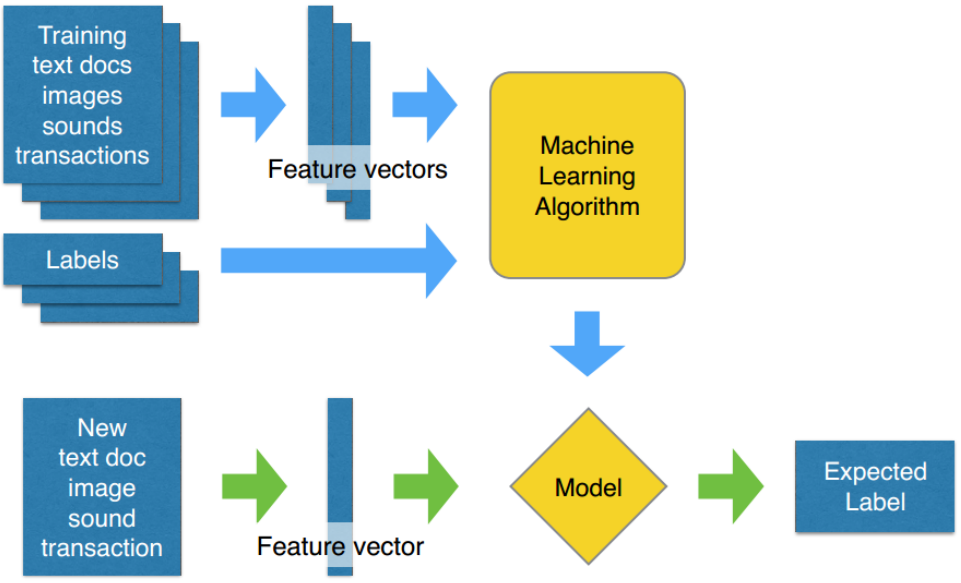
In [8]:
```
Image('Images/SKLearn.png', width=700)
```

Out[8]:



In [9]:
```
Image('Images/ML_Pipeline.png', width=600)
```

Out[9]:



- We now learn the basics of machine learning package **scikit-learn** (http://scikit-learn.org).
- Its clean API makes it really easy to define, train, and test models.
- Plus, scikit-learn is specifically designed for speed and (relatively) big data.

- The data points should be stored in a (N, D) matrix X, where N is the number of observations and D is the number of features. In other words, each row is an observation (data sample).
- The first step in a machine learning task is to define what the matrix X is exactly.
- In a supervised learning setup, we also have a target, an N-long vector y with a scalar value for each observation. This value is either continuous or discrete, depending on whether we have a regression or classifcation problem, respectively.
- In scikit-learn, models are implemented in classes that have the **fit()** and **predict()** methods.
- The **fit()** method accepts the data matrix X as input, and y as well for supervised learning models. This method trains the model on the given data.
- The **predict()** method also takes data points as input (as a (M, D) matrix). It returns the labels or transformed points as predicted by the trained model.

- Remember 3 steps in scikit-learn for machine learning:

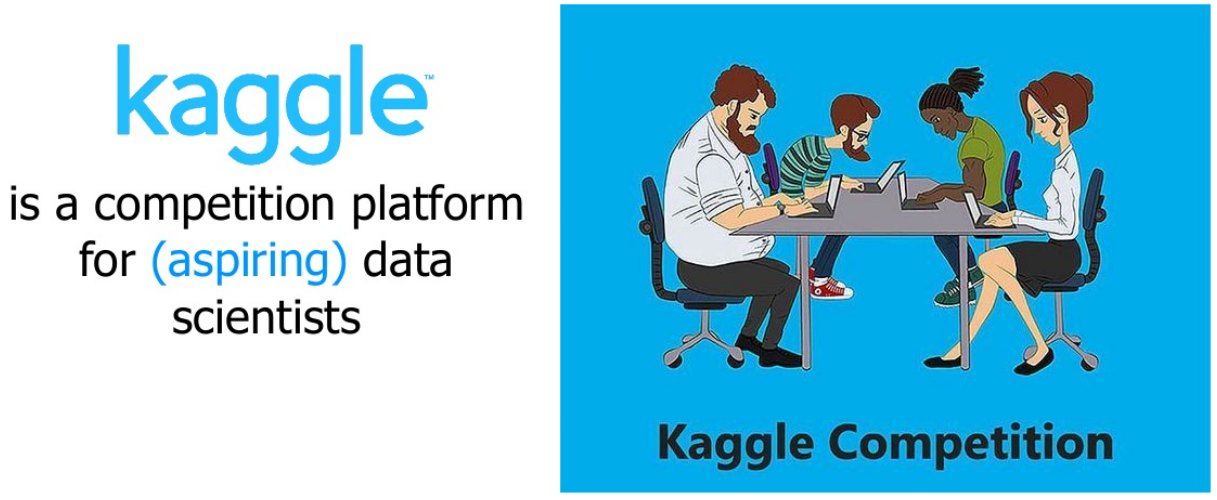
> First, we **create the model** .
>
> Then, we **fit the model to our data**.
>
> Finally, we **predict values** from our trained model.

==========

Logistic Regression: Predicting who will survive on the Titanic

In [10]:
```
Image('Images/Kaggle.png')
```

Out[10]:



Based on a Kaggle competition (http://www.kaggle.com/c/titanic-gettingStarted) where the **goal is to predict survival on the Titanic**, based on real data.

Kaggle (http://www.kaggle.com/competitions) hosts machine learning competitions where anyone can download a dataset, train a model, and test the predictions on the website. The author of the best model wins a price.

Here, we use this example to introduce **logistic regression, a basic classifier**.

Goals:

- **To predict if a passenger survived the sinking of the Titanic or not.**
- **For each PassengerId in the test set, predict a 0 or 1 value for the Survived variable.**

Note:

Logistic regression is not a regression model, it is a classification model. Yet, it is closely related to linear regression.

This model predicts the probability that a binary variable is 1, by applying a sigmoid function (more precisely, a logistic function) to a linear combination of the variables.

$$P(y=1|x) = h_\theta(x) = \frac{1}{1+\exp(-\theta^\top x)} \equiv \sigma(\theta^\top x),$$

$$P(y=0|x) = 1 - P(y=1|x) = 1 - h_\theta(x).$$

$$\theta^\intercal x = \sum_{i=1}^{m} \theta_i x_i = \theta_1 x_1 + \theta_2 x_2 + \ldots \theta_m x_m$$

- Output $Y \sim Ber(p)$, where $p = \sigma(\theta^\intercal x)$
- PMF: $P(Y=y|x) = p^y (1-p)^{1-y}$
- Likelihood:

$$L(\theta) = \prod_{i=1}^{n} P(Y=y^{(i)}|x^{(i)}) = \prod_{i=1}^{n} \sigma(\theta^\intercal x^{(i)})^{y^{(i)}} (1-\sigma(\theta^\intercal x^{(i)}))^{1-y^{(i)}}$$

- Log Likelihood:

$$LL(\theta) = \sum_{i=1}^{n} y_i \log[\sigma(\theta^\intercal x^{(i)})] + (1-y^{(i)}) \log[(1-\sigma(\theta^\intercal x^{(i)}))]$$

- Gradient of LL:

$$\frac{\partial LL(\theta)}{\partial \theta_j} = \frac{\partial LL(\theta)}{\partial p} \frac{\partial p}{\partial \theta_j}, \quad \text{where } p = \sigma(\theta^\intercal x)$$

$$\frac{\partial LL(\theta)}{\partial \theta_j} = \frac{\partial LL(\theta)}{\partial p} \frac{\partial p}{\partial z} \frac{\partial z}{\partial \theta_j}, \quad \text{where } z = \theta^\intercal x$$

$$LL(\theta) = y \log(p) + (1-y) \log(1-p)$$

$$\frac{\partial LL(\theta)}{\partial p} = \frac{y}{p} - \frac{1-y}{1-p}$$

$$\frac{\partial p}{\partial z} = \sigma(z)[1-\sigma(z)]$$

$$\frac{\partial z}{\partial \theta_j} = x_j$$

$$\frac{\partial LL(\theta)}{\partial \theta_j} = \frac{y}{p} - \frac{1-y}{1-p} \cdot \sigma(z)[1-\sigma(z)] \cdot x_j$$

$$\frac{\partial LL(\theta)}{\partial \theta_j} = \frac{y}{p} - \frac{1-y}{1-p} \cdot p[1-p] \cdot x_j$$

$$\frac{\partial LL(\theta)}{\partial \theta_j} = (y - \sigma(\theta^\intercal x)) \cdot x_j$$

$$\frac{\partial LL(\theta)}{\partial \theta_j} = \sum_{i=1}^{m} (y^{(i)} - \sigma(\theta^\intercal x^{(i)})) \cdot x_j^{(i)}$$

- Update:

$$\theta_j^{new} = \theta_j^{old} + \eta \frac{\partial LL(\theta^{old})}{\partial \theta_j^{old}}$$

In [11]:
```
import numpy as np
import pandas as pd
import sklearn
import sklearn.linear_model as lm
import sklearn.model_selection as cv
import matplotlib.pyplot as plt
%matplotlib inline
```

In [12]:
```
# Load the dataset with Pandas
train = pd.read_csv('Data/titanic_train.csv')
train.head()
```

Out[12]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

In [13]:
```
# Observe columns Pclass, Sex, Age, Survived
train[train.columns[[2,4,5,1]]].head()
```

Out[13]:

	Pclass	Sex	Age	Survived
0	3	male	22.0	0
1	1	female	38.0	1
2	3	female	26.0	1
3	1	female	35.0	1
4	3	male	35.0	0

- We will keep only a few fields for this example.
- We also convert the `sex` field to a binary variable, so that it can be handled correctly by NumPy and scikit-learn.
- Finally, we remove the rows containing `NaN` values.

In [14]:
```
data = train[['Sex', 'Age', 'Pclass', 'Survived']].copy()
data['Sex'] = data['Sex'] == 'female'
data = data.dropna()
data.head()
```

Out[14]:

	Sex	Age	Pclass	Survived
0	False	22.0	3	0
1	True	38.0	1	1
2	True	26.0	3	1
3	True	35.0	1	1
4	False	35.0	3	0

- Now, we convert this `DataFrame` to a NumPy array, so that we can pass it to scikit-learn.

In [15]:
```
data_np = data.astype(np.int32).values
X = data_np[:,:-1] # Features
y = data_np[:,-1] # Target (survived or not)
print(X[:5,:])
print(y[:5])
```

```
[[ 0 22  3]
 [ 1 38  1]
 [ 1 26  3]
 [ 1 35  1]
 [ 0 35  3]]
[0 1 1 1 0]
```

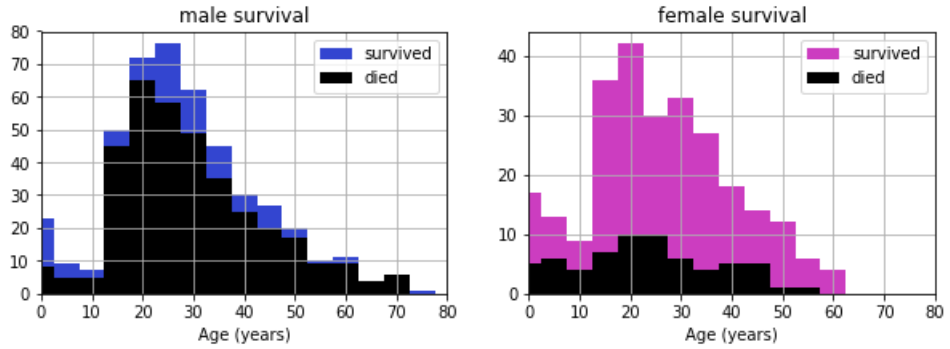
- Let's have a look at the **survival of male and female passengers, as a function of their age**:

In [16]:
```
# Need few boolean vectors
female = (X[:,0] == 1)
survived = (y == 1)

# This vector contains the age of the passengers.
age = X[:,1]

# Compute few histograms
mybins = np.arange(0, 81, 5)
S = {'male': np.histogram(age[survived & ~female],
                          bins=mybins)[0],
     'female': np.histogram(age[survived & female],
                            bins=mybins)[0]}
D = {'male': np.histogram(age[~survived & ~female],
                          bins=mybins)[0],
     'female': np.histogram(age[~survived & female],
                            bins=mybins)[0]}
```

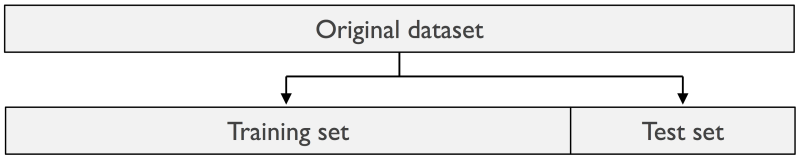
In [17]:
```
# Now plot the data
bins = mybins[:-1]
plt.figure(figsize=(10,3));
for i, sex, color in zip((0, 1),
                         ('male', 'female'),
                         ('#3345d0', '#cc3dc0')):
    plt.subplot(121 + i);
    plt.bar(bins, S[sex], bottom=D[sex], color=color,
            width=5, label='survived');
    plt.bar(bins, D[sex], color='k', width=5, label='died');
    plt.xlim(0, 80);
    plt.grid(None);
    plt.title(sex + " survival");
    plt.xlabel("Age (years)");
    plt.legend();
```



- Let's try to train a `LogisticRegression` classifier. We first need to create a **train and a test dataset**.

In [18]:
```
Image('Images/TrainTestSplit.png', width=500)
```

Out[18]:



In [19]:
```
# Split X and y into train and test datasets
(X_train, X_test, y_train, y_test) = cv.train_test_split(X, y, test_size=.25)
y_test
```

Out[19]:
```
array([0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
       0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0,
       0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1,
       0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1,
       0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0,
       0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0,
       0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1,
       1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0,
       1, 0, 1])
```

In [20]:
```
# Instantiate the classifier / 'Create the model'
logreg = lm.LogisticRegression(solver='lbfgs')
```

- Train the model (**fit**) and get the predicted values (**predict**) on the test set.

In [21]:
```
logreg.fit(X_train, y_train)
y_predicted = logreg.predict(X_test)
y_predicted
```

Out[21]:
```
array([0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0,
       0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0,
       1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1,
       0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0,
       0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0,
       0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0,
       0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1,
       0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0,
       1, 1, 1])
```

In [22]:
```
from sklearn.metrics import accuracy_score
accuracy_score(y_test, y_predicted)
```

Out[22]:
```
0.8324022346368715
```

In [23]:
```
from sklearn.metrics import classification_report
print(classification_report(y_test, y_predicted))
```

```
             precision    recall  f1-score   support

          0       0.88      0.85      0.86       111
          1       0.76      0.81      0.79        68

avg / total       0.83      0.83      0.83       179
```

In [24]:
```
from sklearn.metrics import confusion_matrix
print(confusion_matrix(y_test, y_predicted))
```

```
[[94 17]
 [13 55]]
```

In [25]:
```
# Run Logistic Regression on the IRIS Dataset
import sklearn.datasets as ds
iris = ds.load_iris()
X, y = iris.data, iris.target
(X_train, X_test, y_train, y_test) = cv.train_test_split(X, y, test_size=.25)

logreg = lm.LogisticRegression(solver='lbfgs')
logreg.fit(X_train, y_train)
y_predicted = logreg.predict(X_test)

from sklearn.metrics import accuracy_score
print(accuracy_score(y_test, y_predicted))
print(classification_report(y_test, y_predicted))
print(confusion_matrix(y_test, y_predicted))
```

```
0.9473684210526315
             precision    recall  f1-score   support

          0       1.00      1.00      1.00        15
          1       1.00      0.87      0.93        15
          2       0.80      1.00      0.89         8

avg / total       0.96      0.95      0.95        38

[[15  0  0]
 [ 0 13  2]
 [ 0  0  8]]
```

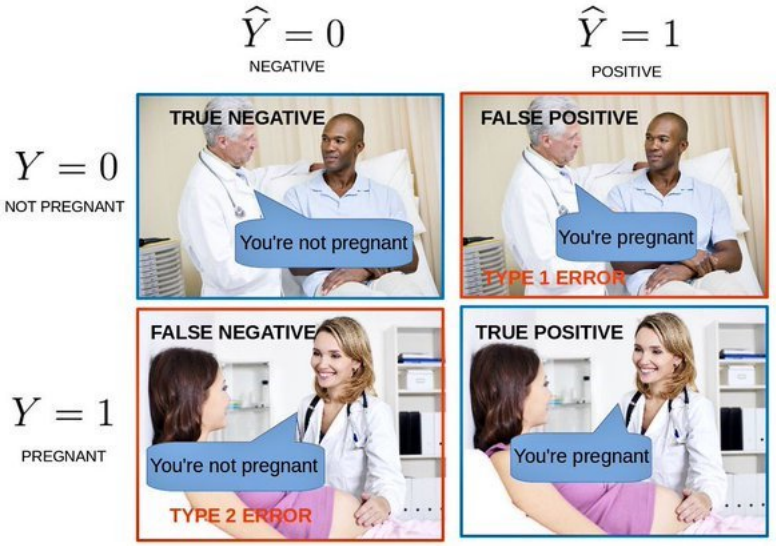
==========

Model Evaluation

Confusion Matrix

In [26]:
```
from IPython.display import Image
Image('Images/ConfMat2.jpg', width = 500)
```

Out[26]:



- **True Positive (TP)**: *correctly predicted* $\hat{Y} = 1$ and $Y = 1$.
- **True Negative (TN)**: *correctly predicted* $\hat{Y} = 0$ and $Y = 0$.
- **False Positive (FP)**: *incorrectly predicted* $\hat{Y} = 1$ but $Y = 0$.
- **False Negative (FN)**: *incorrectly predicted* $\hat{Y} = 0$ but $Y = 1$.

In [27]:
```
Image('Images/ConfMatExample.png', width=300)
```

Out[27]:

n=100	Predicted No:	Predicted Yes:
Actual No 60	50	10
Actual Yes 40	5	35

Accuracy: Overall, how often does the classifier predict right?

$$Accuracy = \frac{TP + TN}{Total} = \frac{35 + 50}{100} = 85\%$$

Misclassification Rate (aka Error Rate): Overall, how often is it wrong?

$$ErrorRate = \frac{FP + FN}{Total} = \frac{5 + 10}{100} = 15\%$$

- This is the equivalent of $1 - Accuracy$.

Accuracy can be misleading

- Suppose we have a model to **identify terrorists** trying to board flights. The model is: *simply label every single person flying from an airport in India as not a terrorist*.
- Given about 100 million average passengers on Indian flights per year and the 10 (confirmed) terrorists who boarded Indian flights from 1971–2017, this model achieves an accuracy of almost 99.999999%!
- Sounds impressive, but is it?
- While this solution has nearly-perfect accuracy, this problem is one in which accuracy is clearly not an adequate metric!
- The terrorist detection task is an **imbalanced classification problem**: we have two classes we need to identify—terrorists and not terrorists—with one category representing the overwhelming majority of the data points.
- Another example: disease detection when the rate of the disease in the public is very low.

Better Performance Indicators

Precision (P): When an event is predicted "yes," how often it is correct?

$$P = \frac{TP}{TP + FP} = \frac{35}{45} = 77\%$$

True Positive Rate (TPR) / Recall (R) / Sensitivity:

$$TPR = Recall = Sensitivity = \frac{TP}{TP + FN} = \frac{35}{40} = 87\%$$

False Positive Rate (FPR):

$$FPR = \frac{FP}{ActualNo} = \frac{10}{60} = 17\%$$

Specificity: When an event was predicted as "no," and it was actually a "no."

$$Specificity = \frac{TN}{ActualNo} = \frac{50}{60} = 83\%$$

F-score (F): harmonic mean of $Precision$ and $Recall$

$$F = \frac{2PR}{P + R} = \frac{2 * 0.87 * 0.77}{0.87 + 0.77} = 82\%$$

Receiver operating characteristic (ROC) curve: Plots TPR versus FPR as a function of the model's **threshold**

Area under the curve (AUC): Metric to calculate the overall performance of a classification model based on area under the ROC curve

Example

- Diagnose 100 patients with a disease present in 50% of the general population.
- Consider a model where we put in information about patients and receive a score between 0 and 1.
- We can modify the threshold for labeling a patient as positive (has the disease) to maximize the classifier performance.
- We will evaluate thresholds from 0.0 to 1.0 in increments of 0.1, and at each step, calculate the $Precision$, $Recall$, F, and location on the ROC curve.

In [28]:
```
Image('Images/ConfMatExampleTable.png', width = 500)
```

Out[28]:

Threshold	TP	FP	TN	FN
0.0	50	50	0	0
0.1	48	47	3	2
0.2	47	40	9	4
0.3	45	31	16	8
0.4	44	23	22	11
0.5	42	16	29	13
0.6	36	12	34	18
0.7	30	11	38	21
0.8	20	4	43	33
0.9	12	3	45	40
1.0	0	0	50	50

In [29]:
```
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
%matplotlib inline

import seaborn as sns
```

In [30]:
```
results = pd.DataFrame({'threshold': [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0],
                        'tp': [50, 48, 47, 45, 44, 42, 36,  30,  20, 12, 0],
                        'fp': [50, 47, 40, 31, 23, 16, 12,  11,   4,  3,  0],
                        'tn': [0,   3,  9, 16, 22, 29, 34,  38,  43, 45, 50],
                        'fn': [0,   2,  4,  8, 11, 13, 18,  21,  33, 40, 50]
                       })
```

In [31]:
```
results
```

Out[31]:

	threshold	tp	fp	tn	fn
0	0.0	50	50	0	0
1	0.1	48	47	3	2
2	0.2	47	40	9	4
3	0.3	45	31	16	8
4	0.4	44	23	22	11
5	0.5	42	16	29	13
6	0.6	36	12	34	18
7	0.7	30	11	38	21
8	0.8	20	4	43	33
9	0.9	12	3	45	40
10	1.0	0	0	50	50

- Let us do **one sample calculation** at a **threshold of 0.5**.
- Confusion matrix: $TN = 29, FP = 16, FN = 13, TP = 42$

$Recall = \frac{TP}{TP+FN} = \frac{42}{42+13} = 76\%$

$Precision = \frac{TP}{TP+FP} = \frac{42}{42+16} = 72.4\%$

$F = \frac{2PR}{P+R} = 74\%$

- Now, calculate the TPR and FPR to find the y and x coordinates for the ROC curve:

$$TPR = Recall = 76\%$$

$$FPR = \frac{FP}{FP+TN} = \frac{16}{16+29} = 36\%$$

- Likewise, **repeat for all thresholds**

In [32]:
```
def calculate_metrics(results):
    roc = pd.DataFrame(index = results['threshold'],
                       columns=['recall', 'precision', 'f', 'tpr', 'fpr'])

    for i in results.iterrows():
        t, tp, fp, tn, fn = i[1]
        assert tp + fp + tn + fn == 100, '#Patients must add up to 100'

        recall = tp / (tp + fn)

        if tp == fp == 0:
            precision = 0
            true_positive_rate = 0

        else:
            precision = tp / (tp + fp)
            true_positive_rate = tp / (tp + fn)

        if precision == recall == 0:
            f = 0
        else:
            f = 2 * (precision * recall) / (precision + recall)

        false_positive_rate = fp / (fp + tn)

        roc.loc[t, 'recall'] = recall
        roc.loc[t, 'precision'] = precision
        roc.loc[t, 'f'] = f
        roc.loc[t, 'tpr'] = true_positive_rate
        roc.loc[t, 'fpr'] = false_positive_rate

    return roc
```

In [33]:
```
roc = calculate_metrics(results)
roc.reset_index()
```

Out[33]:

	threshold	recall	precision	f	tpr	fpr
0	0.0	1	0.5	0.666667	1	1
1	0.1	0.96	0.505263	0.662069	0.96	0.94
2	0.2	0.921569	0.54023	0.681159	0.921569	0.816327
3	0.3	0.849057	0.592105	0.697674	0.849057	0.659574
4	0.4	0.8	0.656716	0.721311	0.8	0.511111
5	0.5	0.763636	0.724138	0.743363	0.763636	0.355556
6	0.6	0.666667	0.75	0.705882	0.666667	0.26087
7	0.7	0.588235	0.731707	0.652174	0.588235	0.22449
8	0.8	0.377358	0.833333	0.519481	0.377358	0.0851064
9	0.9	0.230769	0.8	0.358209	0.230769	0.0625
10	1.0	0	0	0	0	0

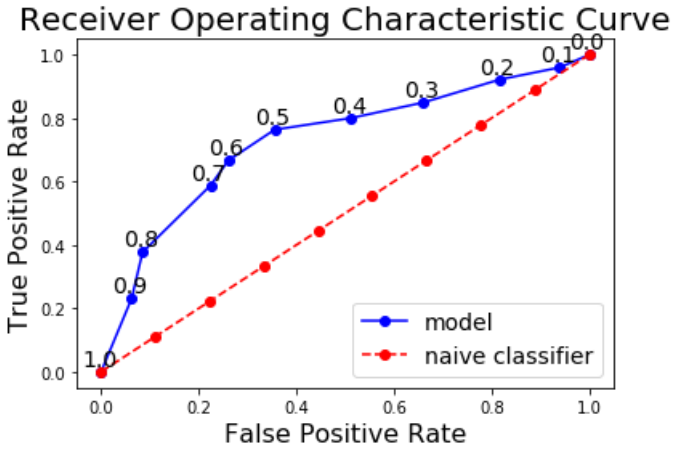
Plotting the ROC Curve

In [34]:
```
plt.style.use('seaborn-dark-palette')
thresholds = [str(t) for t in results['threshold']]
plt.plot(roc['fpr'], roc['tpr'], 'bo-', label = 'model');
plt.plot(list(np.linspace(0, 1, num = 10)),
         list(np.linspace(0, 1, num = 10)),
         'ro--', label = 'naive classifier');

for x, y, s in zip(roc['fpr'], roc['tpr'], thresholds):
    plt.text(x - 0.04,y + 0.02, s, fontdict={'size': 14});

plt.legend(prop={'size':14})
plt.ylabel('True Positive Rate', size = 16)
plt.xlabel('False Positive Rate', size = 16)
plt.title('Receiver Operating Characteristic Curve', size = 20)
```

Out[34]: Text(0.5,1,'Receiver Operating Characteristic Curve')



- At a threshold of 1.0, we classify no patients as having the disease and hence have a $Recall$ and $Precision$ of 0.0.
- As the threshold decreases, the $Recall$ increases because we identify more patients that have the disease. However, as $Recall$ increases, $Precision$ decreases because in addition to increasing the true positives, we increase the false positives.
- At a threshold of 0.0, $Recall$ is perfect—we find all patients with the disease—but $Precision$ is low because we have many false positives.
- We can move along the curve for a given model by changing the threshold and select the threshold that maximizes the F score.
- **Based on the F score, the overall best model occurs at a threshold of 0.5.**
- To shift the entire curve, we would need to build a different model.

==========