# CS 236756 - Technion - Intro to Machine Learning

**Tal Daniel**

## Tutorial 08 - Linear Models

### Agenda

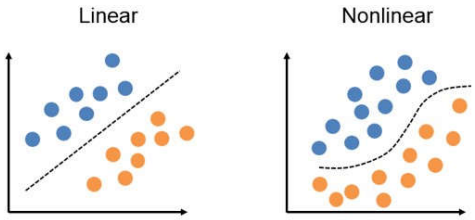
- Definition
- Discriminative Models
  - Perceptron
  - Least Mean Square - LMS (Adaptive Linear Neuron - ADALINE)
  - Logistic Regression
- Generative Models
  - Maximum A Posteriori - MAP
  - Quadratic Discriminant Analysis - QDA
  - Naive Bayes
  - Linear Discriminant Analysis - LDA
- One vs. All Classification

## Definition of Linear Models

Methods that give **linear decision boundries** between classes

$$\{x|w^Tx + w_0 = 0\}$$

where $x$ are the data points. Given binary classification problem with classes 0 and 1, when $w^Tx + w_0 \geq 0$ then points are usually classified as 1 and otherwise as 0.




In statistical classification, including machine learning, two main approaches are called the **generative** approach and the **discriminative** approach. These compute classifiers by different approaches, differing in the degree of statistical modeling.

## Discriminative Models

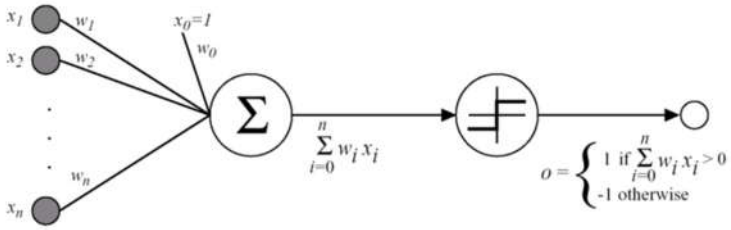
- **Discriminative models** are a class of models used in statistical classification, especially in supervised machine learning. A discriminative classifier tries to model by just depending on the observed data while learning how to do the classification from the given statistics. Comparing with the generative models, discriminative model makes fewer assumptions on the distributions but depends heavily on the quality of the data.
- For example, given a set of labeled pictures of dog and rabbit, discriminative models will be matching a new, unlabeled picture to a most similar labeled picture and then give out the label class, a dog or a rabbit.
- The typical discriminative learning approaches include Logistic Regression(LR), Support Vector Machine(SVM), conditional random fields(CRFs) (specified over an undirected graph), and others.

## The Perceptron

- One of the first and simplest linear model.
- Based on a *linear threshold unit* (LTU): the input and output are numbers (not binary values), and each connection is associated with a weight.
- The LTU computes a weighted sum of its inputs: $z = w_1x_1 + w_2x_2 + \ldots + w_nx_n = w^Tx$, and then it applies a **step function** to that sum and outputs the result:

$$h_w(x) = step(z) = step(w^Tx)$$

- Illustration:




- The most common step function used is the *Heaviside step function* but sometimes the *sign function* is used (as is the illustration).
- **Perceptron Training** draws inspiration from biological neurons: the connection weight between two neurons is increased whenever they have **the same output.** Perceptrons are trained by considering the error made.
  - At each iteration, the Perceptron is fed with one training instance and makes a prediction for it.
  - For every output that produced a wrong prediction, it reinforces the connection weights from the inputs that would have contributed to the correct prediction.
  - Criterion: $E^{perc}(w) = -\sum_{i \in D_{miss}} w^T(x^iy^i)$
- **Perceptron Learning Rule (weight update)**:

$$w_{t+1} = w_t + \eta(y_i - sign(w_t^Tx_i))x_i$$

  - $\eta$ is the learing rate
- The decision boundary learned is linear, the Perceptron is incapable of learning complex patterns.
- **Perceptron Convergence Theorem**: If the training instances are **linearly seperable**, the algorithm would converge to a solution.
  - **There can be multiple solutions (multiple hyperplanes)**
- Perceptrons do not output a class probability, they just make predicitons based on a **hard threshold.**

In [1]:
```
# imports for the tutorial
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib notebook
```

In [2]:
```
# let's load the cancer dataset, shuffle it and  speratre into train and test set
dataset = pd.read_csv('./datasets/cancer_dataset.csv')
# print the number of rows in the data set
number_of_rows = len(dataset)
num_train = int(0.8 * number_of_rows)
# reminder, the data looks like this
dataset.sample(10)
```

Out[2]:

| | id | diagnosis | radius_mean | texture_mean | perimeter_mean | area_mean | smoothness_mean | compactness_mean | conc |
|---|---|---|---|---|---|---|---|---|---|
| **276** | 8911230 | B | 11.33 | 14.16 | 71.79 | 396.6 | 0.09379 | 0.03872 | 0.00 |
| **502** | 91505 | B | 12.54 | 16.32 | 81.25 | 476.3 | 0.11580 | 0.10850 | 0.05 |
| **297** | 892189 | M | 11.76 | 18.14 | 75.00 | 431.1 | 0.09968 | 0.05914 | 0.02 |
| **347** | 89869 | B | 14.76 | 14.74 | 94.87 | 668.7 | 0.08875 | 0.07780 | 0.04 |
| **168** | 8712766 | M | 17.47 | 24.68 | 116.10 | 984.6 | 0.10490 | 0.16030 | 0.21 |
| **476** | 911654 | B | 14.20 | 20.53 | 92.41 | 618.4 | 0.08931 | 0.11080 | 0.05 |
| **310** | 893783 | B | 11.70 | 19.11 | 74.33 | 418.7 | 0.08814 | 0.05253 | 0.01 |
| **225** | 88143502 | B | 14.34 | 13.47 | 92.51 | 641.2 | 0.09906 | 0.07624 | 0.05 |
| **202** | 878796 | M | 23.29 | 26.67 | 158.90 | 1685.0 | 0.11410 | 0.20840 | 0.35 |
| **215** | 8810987 | M | 13.86 | 16.93 | 90.96 | 578.9 | 0.10260 | 0.15170 | 0.09 |

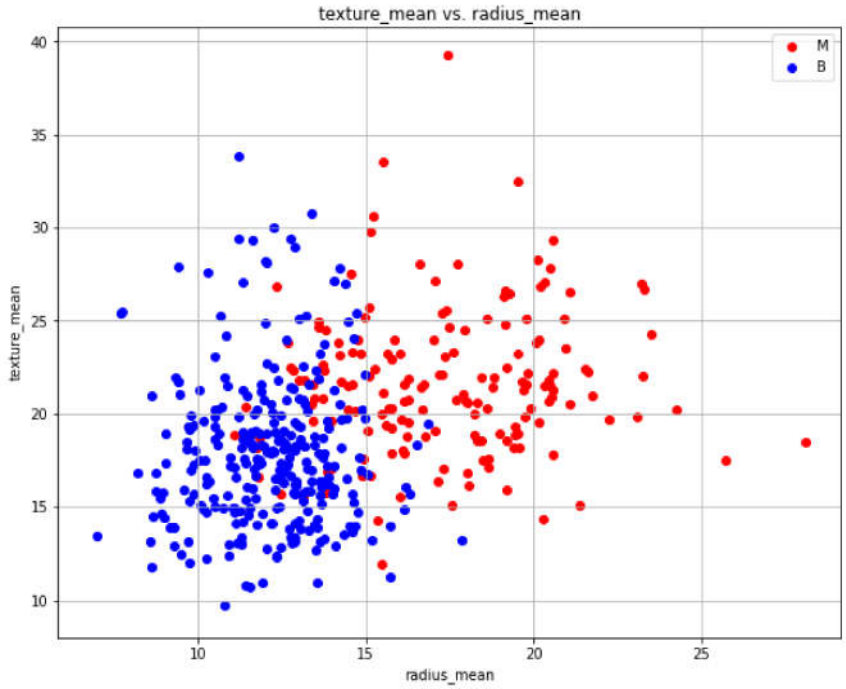
10 rows × 33 columns

In [3]:
```
# we will take the first 2 features as our data (X) and the diagnosis as labels (y)
x = dataset[['radius_mean', 'texture_mean']].values
y = dataset['diagnosis'].values == 'M'  # 1 for Malignat, 0 for Benign
# shuffle
rand_gen = np.random.RandomState(0)
shuffled_indices = rand_gen.permutation(np.arange(len(x)))

x_train = x[shuffled_indices[:num_train]]
y_train = y[shuffled_indices[:num_train]]
x_test = x[shuffled_indices[num_train:]]
y_test = y[shuffled_indices[num_train:]]

print("total training samples: {}, total test samples: {}".format(num_train, number_of_rows - num_train))
```

```
total training samples: 455, total test samples: 114
```

In [4]:
```
# let's see it
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(1,1,1)
ax.scatter(x_train[y_train,0], x_train[y_train, 1], color='r', label="M")
ax.scatter(x_train[~y_train,0], x_train[~y_train, 1], color='b', label="B")
ax.legend()
ax.grid()
ax.set_xlabel("radius_mean")
ax.set_ylabel("texture_mean")
ax.set_title("texture_mean vs. radius_mean")
```

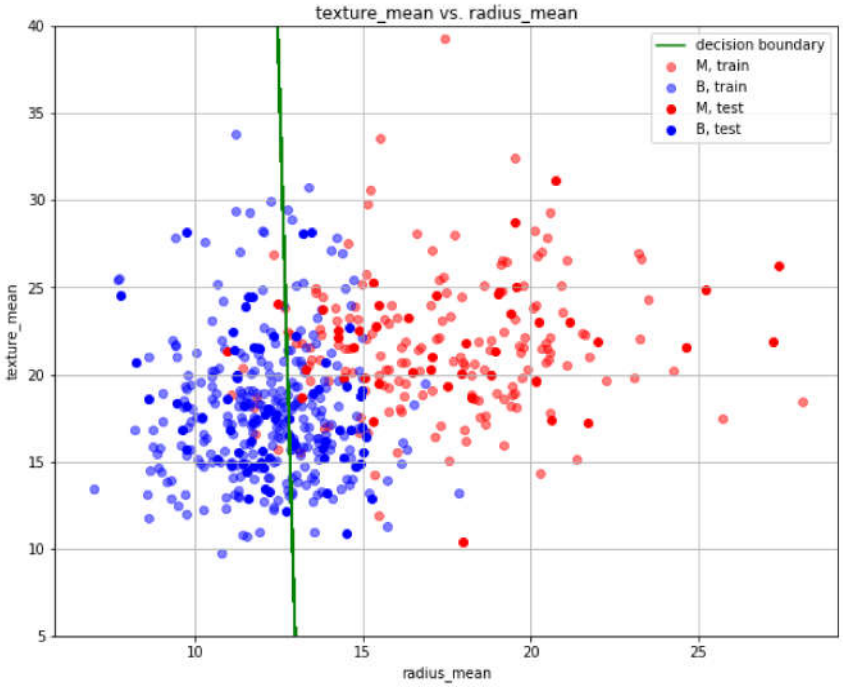
Out[4]: Text(0.5,1,'texture_mean vs. radius_mean')



In [5]:
```
# perceptron using Scikit-Learn
from sklearn.linear_model import Perceptron
per_clf = Perceptron(random_state=42, max_iter=80)
per_clf.fit(x_train, y_train)
y_pred = per_clf.predict(x_test)
accuracy = np.sum(y_pred == y_test) / len(y_test)
print("perceptron accuracy: {:.3f} %".format(accuracy * 100))
w = (per_clf.coef_).reshape(-1,)
b = (per_clf.intercept_).reshape(-1,)
boundary = (-b -w[0] * x_train[:, 0]) / w[1]
# plot
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(1,1,1)
ax.scatter(x_train[y_train,0], x_train[y_train, 1], color='r', label="M, train", alpha=0.5)
ax.scatter(x_train[~y_train,0], x_train[~y_train, 1], color='b', label="B, train", alpha=0.5)
ax.scatter(x_test[y_test,0], x_test[y_test, 1], color='r', label="M, test", alpha=1)
ax.scatter(x_test[~y_test,0], x_test[~y_test, 1], color='b', label="B, test", alpha=1)
ax.plot(x_train[:,0], boundary, label="decision boundary", color='g')
ax.legend()
ax.grid()
ax.set_ylim([5, 40])
ax.set_xlabel("radius_mean")
ax.set_ylabel("texture_mean")
ax.set_title("texture_mean vs. radius_mean")
```

```
perceptron accuracy: 74.561 %
```

Out[5]:
```
Text(0.5,1,'texture_mean vs. radius_mean')
```

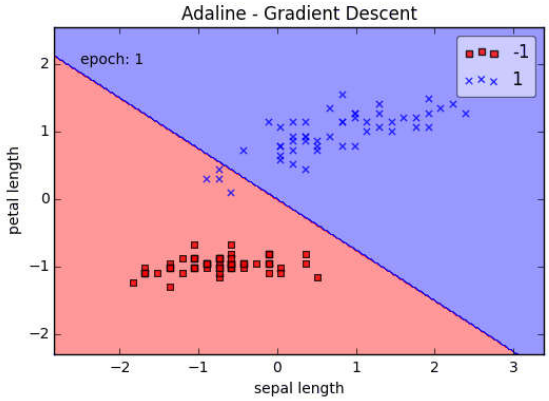


## Least Mean Square - LMS (Adaptive Linear Neuron - ADALINE)

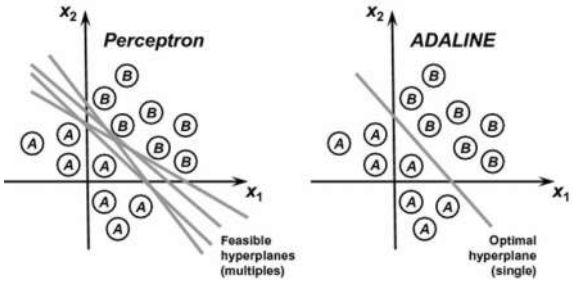
- Adaline is a single layer neural network with multiple nodes where each node accepts multiple inputs and generates one output.
- The difference between Adaline and the standard Perceptron is that in the learning phase, the weights are adjusted according to the weighted sum of the inputs (the net). In the standard Perceptron, the net is passed to the activation function (step() or sign() and the function's output is used for adjusting the weights.
- Criterion: $E(w) = \frac{1}{2} \sum_{i \in D} (y_i - w^T x_i)^2$
- **ADALINE Training**:
  - SGD or Pseudo-inverse
  - **ADALINE Learning Rule (weight update):**

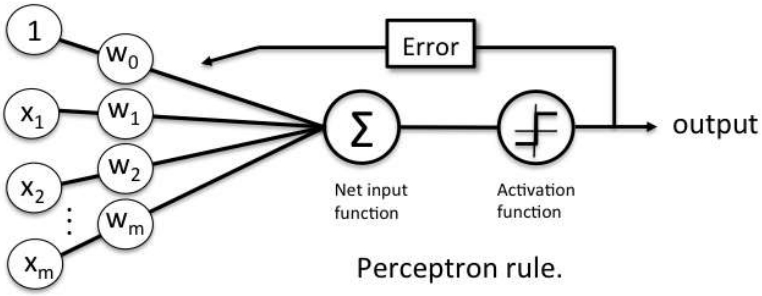
$$w_{t+1} = w_t + \eta (y_i - w_t^T x_i) x_i$$

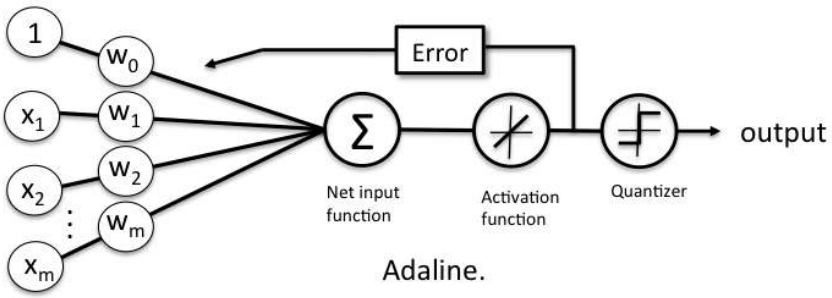
- Training Process:




### LMS vs. Pereptron




- Image and animation by Sebastian Raschka (https://sebastianraschka.com/Articles/2015_singlelayer_neurons.html)
- Python implementation of ADALAINE using SGD can be found here (https://sebastianraschka.com/Articles/2015_singlelayer_neurons.html).

## Recap: Maximum Likelihood Estimation

- Maximum Likelihood Estimation (MLE) is the most common way to estimate parameters of a statistical model by calculating:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} \log p(y|x,\theta)$$

- The Negative Log Likelihood (NLL) under **i.i.d** assumption:

$$NLL(\theta) = -\log p(D|\theta) = -\sum_{i=1}^{n} \log p(y_i|x_i,\theta)$$

## MLE with Bernoulli Assumption

- We assume that:

$$P(y|x,\theta) = Bern(y|\sigma(\theta^T x))$$

  - Bernoulli Distribution (coin flip):

$$P(x) = p^x(1-p)^{1-x}$$

- The *Sigmoid* function (also the Logistic Function):

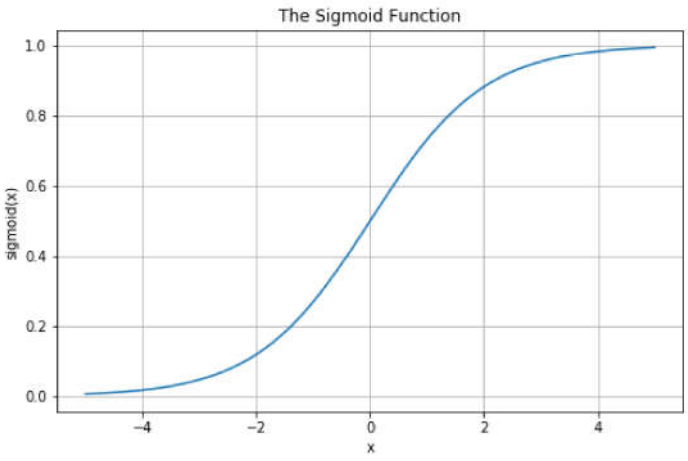
$$\sigma(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{1+e^x}$$

  - The output is in $[0,1]$, which is exaclty what we need to model a probability distribution.
- We will use the following notations:

$$P(y_i|x_i,w) = \begin{cases} \pi_{i1} = \sigma(w^T x) = \frac{1}{1+e^{-x}} & \text{if } y_i = 1 \\ \pi_{i0} = 1 - \sigma(w^T x) = 1 - \frac{1}{1+e^{-x}} & \text{if } y_i = 0 \end{cases}$$

In [6]:
```
# Let's see the sigmoid function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

x = np.linspace(-5, 5, 1000)
sig_x = sigmoid(x)
# plot
fig = plt.figure(figsize=(8,5))
ax = fig.add_subplot(111)
ax.plot(x, sig_x)
ax.grid()
ax.set_title("The Sigmoid Function")
ax.set_xlabel("x")
ax.set_ylabel("sigmoid(x)")
```

Out[6]: Text(0,0.5,'sigmoid(x)')



## Logistic Regression

- Some regression algorithms can be used for classification as well.
- *Logistic Regression* is commonly used to **estimate the probability** that an instance belongs to a particular class.
  - Typically, if the estimated proabibility is greater than 50%, then the model predicts that the instance belongs to that class (called the positive class, labeled "1"), or else it predicts that it does not - a binary classifier.
- **Estimating Probabilities** - Similarly to *Linear Regression*, a Logistic Regression model computes a weighted sum of the input features (plus a bias term), but unlike Linear Regression, it outputs the **logistic** of the weighted sum - $\sigma(w^T x)$, which is a number between 0 and 1.
- **Training and Cost Function**:
  - The objective of training is to set the parameter vector $\theta$ (or $w$) so that the model estimates high probabilities for positive instances ($y = 1$) and low probabilities for negative instances ($y = 0$)
  - Expanding the expression:

$$P(y|x,\theta) = Bern(y|\sigma(\theta^T x)) \rightarrow NLL(\theta) = \frac{1}{m}\sum_{i=1}^{n} \log \sigma(\theta^T x)^{y_i}(1-\sigma(\theta^T x))^{1-y_i} = \frac{1}{m}\sum_{i=1}^{n} \log \pi_{i1}^{y_i}\pi_{i0}^{1-y_i}$$

  - This yields **the Logistic Regression cost function (log loss)**:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y_i \log \pi_{i1} + (1-y_i)\log \pi_{i0}\right] = -\frac{1}{m}\sum_{i=1}^{m}\left[y_i \log \pi_{i1} + (1-y_i)\log(1-\pi_{i1})\right]$$

    - Intuition: $-\log(t)$ grows very large when $t$ approaches 0, so the cost will be large if the model estimates a probability close to 0 for a **positive instance**, and it will laso be very large if the estimated probability is close to 1 for a **negative instance.** On the other hand, $-log(t)$ is close to 0 when $t$ is close to 1, so the cost will be close to 0 if the estimated probability is close to 0 for a **negative instance** or close to 1 for a **positive instance.**
    - This expression is also called the **binary cross-entropy (BCE)** loss.
    - The cost function is **convex.**
  - **Logistic cost function dervatives**:

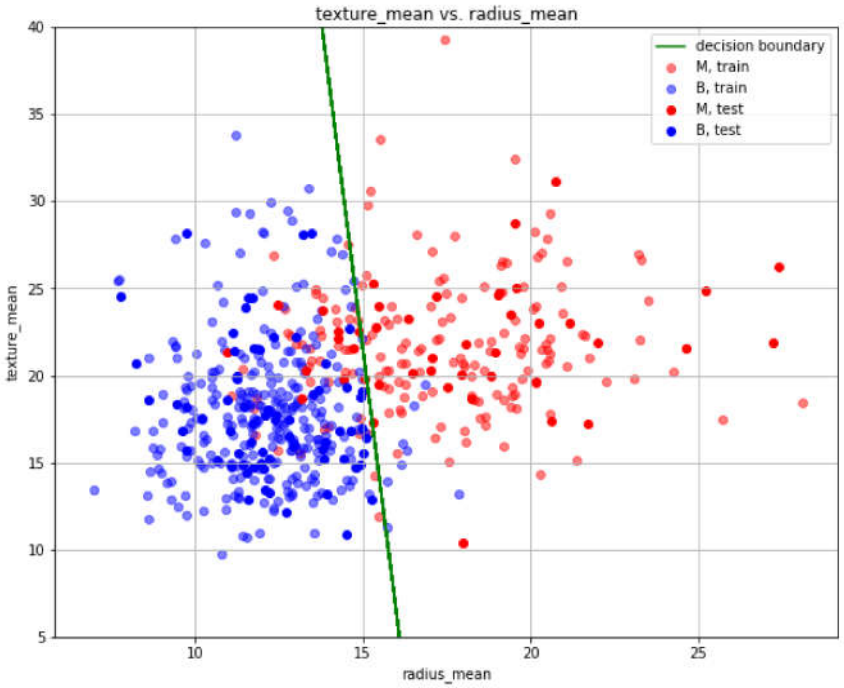
$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(\sigma(\theta^T x^i) - y_i\right)x_j^i$$

    - No closed-form solution.
    - Thanks to the convexity of the cost function, we can use **Gradient Descent** (or SGD, Mini-Batch GD).

In [7]:
```
# logistic regression with scikit-learn
from sklearn.linear_model import LogisticRegression
log_reg = LogisticRegression()
log_reg.fit(x_train, y_train)
y_pred = log_reg.predict(x_test)
accuracy = np.sum(y_pred == y_test) / len(y_test)
print("Logistic Regression accuracy: {:.3f} %".format(accuracy * 100))
w = (log_reg.coef_).reshape(-1,)
b = (log_reg.intercept_).reshape(-1,)
boundary = (-b -w[0] * x_train[:, 0]) / w[1]
# plot
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(1,1,1)
ax.scatter(x_train[y_train,0], x_train[y_train, 1], color='r', label="M, train", alpha=0.5)
ax.scatter(x_train[~y_train,0], x_train[~y_train, 1], color='b', label="B, train", alpha=0.5)
ax.scatter(x_test[y_test,0], x_test[y_test, 1], color='r', label="M, test", alpha=1)
ax.scatter(x_test[~y_test,0], x_test[~y_test, 1], color='b', label="B, test", alpha=1)
ax.plot(x_train[:,0], boundary, label="decision boundary", color='g')
ax.legend()
ax.grid()
ax.set_ylim([5, 40])
ax.set_xlabel("radius_mean")
ax.set_ylabel("texture_mean")
ax.set_title("texture_mean vs. radius_mean")
```

```
Logistic Regression accuracy: 90.351 %
```

Out[7]:
```
Text(0.5,1,'texture_mean vs. radius_mean')
```



## Multi-Class (Multinomial) Logistic Regression - Softmax Regression

- The Logistic Regression model can be generalized to support multiple classes.
- The idea: when given an instance $x$, the Softmax Regression model first computes a score $s_k(x)$ for each class $k$, then estimates a probability of each class by applying the *softmax function* (normalized exponential) to the scores.
- The **Softmax score for class** $k$:

$$s_k(x) = \left(\theta^{(k)}\right)^T \cdot x$$

  - Each class has its own dedicated parameter vector $\theta^{(k)}$, which is usually stored in a row of the parameter matrix $\Theta$.
- **The Softmax Function**:

$$\hat{p}_k = p(y = k|x, \theta) = \sigma(s(x))_k = \frac{e^{s_k(x)}}{\sum_{j=1}^{K} e^{s_j(x)}}$$

  - $K$ is the number of classes.
  - $s(x)$ is a *vector* containing the scores of each class for the instance $x$
  - $\sigma(s(x))_k$ is the estimated probability that the instance $x$ belongs to class $k$ given the scores of each class for that instance.
- **The Softmax Regression classifier prediction**:

$$\hat{y} = \underset{k}{\operatorname{argmax}} \sigma(s(x))_k = \underset{k}{\operatorname{argmax}} s_k(x) = \underset{k}{\operatorname{argmax}} \left((\theta^{(k)})^T x\right)$$

- **Cross-Entropy cost function**:

$$J(\Theta) = -\frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{K} y_k^{(i)} \log(\hat{p}_k^{(i)})$$

  - $y_k^{(i)}$ is equal to 1 if the target class for the $i^{th}$ instance is $k$, otherwise, it is 0.
  - When $K = 2$ it is the BCE from the previous section.
- **Cross-Entropy gradient vector for class** $k$:

$$\nabla_{\theta^{(k)}} J(\Theta) = \frac{1}{m} \sum_{i=1}^{m} (\hat{p}_k^{(i)} - y_k^{(i)}) x^{(i)}$$

  - Use Gradient Descent or its variants to solve
- In Scikit-Learn: `softmax_reg = LogisticRegression(multi_class="multinomial", solver="lbfgs", C=10)`
  - $C$ is the number of classes to use.

# Generative Models

- A generative model is a model of the conditional probability of the observable X, given a target y:$P(X|Y=y)$ or the joint distribution $P(X,Y)$.
- A generative model can be used to "generate" random instances, either of an observation and target.
- The typical generative model approaches contain Naive Bayes, Gaussian Mixture Model, and others.

## Recap: Bayes Rule

$$P(y|x) = \frac{P(x|y)P(y)}{P(x)}$$

Posterior, Likelihood, Prior, Evidence

- Note that $P(x) = \sum_k P(x|y)P(y)$

## Maximum A Posteriori (MAP)

- MAP estimation seeks to maximize the **posterior distribution** (unlike MLE, which seeks to maximize the likelihood) by taking into account the **prior probability**:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} p(x|y,\theta)p(y)$$

  - When the **prior is constant** then MLE = MAP. More precisely, when the prior is **uniform**, then the MAP estimator is the same as MLE. We can say that MLE is a special case of MAP when the prior is uniform!
  - If we use different prior, say, a Gaussian, then our prior is not constant anymore, as depending on the region of the distribution, the probability is high or low, never always the same.
- MAP estimation **minimizes the classification error** (the best estimator we can get).
- However, estimating the prior is not trivial which leads us to make some strong assumptions (that may be wrong).

## Gaussian Assumption - Quadratic Discriminant Analysis - QDA

- We assume the likelihood (class-conditioned probabilities) to be Gaussians and the **prior to be Categorical**, denoting: $P(y=c) = \pi_c$
- The MAP estimator/classifier:

$$P(y=c|X) = \frac{P(X|y=c)P(y=c)}{P(X)} = \frac{P(X|y=c)P(y=c)}{\sum_{c'} P(X|y=c')P(y=c')}$$

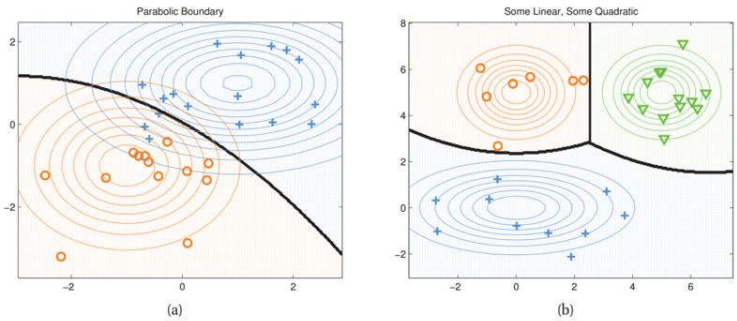
  - We select class $c$ that maximizes the conditional probability.
  - $P(X|y=c,\theta) = (2\pi)^{-\frac{d}{2}}|\Sigma_c|^{-\frac{1}{2}} e^{-\frac{1}{2}(X-\mu_c)^T\Sigma_c^{-1}(X-\mu_c)}$
- QDA for the **Binary** case:

$$\pi_0(2\pi)^{-\frac{d}{2}}|\Sigma_0|^{-\frac{1}{2}} e^{-\frac{1}{2}(X-\mu_0)^T\Sigma_0^{-1}(X-\mu_0)} \lessgtr \pi_1(2\pi)^{-\frac{d}{2}}|\Sigma_1|^{-\frac{1}{2}} e^{-\frac{1}{2}(X-\mu_1)^T\Sigma_1^{-1}(X-\mu_1)}$$

$$e^{-\frac{1}{2}(X-\mu_0)^T\Sigma_0^{-1}(X-\mu_0)} \lessgtr \frac{\pi_1}{\pi_0}\left(\frac{|\Sigma_1|}{|\Sigma_0|}\right)^{-\frac{1}{2}} e^{-\frac{1}{2}(X-\mu_1)^T\Sigma_1^{-1}(X-\mu_1)}$$

$$(X-\mu_0)^T\Sigma_0^{-1}(X-\mu_0) \gtrless \log\left(\frac{\pi_1}{\pi_0}\left(\frac{|\Sigma_1|}{|\Sigma_0|}\right)\right) + (X-\mu_1)^T\Sigma_1^{-1}(X-\mu_1)$$

- The above is **quadratic equation**: decision boundaries are a second order curves.
-




* Machine Learning a probabilistic perspective by Kevin Murphey, figure 4.3

- **QDA Training**:

- Training can be performed by MLE parameter estimation:
  - $\hat{\pi_c} = \frac{n_c}{n}$
  - $\hat{\mu_c} = \frac{1}{n_c} \sum_{i \in c} X_i$
  - $\hat{\Sigma_c} = \frac{1}{n_c} \sum_{i \in c} (X_i - \hat{\mu_c})(X_i - \hat{\mu_c})^T$
- In Scikit-Learn: `from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis`
  - Doc (https://scikit-learn.org/stable/modules/lda_qda.html)

## Gaussian Assumption -Naive Bayes Classifier

- If in the QDA model one assumes that the covariance matrices are diagonal, then the inputs are assumed to be conditionally independent in each class, and the resulting classifier is equivalent to the Gaussian Naive Bayes classifier .
- In Scikit-Learn: `from sklearn.naive_bayes import GaussianNB`
  - Doc (https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html#sklearn.naive_bayes.GaussianNB)

In [8]:
```
# qda
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis as QDA
qda_clf = QDA(store_covariance=True)
qda_clf.fit(x_train, y_train)
y_pred = qda_clf.predict(x_test)
accuracy = np.sum(y_pred == y_test) / len(y_test)
print("QDA accuracy: {:.3f} %".format(accuracy * 100))

# naive bayes
from sklearn.naive_bayes import GaussianNB
nb_clf = GaussianNB()
nb_clf.fit(x_train, y_train)
y_pred = nb_clf.predict(x_test)
accuracy = np.sum(y_pred == y_test) / len(y_test)
print("Naive-Bayes accuracy: {:.3f} %".format(accuracy * 100))
```

```
QDA accuracy: 91.228 %
Naive-Bayes accuracy: 90.351 %
```

## Gaussian Assumption - Linear Discriminant Analysis - LDA

- We assume the likelihood (class-conditioned probabilities) to be Gaussians and the **prior to be Categorical**, denoting: $P(y=c)=\pi_c$
- **LDA** - a special case of QDA where the covarinace matrix is the same, that is, the covarince matrices are shared accross classes: $\Sigma_c=\Sigma$
- The MAP estimator/classifier:

$$P(y=c|X)=\frac{P(X|y=c)P(y=c)}{P(X)}=\frac{P(X|y=c)P(y=c)}{\sum_{c'}P(X|y=c')P(y=c')}$$

  - We select class $c$ that maximizes the conditional probability.
  - $P(X|y=c,\theta)=(2\pi)^{-\frac{d}{2}}|\Sigma|^{-\frac{1}{2}}e^{-\frac{1}{2}(X-\mu_c)^T\Sigma^{-1}(X-\mu_c)}$
  - $(X-\mu_c)^T\Sigma^{-1}(X-\mu_c)=X^T\Sigma^{-1}X-2\mu_c^T\Sigma^{-1}X+\mu_c^T\Sigma^{-1}\mu_c$
  - We denote:
    - $\gamma_c=-\frac{1}{2}\mu_c^T\Sigma^{-1}\mu_c+\log(\pi_c)$
    - $\beta_c=\Sigma^{-1}\mu_c$
  - $\rightarrow P(y=c|X)=\frac{P(X|y=c)P(y=c)}{\sum_{c'}P(X|y=c')P(y=c')}=\frac{e^{\beta_c^TX+\gamma_c}}{\sum_{c'}e^{\beta_{c'}^TX+\gamma_{c'}}}$
    - This is a **softmax function** (you can be asked "show that LDA classification is similar to softmax")
    - In *Logistic Regression* we optimized it directly , whereas here we learn the entire **posterior** (discriminative vs. generative)
- LDA for the **Binary** case:

$$\pi_0(2\pi)^{-\frac{d}{2}}|\Sigma|^{-\frac{1}{2}}e^{-\frac{1}{2}(X-\mu_0)^T\Sigma^{-1}(X-\mu_0)}\lessgtr\pi_1(2\pi)^{-\frac{d}{2}}|\Sigma|^{-\frac{1}{2}}e^{-\frac{1}{2}(X-\mu_1)^T\Sigma^{-1}(X-\mu_1)}$$

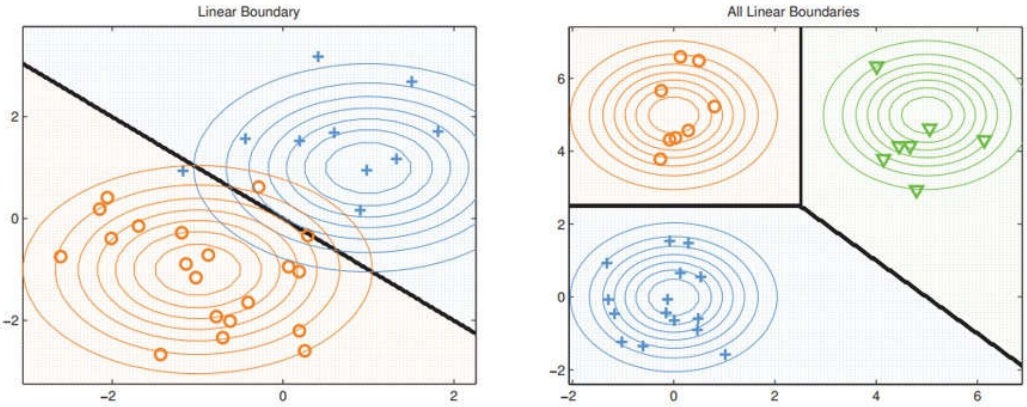
$$e^{-\frac{1}{2}(X-\mu_0)^T\Sigma^{-1}(X-\mu_0)}\lessgtr\frac{\pi_1}{\pi_0}\left(\frac{|\Sigma|}{|\Sigma|}\right)^{-\frac{1}{2}}e^{-\frac{1}{2}(X-\mu_1)^T\Sigma^{-1}(X-\mu_1)}$$

$$(X-\mu_0)^T\Sigma^{-1}(X-\mu_0)\gtrless\log\left(\frac{\pi_1}{\pi_0}\left(\frac{|\Sigma|}{|\Sigma|}\right)\right)+(X-\mu_1)^T\Sigma^{-1}(X-\mu_1)$$

$$\iff e^{\beta_0^TX+\gamma_0}\lessgtr e^{\beta_1^TX+\gamma_1}$$

$$\delta_c(x)=\log\left(\frac{P(y=0|X,\theta)}{P(y=1|X,\theta)}\right)=\log\left(\frac{e^{\beta_0^TX+\gamma_0}}{e^{\beta_1^TX+\gamma_1}}\right)=(\beta_0^T-\beta_1^T)X+\gamma_0-\gamma_1\gtrless0$$

  - This is the same discriminant function from the lecture!
- The above is **linear equation**: decision boundaries are linear.
-




- **LDA Training**:
  - Training can be performed by MLE parameter estimation:
    - $\hat{\pi_c}=\frac{n_c}{n}$
    - $\hat{\mu_c}=\frac{1}{n_c}\sum_{i\in c}X_i$
    - $\hat{\Sigma}=\frac{1}{n}\sum_{i\in c}(X_i-\hat{\mu}_{c_i})(X_i-\hat{\mu}_{c_i})^T$
- In Scikit-Learn: `from sklearn.discriminant_analysis import LinearDiscriminantAnalysis`
  - Doc (https://scikit-learn.org/stable/modules/lda_qda.html)
- LDA maximizes **Rayleigh quotient**:

$$max\frac{w^TS_Bw}{w^TS_Ww}$$

  - $S_B$ - The scatter **between** classes
  - $S_W$ - The scatter **within** classes
- LDA maximizes **Fisher criterion**:

$$max\frac{(\tilde{m_1}-\tilde{m_2})^2}{\tilde{s_1}^2+\tilde{s_1}^2}$$

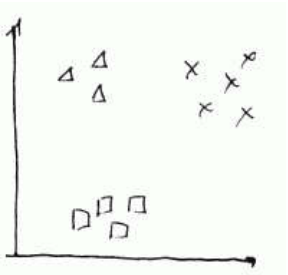
  - $\tilde{m_i},\tilde{s_i}^2$ - The mean and scatter of class $i$ (resp.)
- **Diagonal LDA** - adds another assumption that the covariance matrix is diagonal - features are independent
  - It is a private case of *Naive Bayes*

In [9]:
```
# Lda
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
lda_clf = LDA()
lda_clf.fit(x_train, y_train)
y_pred = lda_clf.predict(x_test)
accuracy = np.sum(y_pred == y_test) / len(y_test)
print("LDA accuracy: {:.3f} %".format(accuracy * 100))
```

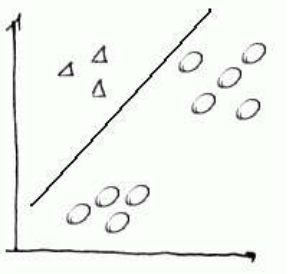
```
LDA accuracy: 90.351 %
```

# One Vs. All Classification

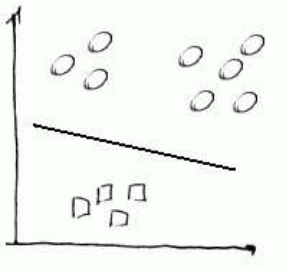
- How can we use binary classifiers when we have multiple labels/classes?
  - The general idea is to build a classifier for each class, that is, if we have $k$ labels/classes, then we would have $k$ classifiers. Each classifier can classify **one** label. Each classifier is trained such that all of the samples that are not of the $i^{th}$ class are 0's, and the samples from the $i^{th}$ class are 1's. In test time, we will have $k$ predictions, one for each class, and we will choose the class that gets the highest probability.
- Formally: Suppose we have the following 3-class problem:



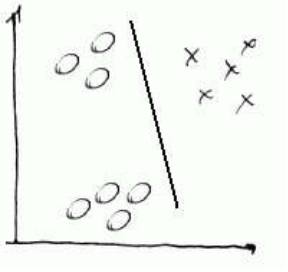
  - We can transform this problem into 3 binary classification problems (i.e. where we predict only $y \in \{0, 1\}$, to be able to use classifiers such as **Logistic Regression.** This is called **One-Vs-All.**
  - We take the values of one class and turn them into *positive* examples, and the rest of classes - *negative*.
  - **Step 1** - *triangles* are positive, the rest are negative and we train a classifier on them. We get $h_\theta^{(1)}(x)$:



  - **Step 2** - the same with the *squares*. We get $h_\theta^{(2)}(x)$:



  - **Step 3** - finally, the *x's* are positive. We get $h_\theta^{(3)}(x)$:



  - We now have 3 classifiers, such that $h_\theta^{(i)}(x) = P(y = i|x; \theta), \text{ for } i = 1, 2, 3$
  - We concatenate the predictions to one vector: $h_\theta(x) = [h_\theta^{(1)}, h_\theta^{(2)}, h_\theta^{(3)}]$
  - We pick the maximal class as the prediciton (i.e., the class that is most probable): $\hat{y} = \underset{i}{argmax}[h_\theta(x)]$
  - This can be done with any classification algorithm that can output probability (Logistic Regression, Decision Trees...)
  - Source: ML Wiki - One-vs-All Classification (http://mlwiki.org/index.php/One-vs-All_Classification)

## Credits

- Icons from Icon8.com (https://icons8.com/) - https://icons8.com (https://icons8.com)
- Datasets from Kaggle (https://www.kaggle.com/) - https://www.kaggle.com/ (https://www.kaggle.com/)
- Examples and code snippets were taken from "Hands-On Machine Learning with Scikit-Learn and TensorFlow" (http://shop.oreilly.com/product/0636920052289.do)