Machine Learning

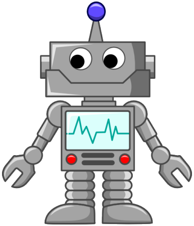
IML

Cédric Buche

ENIB

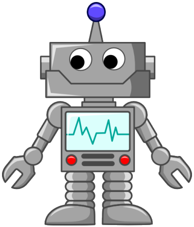
27 août 2019

Human vs Machine



Human vs Machine

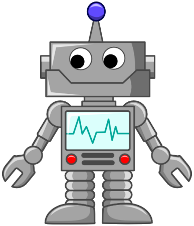
Learn from past experiences



Need to be programmed

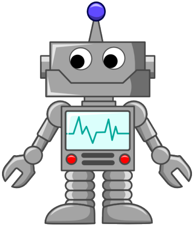
Human vs Machine

Learn from past experiences



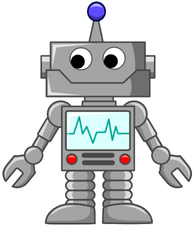
Need to be programmed
Learn from past experiences ?

Human vs Machine



Machine Learning :
teaching computers to learn
to perform tasks
from past experiences

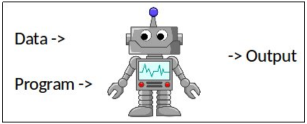
Human vs Machine



Machine Learning :
teaching computers to learn
to perform tasks
from past experiences
Past experiences == data

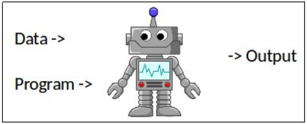
Machine Learning : What is it ?

Traditional Programming

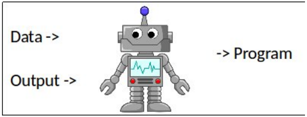


Machine Learning : What is it ?

Traditional Programming



Machine Learning



Goals

- **Classification**
  - Is this cancer ?
  - What did you say ?
- **Prediction**
  - which advertisement a shopper is most likely to click on ?
  - which football team is going to win the Super Bowl ?

$20, 000

$20, 000

$300, 000

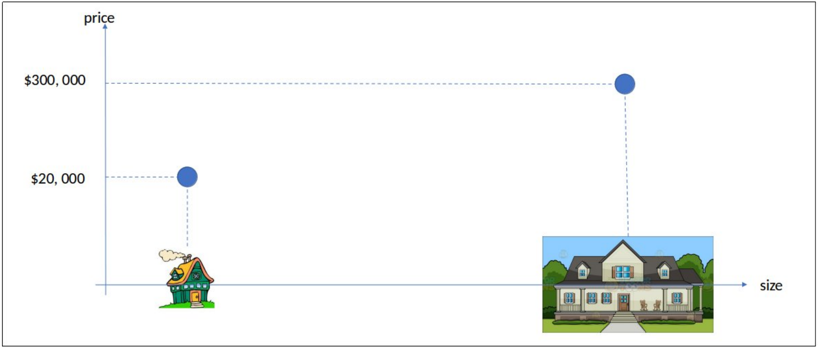
$20, 000

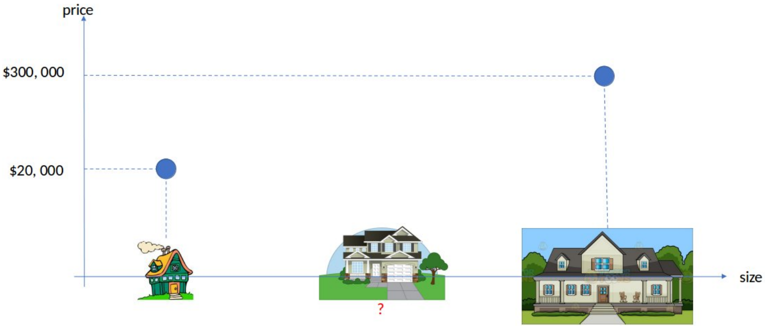
$300, 000

?

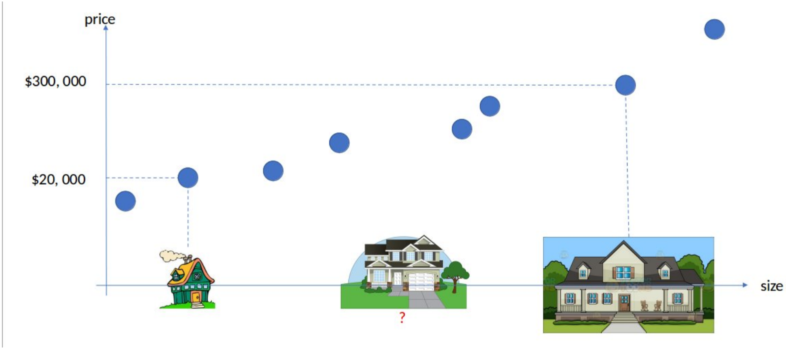
Example : Price of a house

Example : Price of a house

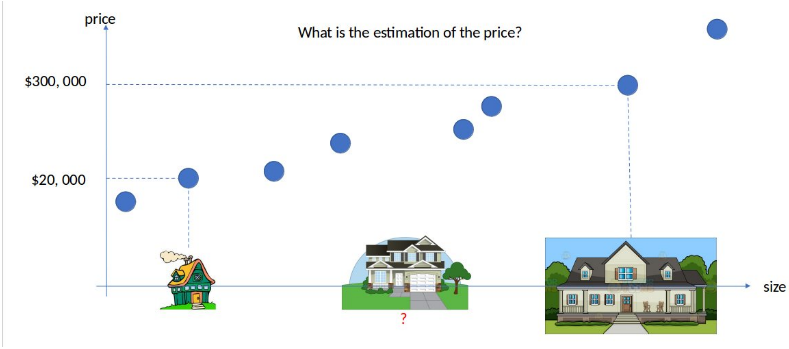



price
$300, 000
$20, 000
?
size

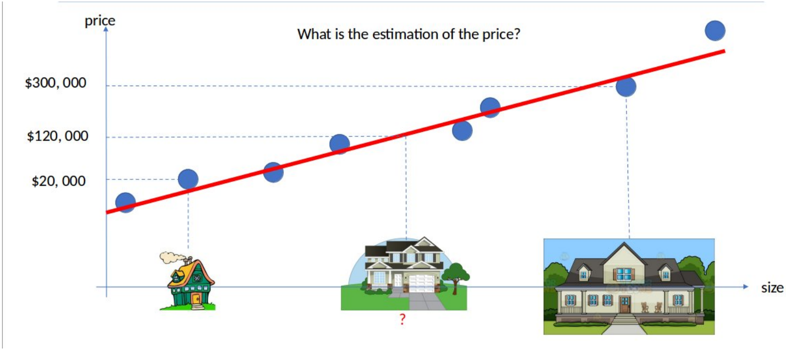
Example : Price of a house



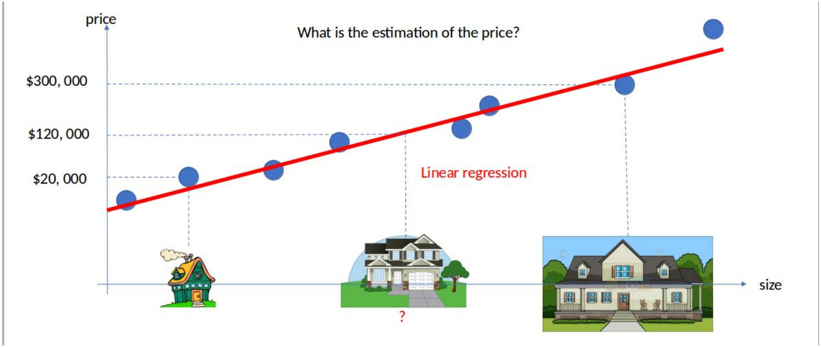
Example : Price of a house



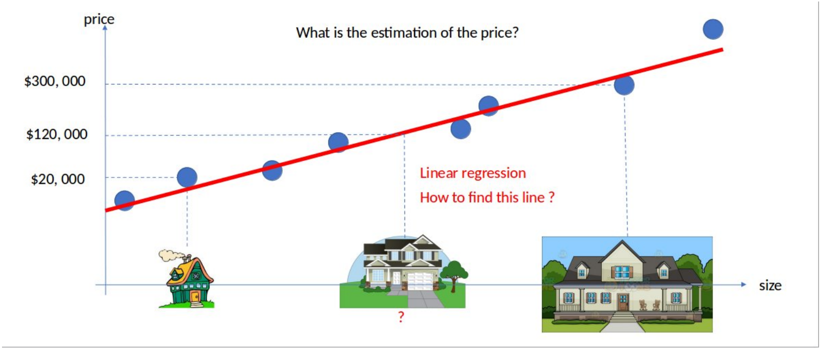
Example : Price of a house



Example : Price of a house



Example : Price of a house



Linear Regression

$$y_i = \beta * x_i + \alpha + \epsilon_i$$

ϵ_i is a (hopefully small) error term representing the fact that there are other factors not accounted for by this simple model.

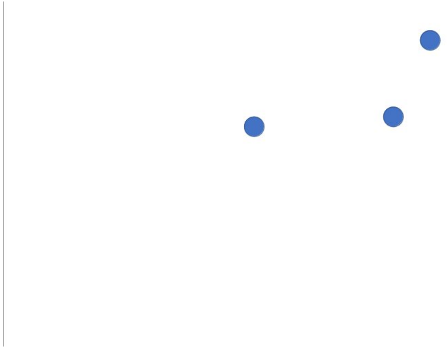
Linear Regression

Assuming we've determined such an alpha and beta, then we make predictions simply with :

```
def predict(alpha,beta,x_i):
       return beta * x_i +alpha
```

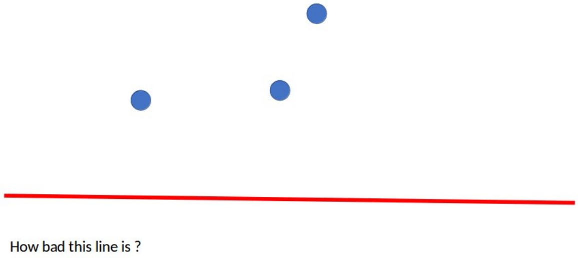
Linear Regression

How do we choose α and β ?



Linear Regression

How do we choose α and β ?



Linear Regression

Any choice of α and β gives us a predicted output for each input x_i . Since we know the actual output y_i we can compute the error for each pair :

```
def error ( alpha , beta , x_i , y_i ):
        return y_i - predict ( alpha , beta , x_i )
```

Linear Regression

We'd really like to know is the total error over the entire data set. But we don't want to just add the errors — if the prediction for x_1 is too high and the prediction for x_2 is too low, the errors may just cancel out.
So instead we add up the squared errors :

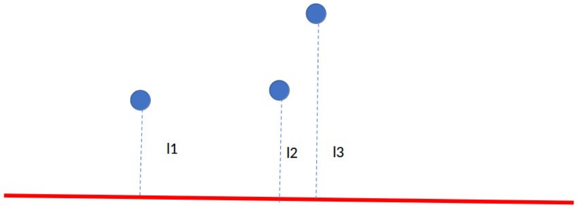
```
def sum_of_squared_errors ( alpha , beta , x , y ):
        return sum ( error ( alpha , beta , x_i , y_i ) ** 2 for x_i , y_i in zip ( x , y ))
```

Linear Regression

The least squares solution is to choose the α and β that make `sum_of_squared_errors` as small as possible. Using calculus (or tedious algebra), the error-minimizing alpha and beta are given by :

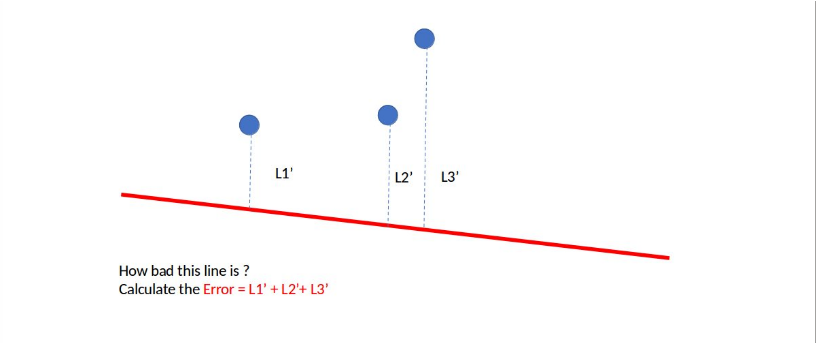
```
def least_squares_fit ( x , y ):
        beta = correlation ( x , y ) * standard_deviation ( y ) / standard_deviation ( x )
        alpha = mean ( y ) - beta * mean ( x )
        return alpha , beta
```

Linear Regression



How bad this line is ?
Calculate the Error = l1 + l2+ l3

Linear Regression

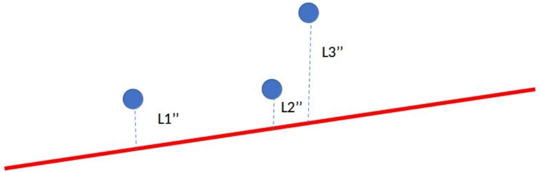


Linear Regression

Of course, we need a better way to figure out how well we've fit the data than staring at the graph. A common measure is the coefficient of determination (or R-squared), which measures the fraction of the total variation in the dependent variable that is captured by the model :

```
def total_sum_of_squares ( y ):
        return sum ( v ** 2 for v in de_mean ( y ))

def r_squared ( alpha , beta , x , y ):
        return 1.0 - ( sum_of_squared_errors ( alpha , beta , x , y ) / total_sum_of_squares ( y
               ))
```

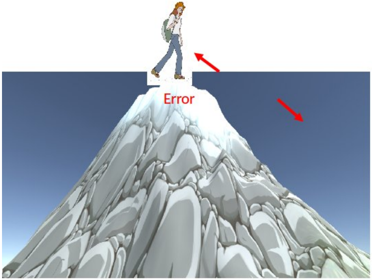
Linear Regression

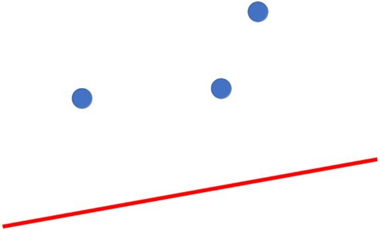


How bad this line is ?
Calculate the Error = l1'' + l2''+ l3''

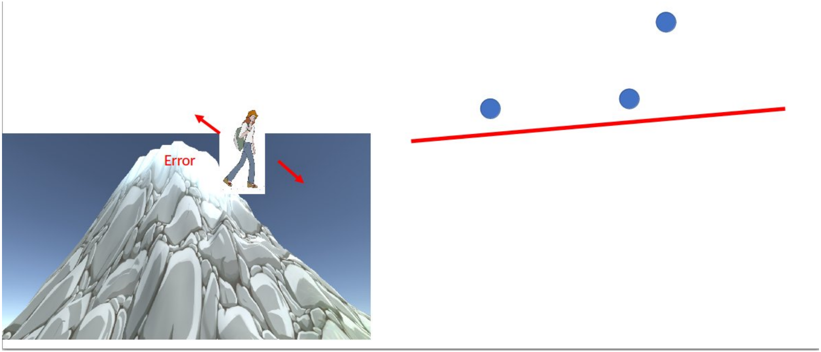
Procedure to decrease the error :
Gradient Descent

Linear Regression

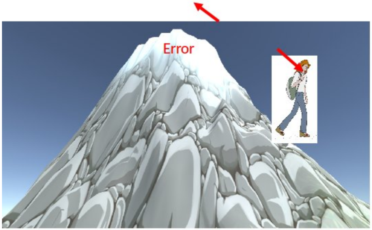


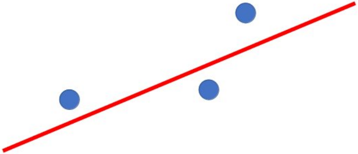


Linear Regression



Linear Regression





Linear Regression

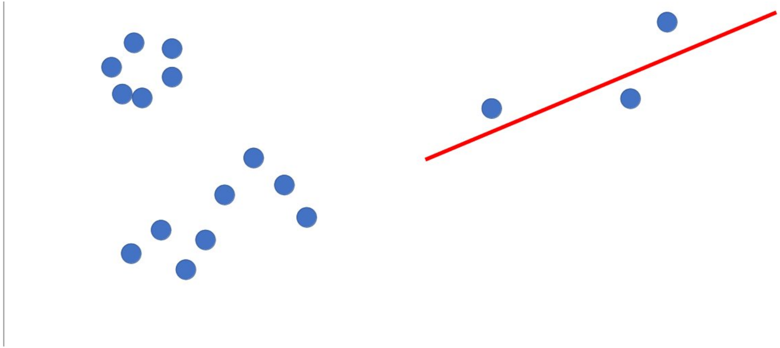
If we write theta = [alpha, beta] , then we can also solve this using gradient descent :

```
def squared_error ( x_i , y_i , theta ):
        alpha , beta = theta
        return error ( alpha , beta , x_i , y_i ) ** 2

def squared_error_gradient ( x_i , y_i , theta ):
        alpha , beta = theta
        return [ - 2 * error ( alpha , beta , x_i , y_i ), # alpha partial deriv
                 - 2 * error ( alpha , beta , x_i , y_i ) * x_i ] # beta partial deriv

# choose random value to start
random.seed ( 0 )
theta = [ random . random (), random . random ()]
alpha , beta = minimize_stochastic ( squared_error , squared_error_gradient , entry , entry2 ,
      theta , 0.0001 )

print alpha , beta
```

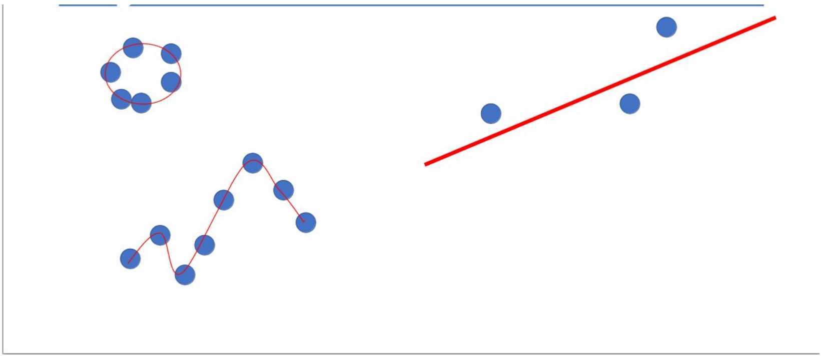
Linear Regression

Demo !

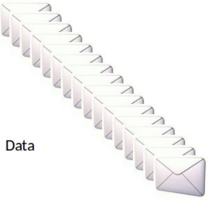
Polynomial Regression



Polynomial Regression

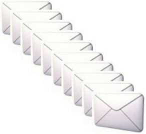


Example : Spam Detector



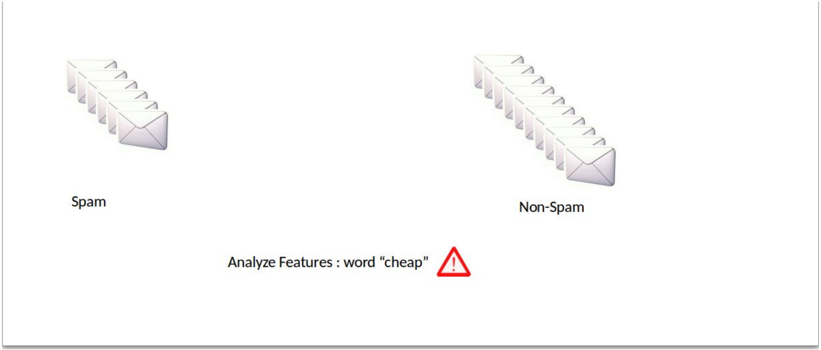
Example : Spam Detector

Spam

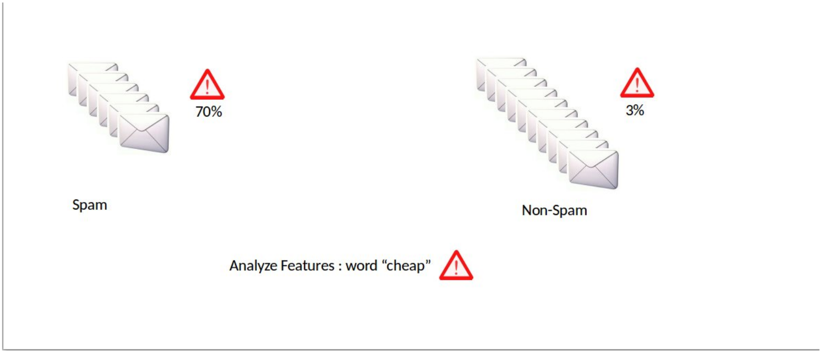


Non-Spam

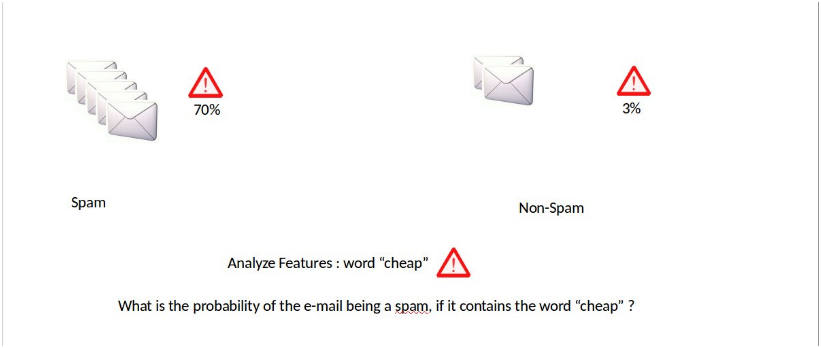
Example : Spam Detector



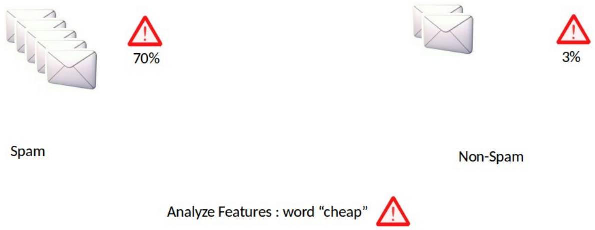
Example : Spam Detector



Example : Spam Detector



Example : Spam Detector



Example : Spam Detector

70%

3%

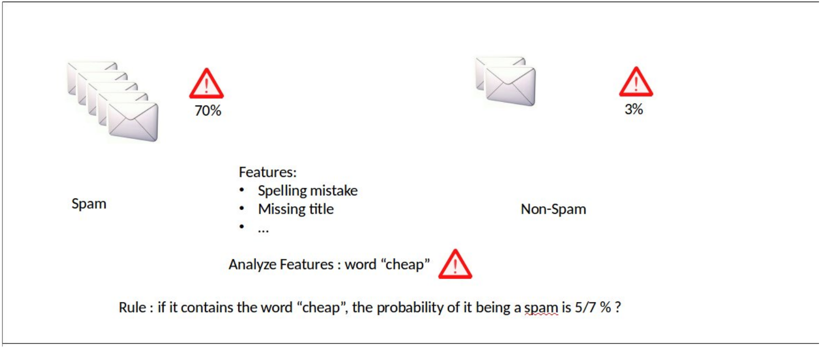
Spam

Non-Spam

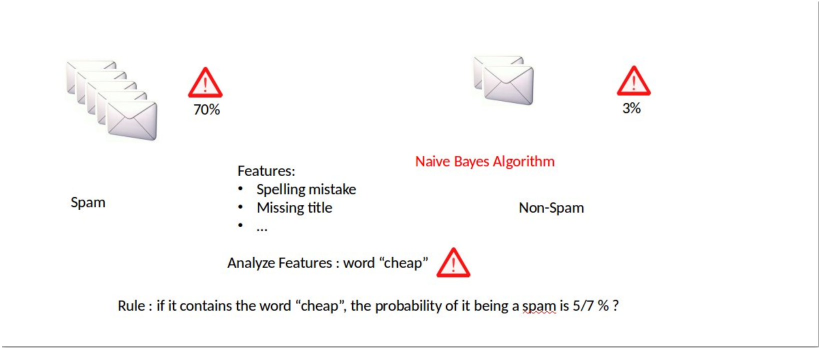
Analyze Features : word “cheap”

Rule : if it contains the word “cheap”, the probability of it being a spam is 5/7 % ?

Example : Spam Detector



Example : Spam Detector



Naive Bayes

- Let S be the event "the message is spam"
- a vocabulary of many words $w_1, ...w_n$
- $P(X_i|S)$: probability that a spam message contains the ith word
- The key to Naive Bayes is making the (big) assumption that the presences (or absences) of each word are independent of one another, conditional on a message being spam or not.
- $P(X_1 = x_1, ...X_n = x_n|S) = P(X_1 = x_1|S) * ...P(X_n = x_n|S)$
- Bayes's Theorem :
  $P(S\ |X = x) = P(X = x|S)/[P(X = x|S) + P(X = x|\neg S)]$

Naive Bayes

- we usually compute $p_1 * ... * p_n$ as the equivalent : $exp(log(p-1) + ... + log(p_n))$
- Imagine that in our training set the vocabulary word “data” only occurs in nonspam messages. Then we'd estimate $P(''data''|S) = 0$
- $P(X_i|S) = (k + numberSpamsContainingw_i)/(2k + numberSpams)$

Naive Bayes

```
def tokenize(message):
        message = message.lower ()                          # convert to lowercase
        all_words = re.findall ( "[a-z0-9']+" , message ) # extract the words
        return set ( all_words )                            # remove duplicates
```

```
def count_words ( training_set ):
        """training␣set␣consists␣of␣pairs␣(message,␣is_spam)"""
        counts = defaultdict ( lambda : [ 0 , 0 ])
        for message , is_spam in training_set :
                for word in tokenize ( message ):
                        counts [ word ][ 0 if is_spam else 1 ] += 1
        return counts
```

Naive Bayes

```
def word_probabilities ( counts , total_spams , total_non_spams , k = 0.5 ):
        # turn the word_counts into a list of triplets
        # w, p(w | spam) and p(w | ~spam)
        return [        ( w , ( spam + k ) / ( total_spams + 2 * k ),
                        ( non_spam + k ) / ( total_non_spams + 2 * k )) for w ,
                        ( spam , non_spam ) in counts.iteritems ()]
```

Naive Bayes

```
def spam_probability ( word_probs , message ):
        message_words = tokenize ( message ) log_prob_if_spam = log_prob_if_not_spam = 0.0

        # iterate through each word in our vocabulary
        for word , prob_if_spam , prob_if_not_spam in word_probs :

        # if *word* appears in the message ,
        # add the log probability of seeing it
        if word in message_words :
                log_prob_if_spam += math . log ( prob_if_spam )
                log_prob_if_not_spam += math . log ( prob_if_not_spam )

        # if *word* does not appear in the message
        # add the log probability of _not_ seeing it
        # which is log(1 - probability of seeing it)
        else :
                log_prob_if_spam += math . log ( 1.0 - prob_if_spam )
                log_prob_if_not_spam += math . log ( 1.0 - prob_if_not_spam )

        prob_if_spam = math . exp ( log_prob_if_spam )
        prob_if_not_spam = math . exp ( log_prob_if_not_spam )
        return prob_if_spam / ( prob_if_spam + prob_if_not_spam )
```

Naive Bayes

```
class NaiveBayesClassifier :

        def __init__ ( self , k = 0.5 ):
                self . k = k
                self . word_probs = []

        def train ( self , training_set ):
        # count spam and non-spam messages
        num_spams = len ([ is_spam for message , is_spam in training_set if is_spam ])
        num_non_spams = len ( training_set ) - num_spams

        # run training data through our "pipeline"
        word_counts = count_words ( training_set )
        self.word_probs = word_probabilities ( word_counts , num_spams , num_non_spams , self.k
              )

        def classify ( self , message ):
                return spam_probability ( self . word_probs , message)
```

Naive Bayes

```
import glob , re
# modify the path with wherever you have put the files
path = .....
data = []

# glob.glob returns every filename that matches the wildcarded path
for fn in glob.glob ( path ):
        is_spam = "ham" not in fn

        with open ( fn , 'r' ) as file :
                for line in file :
                        if line.startswith( "Subject:" ):
                        # remove the leading "Subject:␣" and keep what is left
                        subject = re.sub( r"^Subject:␣" , "" , line ).strip ()
                        data.append(( subject , is_spam ))
```

Naive Bayes

```
random.seed ( 0 ) # just so you get the same answers as me
train_data , test_data = split_data ( data , 0.75 )

classifier = NaiveBayesClassifier ()
classifier.train ( train_data )

# triplets (subject, actual is_spam, predicted spam probability)
classified = [( subject , is_spam , classifier . classify ( subject )) for subject , is_spam in
      test_data ]
# assume that spam_probability > 0.5 corresponds to spam prediction
# and count the combinations of (actual is_spam, predicted is_spam)
counts = Counter (( is_spam , spam_probability > 0.5 ) for _ , is_spam , spam_probability in
      classified )
```

Naive Bayes

Demo !

Example : Recommending apps

Gender	Age	App
F	15	Facebook
F	25	Instagram
M	32	Snapchat
F	40	Instagram
M	12	Facebook
M	14	Facebook

Example : Recommending apps

Gender	Age	App
F	15	Facebook
F	25	Instagram
M	32	Snapchat
F	40	Instagram
M	12	Facebook
M	14	Facebook

Which feature (Gender or Age) is the more decisive to predict what app will the users download ?

Example : Recommending apps

Gender	Age	App
F	15	Facebook
F	25	Instagram
M	32	Snapchat
F	40	Instagram
M	12	Facebook
M	14	Facebook

Which feature (Gender or Age) is the more decisive to predict what app will the users download ?

$Age < 20$: *Facebook*

Example : Recommending apps

Gender	Age	App
F	15	Facebook
F	25	Instagram
M	32	Snapchat
F	40	Instagram
M	12	Facebook
M	14	Facebook

Which feature (Gender or Age) is the more decisive to predict what app will the users download ?

$Age < 20$: *Facebook*

$Age > 20$:?

Example : Recommending apps

Gender	Age	App
F	15	Facebook
F	25	Instagram
M	32	Snapchat
F	40	Instagram
M	12	Facebook
M	14	Facebook

Which feature (Gender or Age) is the more decisive to predict what app will the users download ?

$Age < 20$: *Facebook*

$Age > 20$:?

$Age > 20$: F : *Instagram* M : *Snapchat*

Example : Recommending apps

Gender	Age	App
F	15	Facebook
F	25	Instagram
M	32	Snapchat
F	40	Instagram
M	12	Facebook
M	14	Facebook

Which feature (Gender or Age) is the more decisive to predict what app will the users download ?

$Age < 20$: *Facebook*

$Age > 20$:?

$Age > 20$: F : *Instagram* M : *Snapchat*

Decision Tree

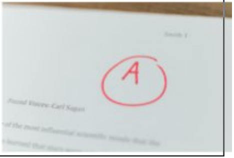
Example : Acceptance at a University

buche@enib.fr

Example : Acceptance at a University

Example : Acceptance at a University

Test



Example : Acceptance at a University

Test

Grades

Student 1
Test : 9/10
Grades : 8/10

ACCEPTED

Example : Acceptance at a University

Test

Grades

Student 1
Test : 9/10
Grades : 8/10

ACCEPTED

Student 3
Test : 7/10
Grades : 6/10

ACCEPTED ??

Example : Acceptance at a University

Test

Grades

Student 1
Test : 9/10
Grades : 8/10

ACCEPTED

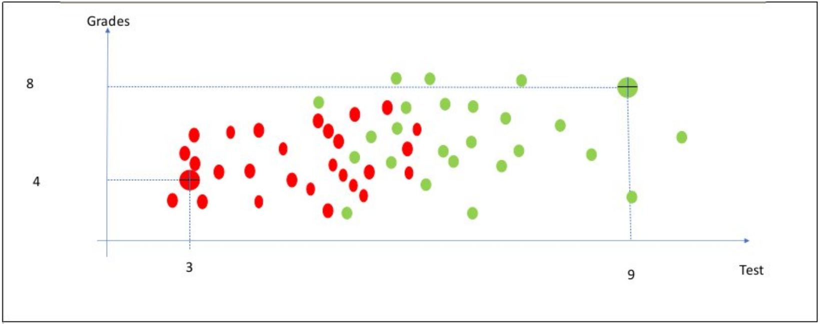
Student 2
Test : 3/10
Grades : 4/10

NOT ACCEPTED

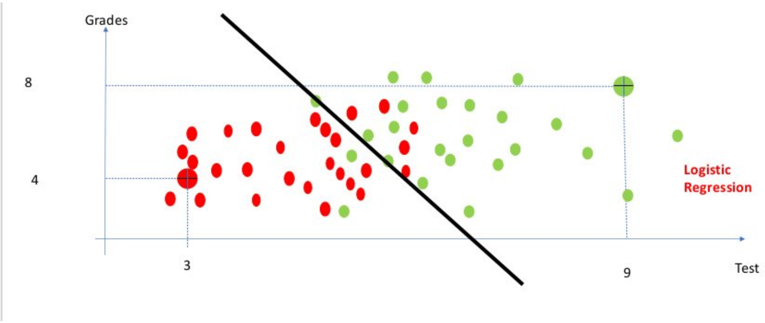
Student 3
Test : 7/10
Grades : 6/10

ACCEPTED ??

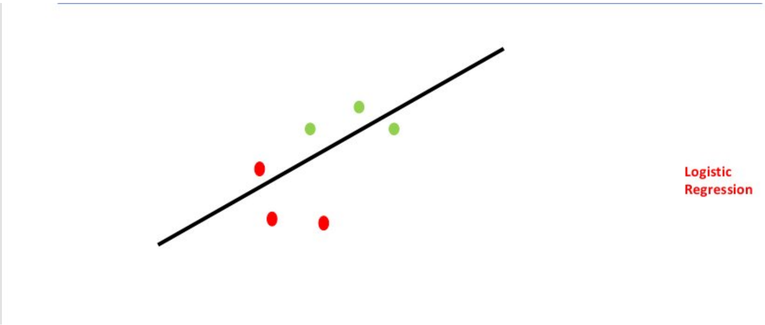
Example : Acceptance at a University



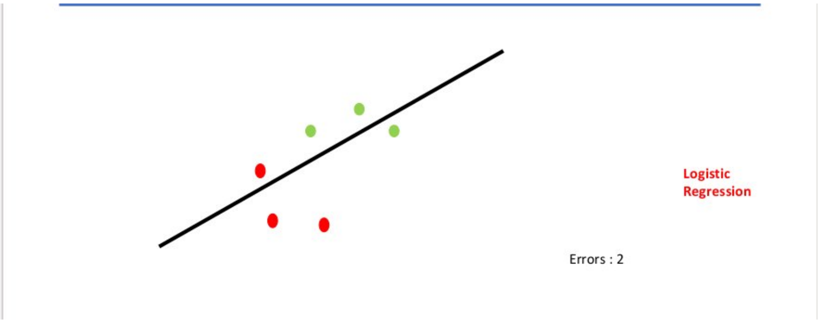
Example : Acceptance at a University



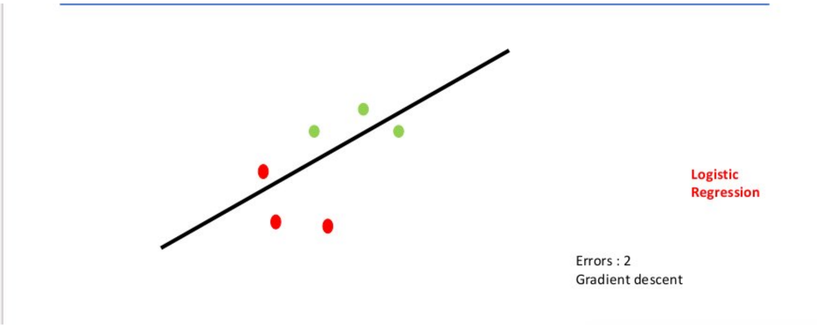
Example : Acceptance at a University



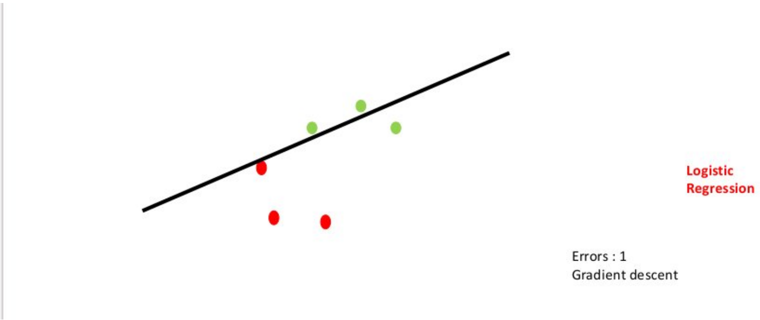
Example : Acceptance at a University



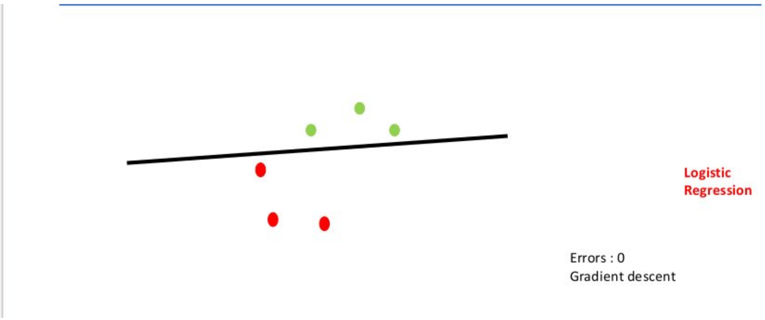
Example : Acceptance at a University



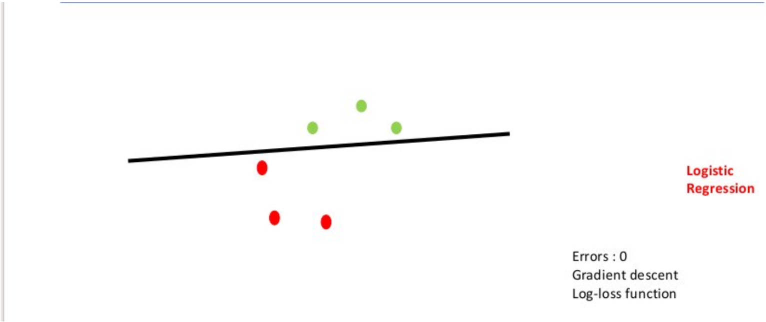
Example : Acceptance at a University



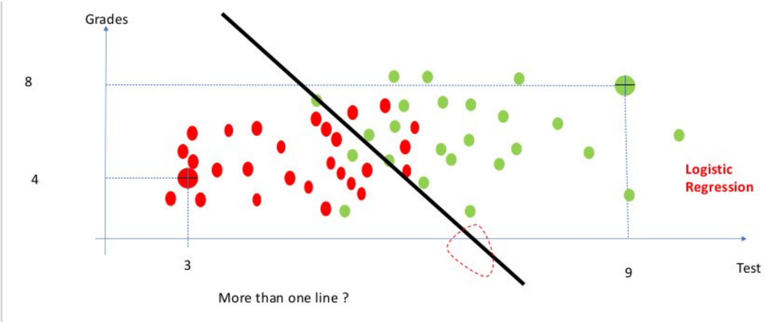
Example : Acceptance at a University



Example : Acceptance at a University

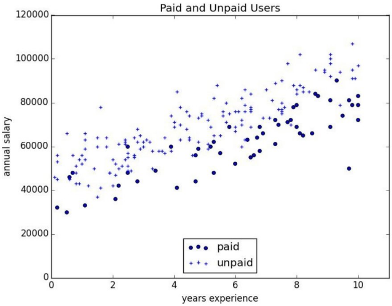


Example : Acceptance at a University



Logistic regression

- We have an anonymized data set of about 200 users, containing each user's salary, her years of experience as a data scientist, and whether she paid for a premium account=



Logistic regression

- As is usual with categorical variables, we represent the dependent variable as either 0 (no premium account) or 1 (premium account).
- our data is in a matrix where each row is a list [experience, salary, paid_account]

```
x = [[ 1 ] + row [: 2 ] for row in data ] # each element is [1, experience , salary]
y = [ row [ 2 ] for row in data ] # each element is paid_account
```

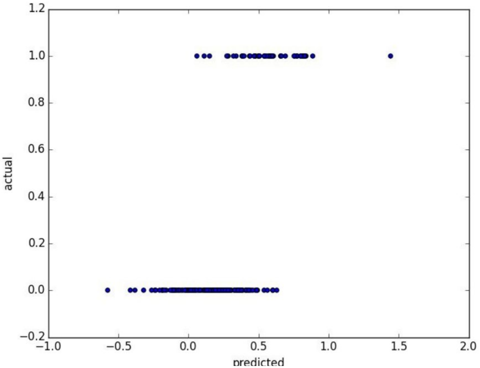
Logistic regression

- linear regression :
  $paidAccount = \beta_0 + \beta_1 * experience + \beta_2 * salary + \epsilon$

```
rescaled_x = rescale ( x )
beta = estimate_beta ( rescaled_x , y ) # [0.26, 0.43, -0.43]
predictions = [ predict ( x_i , beta ) for x_i in rescaled_x ]
plt.scatter ( predictions , y )
plt.xlabel ( "predicted" )
plt.ylabel ( "actual" )
plt.show ()
```

Logistic regression

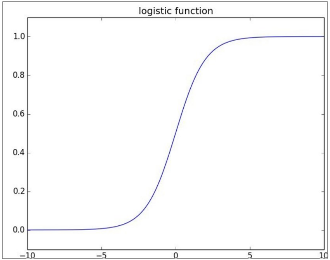
- result (linear regression) :



Logistic regression

- ▷ logistic regression (logistic function) :

```
def logistic ( x ):
      return 1.0 / ( 1 + math . exp ( - x ))
```



Logistic regression

- derivative is given by :

```
def logistic_prime ( x ):
      return logistic ( x ) * ( 1 - logistic ( x ))
```

$$y_i = f(x_i\beta) + \epsilon_i$$

f is the logistic funtion

Logistic regression

```
def logistic_log_likelihood_i ( x_i , y_i , beta ):
        if y_i == 1 :
                return math . log ( logistic ( dot ( x_i , beta )))
        else :
                return math . log ( 1 - logistic ( dot ( x_i , beta )))

def logistic_log_likelihood ( x , y , beta ):
        return sum ( logistic_log_likelihood_i ( x_i , y_i , beta ) for x_i , y_i in zip ( x , y
               ))

def logistic_log_gradient_i ( x_i , y_i , beta ):
        # the gradient of the log likelihood corresponding to the ith data point
        return [ logistic_log_partial_ij ( x_i , y_i , beta , j ) for j , _ in enumerate ( beta
              )]

def logistic_log_gradient ( x , y , beta ):
        return reduce ( vector_add , [ logistic_log_gradient_i ( x_i , y_i , beta ) for x_i ,
              y_i in zip ( x , y )])
```

Logistic regression

```
random . seed ( 0 )
x_train , x_test , y_train , y_test = train_test_split ( rescaled_x , y , 0.33 )

# want to maximize log likelihood on the training data
fn = partial ( logistic_log_likelihood , x_train , y_train )
gradient_fn = partial ( logistic_log_gradient , x_train , y_train )

# pick a random starting point
beta_0 = [ random . random () for _ in range ( 3 )]

# and maximize using gradient descent
beta_hat = maximize_batch ( fn , gradient_fn , beta_0 )
```

Naive Bayes

Demo !

Model : KNN

Examples

- predict how I'm going to vote in the next presidential election. If you know nothing else about me, one approach is to look at how my neighbors are planning to vote. Living in Seattle, my neighbors are planning to vote for the Democratic candidate, which suggests that "Democratic candidate" is a good guess for me as well.
- you know more about me : my age, my income, how many kids I have ... To the extent my behavior is influenced by those things, looking just at my neighbors who are close to me among all those dimensions seems likely to be an even better predictor than looking at all my neighbors. This is the idea behind *nearest neighbors classification*.

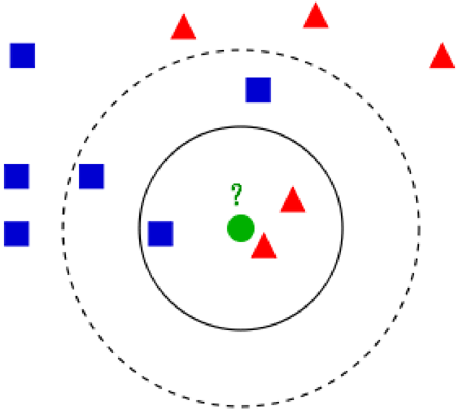
Model : KNN

Requirements

- Some notion of distance
- An assumption that points that are close to one another are similar

the prediction for each new point depends only on the handful of points closest to it.

Model : KNN



Model : KNN

- classify some new data point : find the k nearest labeled points and let them vote on the new output.
- need a function that counts votes : Reduce k until we find a unique winner.

```
def majority_vote(labels):
   # assumes that labels are ordered from nearest to farthest
  vote_counts = Counter(labels)
  winner, winner_count = vote_counts.most_common(1)[0]
  num_winners = len([count for count in vote_counts.values() if count == winner_count])
  if num_winners == 1:
      return winner # unique winner, so return it
  else:
      return majority_vote(labels[:-1]) # try again without the farthest
```

Model : KNN

```
def knn_classify(k, labeled_points, new_point):
  # each labeled point should be a pair (point, label)

  # order the labeled points from nearest to farthest
  by_distance = sorted(labeled_points, key=lambda (point, _): distance(point, new_point))

  # find the labels for the k closest
  k_nearest_labels = [label for _, label in by_distance[:k]]

  # and let them vote
  return majority_vote(k_nearest_labels)
```

Example : Favorite Programming Languages

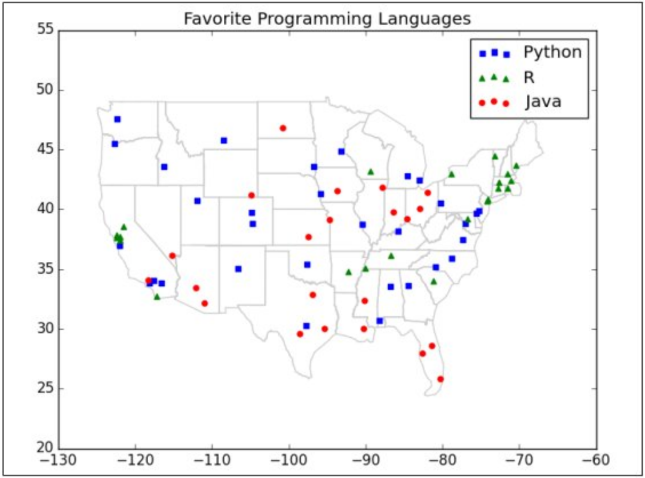
```
# each entry is ([longitude, latitude], favorite_language)
cities = [([-122.3 , 47.53], "Python"), # Seattle
          ([ -96.85, 32.85], "Java"),   # Austin
          ([ -89.33, 43.13], "R"),      # Madison
          # ... and so on
]
```

Example : Favorite Programming Languages

Plotting the data

```
# key is language, value is pair (longitudes, latitudes)
plots = { "Java" : ([], []), "Python" : ([], []), "R" : ([], []) }

# we want each language to have a different marker and color
markers = { "Java" : "o", "Python" : "s", "R" : "^" }
colors = { "Java" : "r", "Python" : "b", "R" : "g" }

for (longitude, latitude), language in cities:
    plots[language][0].append(longitude)
    plots[language][1].append(latitude)

# create a scatter series for each language
for language, (x, y) in plots.iteritems():
  plt.scatter(x, y, color=colors[language], marker=markers[language],
            label=language, zorder=10)

plot_state_borders(plt) # pretend we have a function that does this

plt.legend(loc=0) # let matplotlib choose the location
plt.axis([-130,-60,20,55]) # set the axes

plt.title("Favorite␣Programming␣Languages")
plt.show()
```

Result


Favorite Programming Languages
Python
R
Java
55
50
45
40
35
30
25
20
−130
−120
−110
−100
−90
−80
−70
−60

Example : Favorite Programming Languages

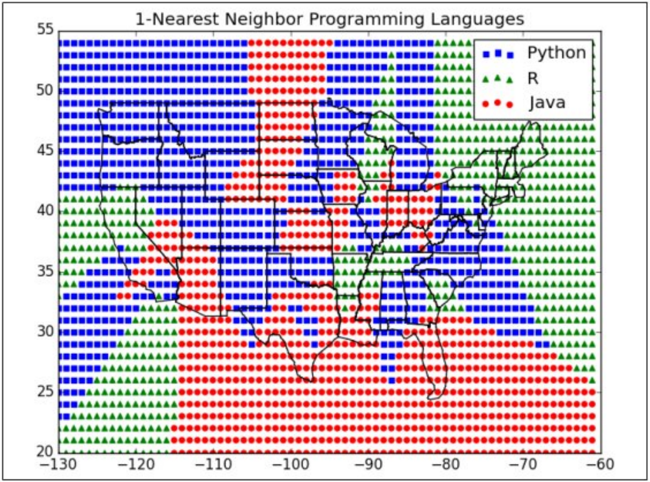
Try several different values for k

```
for k in [1, 3, 5, 7]:
    num_correct = 0
for city in cities:
    location, actual_language = city
    other_cities = [other_city
                    for other_city in cities
                    if other_city != city]

    predicted_language = knn_classify(k, other_cities, location)

    if predicted_language == actual_language:
        num_correct += 1

    print k, "neighbor[s]:", num_correct, "correct out of", len(cities)
```

Example : Favorite Programming Languages

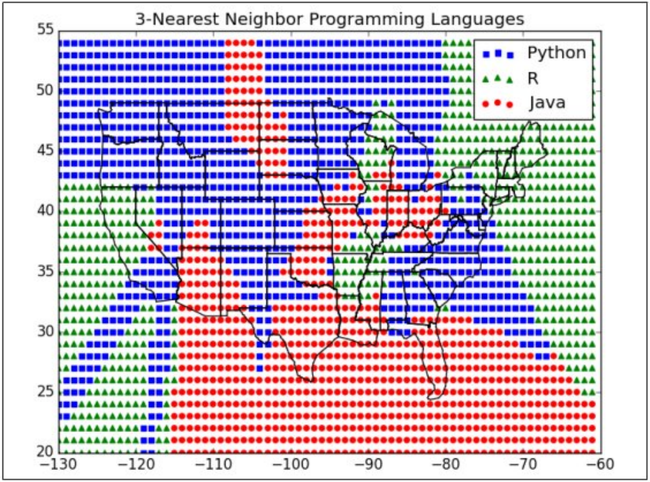
```
plots = { "Java" : ([], []), "Python" : ([], []), "R" : ([], []) }

k = 1 # or 3, or 5, or ...

for longitude in range(-130, -60):
  for latitude in range(20, 55):
    predicted_language = knn_classify(k, cities, [longitude, latitude])
    plots[predicted_language][0].append(longitude)
    plots[predicted_language][1].append(latitude)
```

Result


1-Nearest Neighbor Programming Languages
Python
R
Java
55
50
45
40
35
30
25
20
−130
−120
−110
−100
−90
−80
−70
−60

$k = 1$

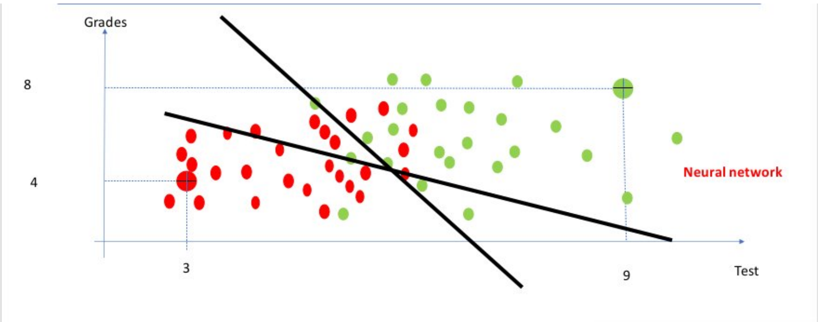
Result


3-Nearest Neighbor Programming Languages
Python
R
Java
55
50
45
40
35
30
25
20
−130
−120
−110
−100
−90
−80
−70
−60

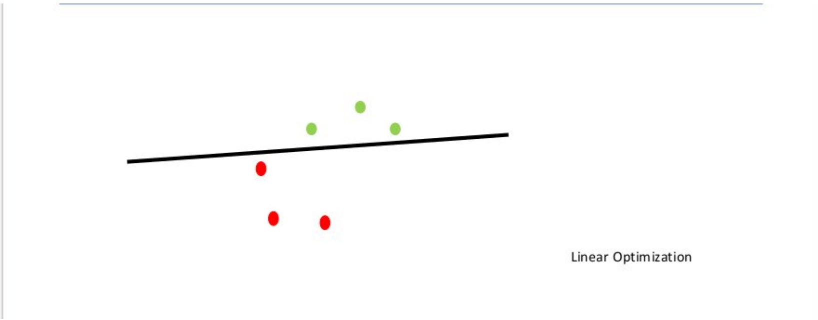
$k = 3$

KNN

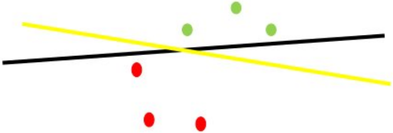
Demo !


Grades
8
4
3
9
Test
Neural network

SVM

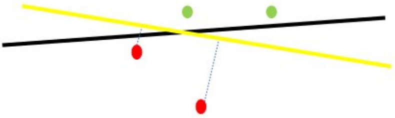


SVM



Linear Optimization

SVM

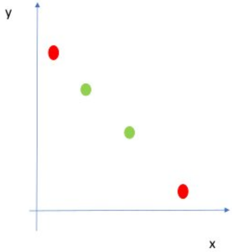


Support Vector Machine

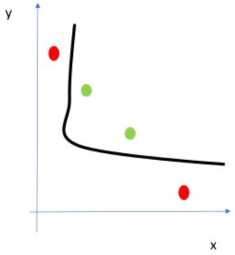
Linear Optimization

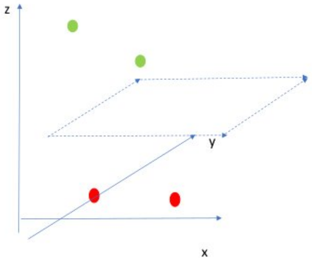
SVM

SVM



SVM : kernel trick





1. Machine Learning
2. Supervised - Regression
   - Linear regression
   - Polynomial regression
3. Supervised - Classification
   - Naive Bayes
   - Decision Tree
   - Logistic regression
   - KNN
   - Neural network
   - SVM
4. Unsupervised - Clustering
   - k-means
   - Hierarchical clustering
   - Distance

Unsupervised learning

Learning mode

- supervised learning : set of labeled data for making predictions about new, unlabeled data.
- unsupervised learning : no label at all
- Whenever you look at some source of data, the data will somehow form *clusters*.

Idea

Examples

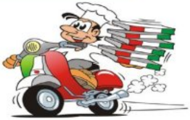
- A data set showing where millionaires live probably has clusters in places like Beverly Hills and Manhattan.
- A data set showing how many hours people work each week probably has a cluster around 40.
- A data set of demographics of registered voters likely forms a variety of clusters (e.g., "soccer moms", "bored retirees" ...)

the clusters won't label themselves. You'll have to do that by looking at the data underlying each one.

Model : k-means

1. Start with a set of k-means, which are points in d-dimensional space.
2. Assign each point to the mean to which it is closest.
3. If no point's assignment has changed, stop and keep the clusters.
4. If some point's assignment has changed, recompute the means and return to step 2.

Example : pizza



Pizza chain

Optimal location ?

Example : pizza

Pizza chain

Optimal location ?

Example : pizza

How to teach the PC to do that ?

Example : pizza

Example : pizza

Example : pizza

Model : k-means

```
def vector_mean(vectors):
  # compute the vector whose ith element is the mean of the ith elements of the input vectors
  n = len(vectors)
  return scalar_multiply(1/n, vector_sum(vectors))

class KMeans:

  def __init__(self, k):
    self.k = k         # number of clusters
    self.means = None # means of clusters

  def classify(self, input):
    # return the index of the cluster closest to the input
    return min(range(self.k),key=lambda i: squared_distance(input, self.means[i]))
```

Model : k-means

```
def train(self, inputs):
  # choose k random points as the initial means
  self.means = random.sample(inputs, self.k)
  assignments = None

while True:
  # Find new assignments
  new_assignments = map(self.classify, inputs)

# If no assignments have changed, we are done
if assignments == new_assignments:
  return

# Otherwise keep the new assignments,
assignments = new_assignments

# And compute new means based on the new assignments
for i in range(self.k):
  # find all the points assigned to cluster i
  i_points = [p for p, a in zip(inputs, assignments) if a == i]

  # make sure i_points is not empty so do not divide by 0
  if i_points:
    self.means[i] = vector_mean(i_points)
```

Example : stickers

Context

- sticker printer can print at most five colors per sticker.
- there's some way to take a design and modify it so that it only contains five colors ?

Data

- images can be represented as two-dimensional array of pixels, where each pixel is itself a three-dimensional vector (red, green, blue) indicating its color.
- five-color version of the image
  1. Choosing five colors
  2. Assigning one of those colors to each pixel

Example : stickers

```
path_to_png_file = r"C:\images\image.png"
import matplotlib.image as mpimg
img = mpimg.imread(path_to_png_file)

top_row = img[0]
top_left_pixel = top_row[0]
red, green, blue = top_left_pixel

pixels = [pixel for row in img for pixel in row]

clusterer = KMeans(5)
clusterer.train(pixels)

def recolor(pixel):
  cluster = clusterer.classify(pixel)
  return clusterer.means[cluster]

new_img = [[recolor(pixel) for pixel in row]
           for row in img]

plt.imshow(new_img)
plt.axis('off')
plt.show()
```

K means

Demo !

Alternative approach

"grow" clusters from the bottom up

1. Make each input its own cluster of one.
2. As long as there are multiple clusters remaining, find the two closest clusters and merge them.
3. At the end, we'll have one giant cluster containing all the inputs. If we keep track of the merge order, we can recreate any number of clusters by unmerging. For example, if we want three clusters, we can just undo the last two merges.

k-means vs Hierarchical Clustering : HC do not need to specify k

Distance

Name	Egg-laying	Scales	Poisonous	Cold-blooded	Legs nb	**Reptile**
Cobra	True	True	True	True	0	Yes
Rattlesnake	True	True	True	True	0	Yes
Boa	False	True	False	True	0	Yes
Chicken	True	True	False	False	2	No
Alligator	True	True	False	True	4	Yes
Frog	True	False	True	True	4	No
Salmon	True	True	False	True	0	No
Python	True	True	False	True	0	Yes

Distance

Name	Egg-laying	Scales	Poisonous	Cold-blooded	Legs nb	**Reptile**
Cobra	True	True	True	True	0	Yes
Rattlesnake	True	True	True	True	0	Yes
Boa	False	True	False	True	0	Yes
Chicken	True	True	False	False	2	No
Alligator	True	True	False	True	4	Yes
Frog	True	False	True	True	4	No
Salmon	True	True	False	True	0	No
Python	True	True	False	True	0	Yes

Features = four binary and one integer

Distance

Name	Egg-laying	Scales	Poisonous	Cold-blooded	Legs nb	**Reptile**
Cobra	True	True	True	True	0	Yes
Rattlesnake	True	True	True	True	0	Yes
Boa	False	True	False	True	0	Yes
Chicken	True	True	False	False	2	No
Alligator	True	True	False	True	4	Yes
Frog	True	False	True	True	4	No
Salmon	True	True	False	True	0	No
Python	True	True	False	True	0	Yes

Features = four binary and one integer
Boa = (0,1,0,1,0)

Distance

Name	Egg-laying	Scales	Poisonous	Cold-blooded	Legs nb	**Reptile**
Cobra	True	True	True	True	0	Yes
Rattlesnake	True	True	True	True	0	Yes
Boa	False	True	False	True	0	Yes
Chicken	True	True	False	False	2	No
Alligator	True	True	False	True	4	Yes
Frog	True	False	True	True	4	No
Salmon	True	True	False	True	0	No
Python	True	True	False	True	0	Yes

Features = four binary and one integer
Boa = (0,1,0,1,0)
Frog =(1,0,1,0,4)

Distance

Name	Egg-laying	Scales	Poisonous	Cold-blooded	Legs nb	**Reptile**
Cobra	True	True	True	True	0	Yes
Rattlesnake	True	True	True	True	0	Yes
Boa	False	True	False	True	0	Yes
Chicken	True	True	False	False	2	No
Alligator	True	True	False	True	4	Yes
Frog	True	False	True	True	4	No
Salmon	True	True	False	True	0	No
Python	True	True	False	True	0	Yes

Features = four binary and one integer
Boa = (0,1,0,1,0)
Frog =(1,0,1,0,4)
Distance to separate ?

Distance : Euclidean

	rattlesnake	boa	frog
rattlesnake		1.4	4.2
boa	1.4		4.4
frog	4.2	4.4	

Distance : Euclidean

	rattlesnake	boa	frog	Alligator
rattlesnake		1.4	4.2	4.1
boa	1.4		4.4	4.1
frog	4.2	4.4		1.7
Alligator	4.1	4.1	1.7	

Alligator is closer to a frog than a snake

Distance : Euclidean

	rattlesnake	boa	frog	Alligator
rattlesnake		1.4	1.7	1.4
boa	1.4		2.2	1.4
frog	1.7	2.2		1.7
Alligator	1.4	1.4	1.7	

Using binary Feature : Alligator is closer to a snake than a frog
Feature Engineering Matters

Machine Learning

IML

Cédric Buche

ENIB

27 août 2019