Kernel Methods and SVMs

Dr. Johan Hagelbäck

johan.hagelback@lnu.se

http://aiguy.org

Kernel Methods and SVMs

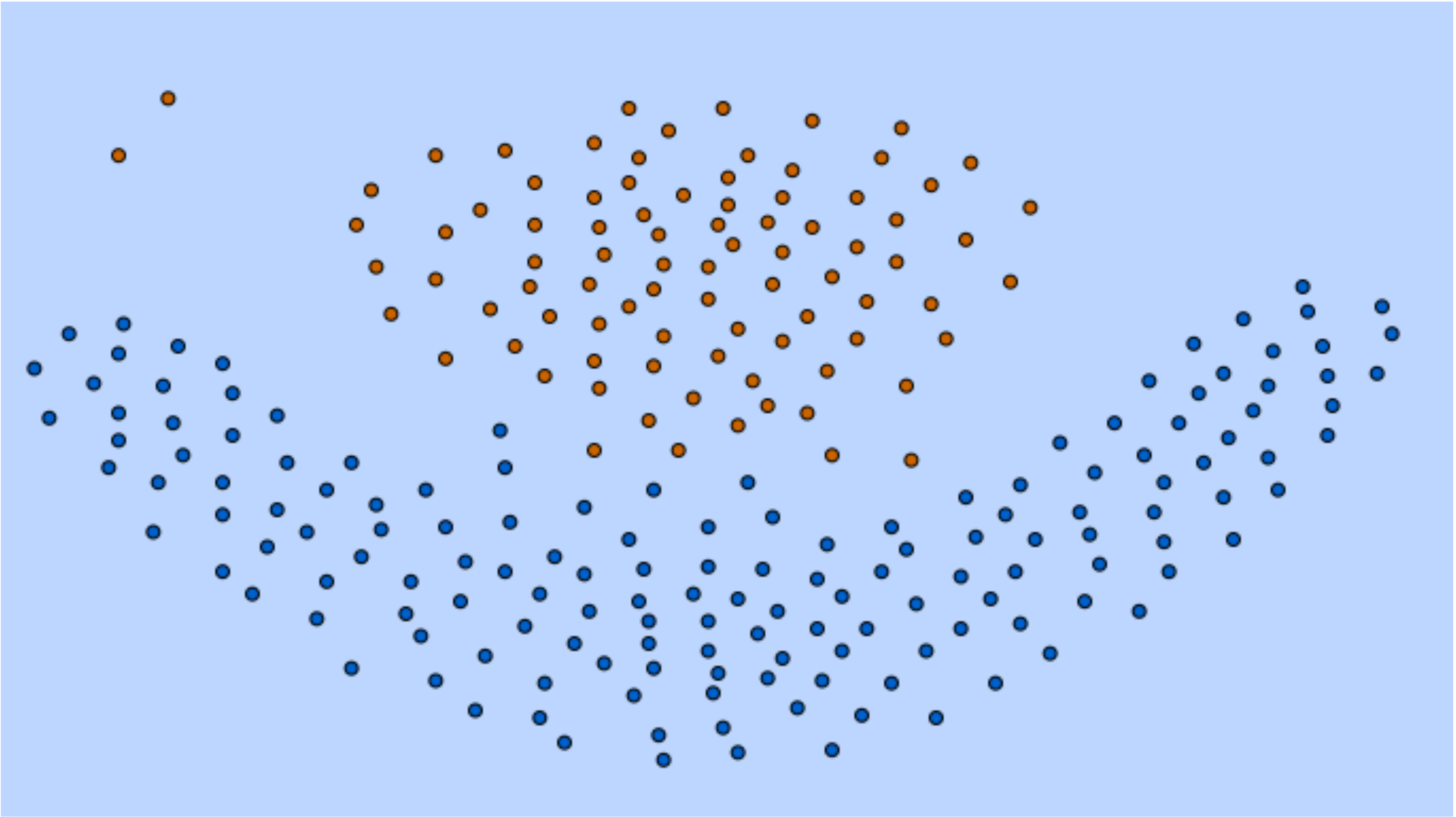
- In this lecture we will cover the linear kernel classifier that forms the basis for more advanced kernel methods classifiers,
- … which in turn is an essential part of the very advanced and powerful classifier *Support-Vector Machine* (SVM)

- We will use the Flame dataset as example in this lecture:

Flame dataset

- Generated dataset with two numerical attributes (x and y) and two categories (0 and 1)
- 240 examples
- Toy problem, not a real-world dataset

Flame dataset



Flame dataset

	A	B	C
1	x	y	class
2	0,12	0,27	0
3	0,09	0,31	0
4	0,09	0,43	1
5	0,06	0,43	1
6	0,03	0,46	1
7	0,04	0,49	1
8	0,07	0,47	1
9	0,09	0,45	1
10	0,13	0,44	1
11	0,16	0,45	1
12	0,12	0,47	1
13	0,17	0,47	1
14	0,20	0,49	1
15	0,13	0,49	1
16	0,09	0,49	1
17	0,09	0,50	1
18	0,08	0,52	1
19	0,12	0,53	1
20	0,13	0,51	1
21	0,17	0,50	1
22	0,11	0,57	1
23	0,16	0,53	1
24	0,20	0,52	1
25	0,25	0,52	1

240 examples

Linear Kernel Classifier

Linear Kernel Classifier

- The linear kernel classifier works like this:
  - Calculate a center point for each category by calculating the average of each attribute value, for all examples in that category
  - When classifying an example, the category of the closest center point is returned
  - Euclidean distance is commonly used as distance measure:

$$distance = \sqrt{(e_0 - C0_0)^2 + (e_1 - C0_1)^2}$$

Testing it

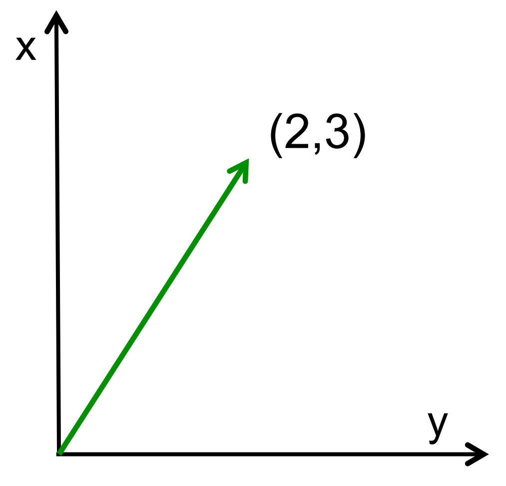
- We train and test the model on the Flame dataset
- Result:

Notifications | Output – WebIntA4 (run) | Java Call Hierarchy | Search Results

```
run:
Classifier: Basic Linear classifier (Euclidean distance)
Accuracy (whole dataset): 87.50%
Classifier: Basic Linear classifier (Euclidean distance)
Accuracy (10-fold CV): 85.83%
```

Dot-product

- We can use another measure of closeness based on *vectors* and *dot-products*
- A vector consists of a magnitude and direction, and is usually drawn as an arrow in a plane:



Dot-product

- The vector is defined by its ending point: x = 2 and y = 3
- Vectors in 3D space then consists of an x, y and a z value
- The dot-product is a single numerical value calculated as the sum of the products between each value in the first vector and the corresponding value in the second vector:

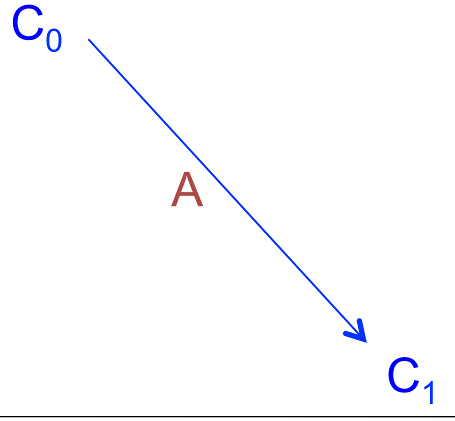
$$dot = v0_0 * v1_0 + v0_1 * v1_1 + ... + v0_n * v1_n$$

Meaning of the dot-product

- The dot-product is equal to the length of the two vectors multiplied together, multiplied by the cosine of the angle between the two vectors
- This has an important implication:
  - If the angle is greater than 90 degrees, the dot-product will be negative
  - If the angle is between 0 to 90 degrees, the dot-product will be positive
- How can this be used to calculate closeness?

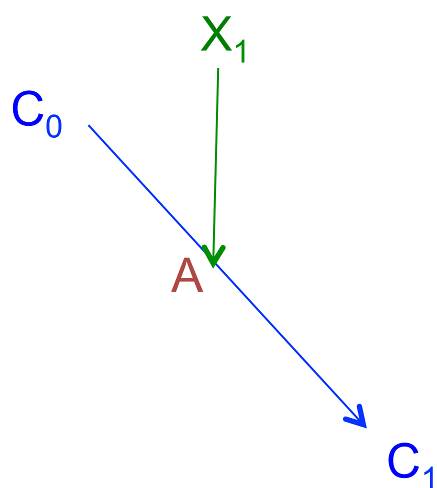
Closeness using dot-product

- Assume we have two center points C_0 and C_1
- We define a vector C_0C_1 as the vector between C_0 and C_1
- We calculate A as the middle point between C_0 and C_1 by calculating $(C_1 - C_0) / 2$



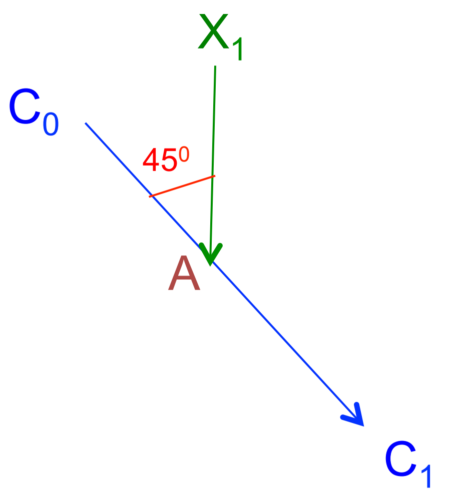
Closeness using dot-product

- We want to classify an example X_1 located as shown in the figure
- We define a vector X_1A going from X_1 to A



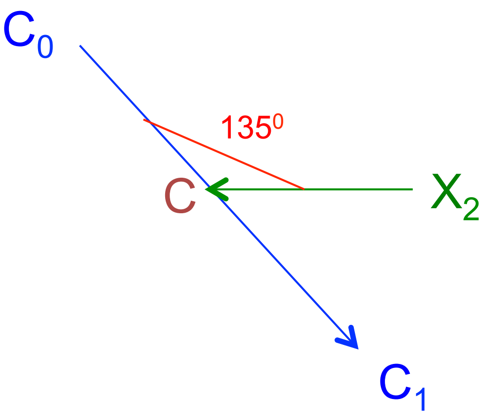
Closeness using dot-product

- Assume that the angle between the vectors C_0C_1 and X_1A is 45 degrees
- This is less than 90 degrees, therefore the dot-product is positive
- The sign tells us that X_1 is closer to C_0 than C_1



Closeness using dot-product

- If we have another example X_2 located as shown in the figure, assume the angle between C_0C_1 and X_2A is 135 degrees
- This is more than 90 degrees, therefore the dot-product is negative
- The sign tells us that X_2 is closer to C_1 than C_0



Closeness using dot-product

- The formula for finding the category is:

  $category = sign[\ (X – A) \bullet (C_1 – C_0)\ ]$

- A is calculated as $(C_1 - C_0) / 2$

  $category = sign[\ (X – (C_1 − C_0) / 2) \bullet (C_1 – C_0)\ ]$

- This can be simplified to:

  $category = sign[\ (X \bullet C_0 – X \bullet C_1 + (C_1 \bullet C_1 – C_0 \bullet C_0) / 2\ ]$

Testing it

- We train and test the model on the Flame dataset
- Dot-product is used for closeness instead of Euclidean distance
- Result:

Notifications | Output – WebIntA4 (run) | Java Call Hierarchy | Search Results | Us

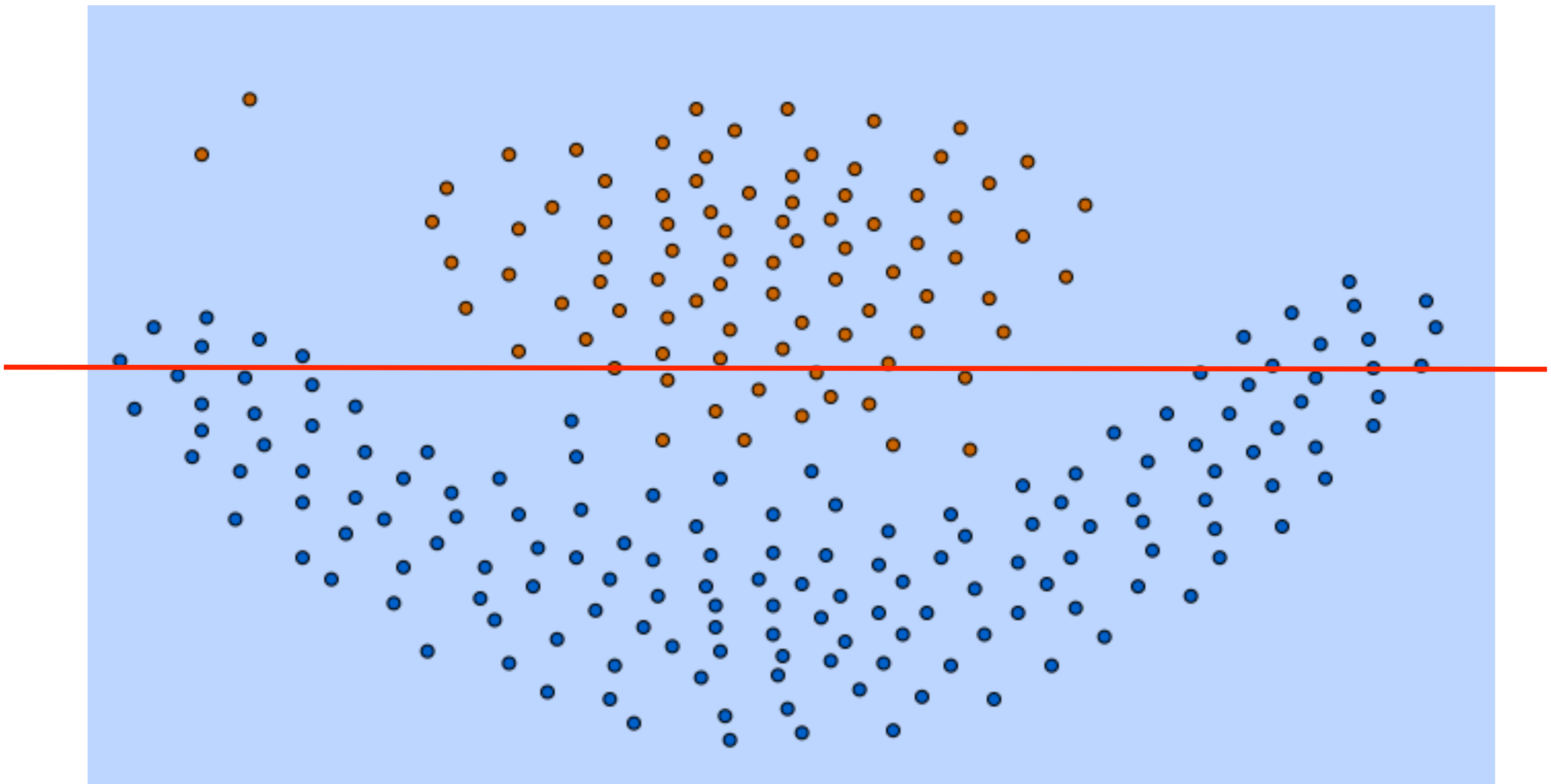
```
run:
Classifier: Basic Linear classifier (dot-product distance)
Accuracy (whole dataset): 87.50%
Classifier: Basic Linear classifier (dot-product distance)
Accuracy (10-fold CV): 85.83%
```

Actually equal to Euclidean

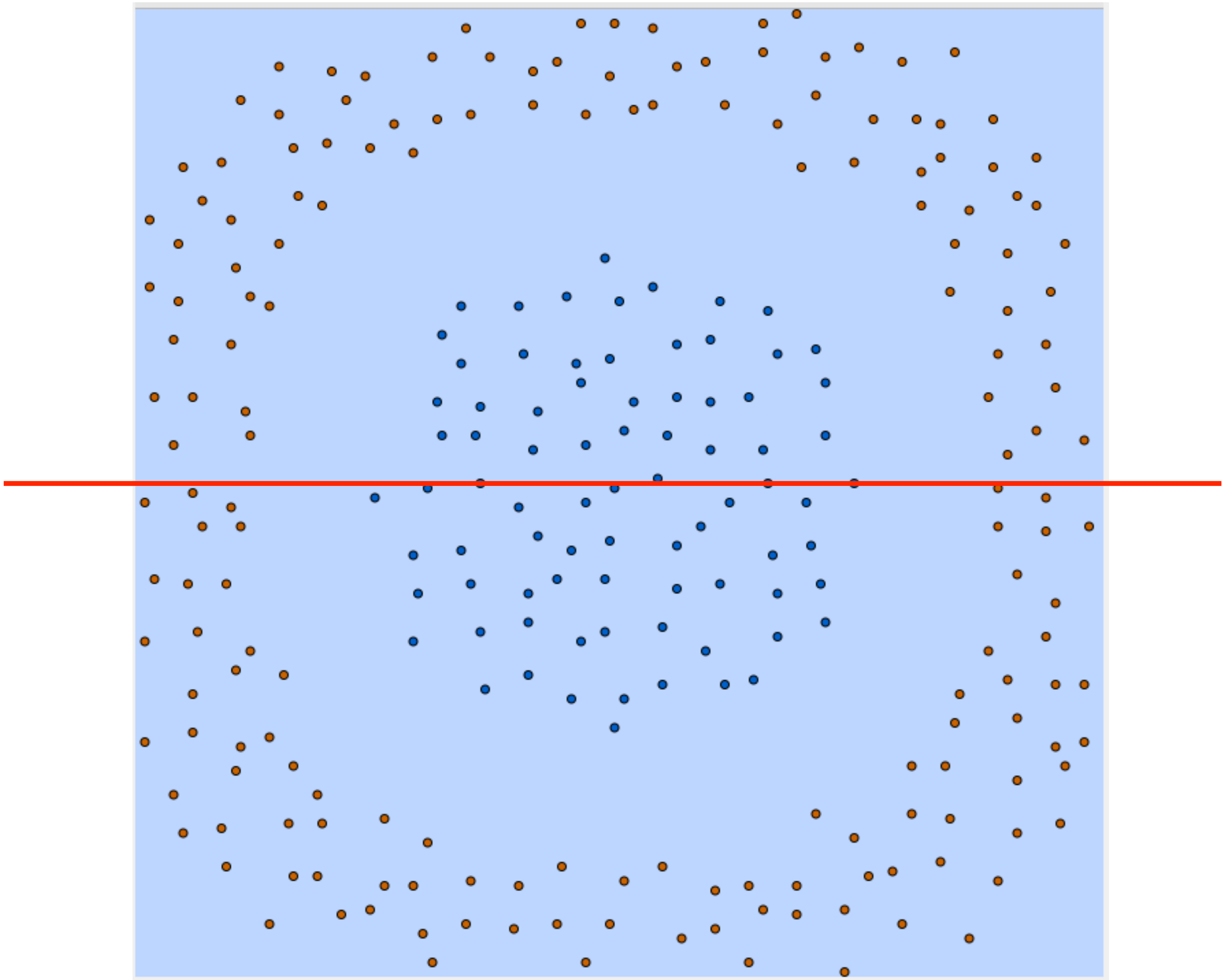
Notes on the result

- Even if we tested on the same data as we trained the classifier, the accuracy was rather low: 87.50%
- This is because the classifier only finds a dividing line between the two categories
- If there isn't a straight line divided the categories, the classifier will not be very accurate

Almost linearly separable

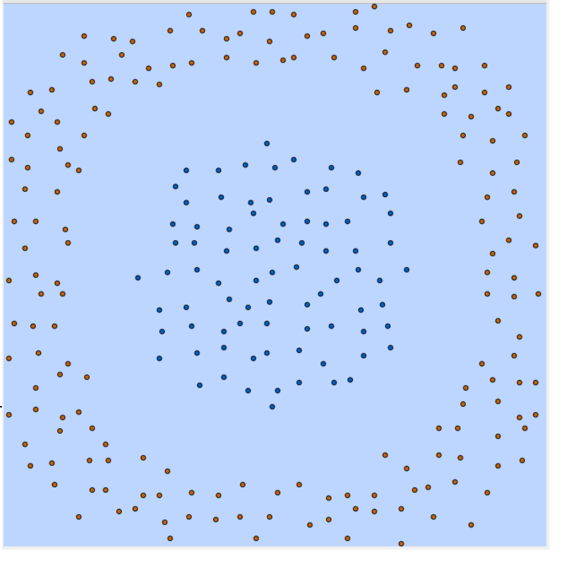


Not linearly separable



Bad linear separation

- Where would the average points be for each category?
- It turns out that they will be placed at almost the exact same location
- A linear classifier is therefore unable to distinguish between the two categories

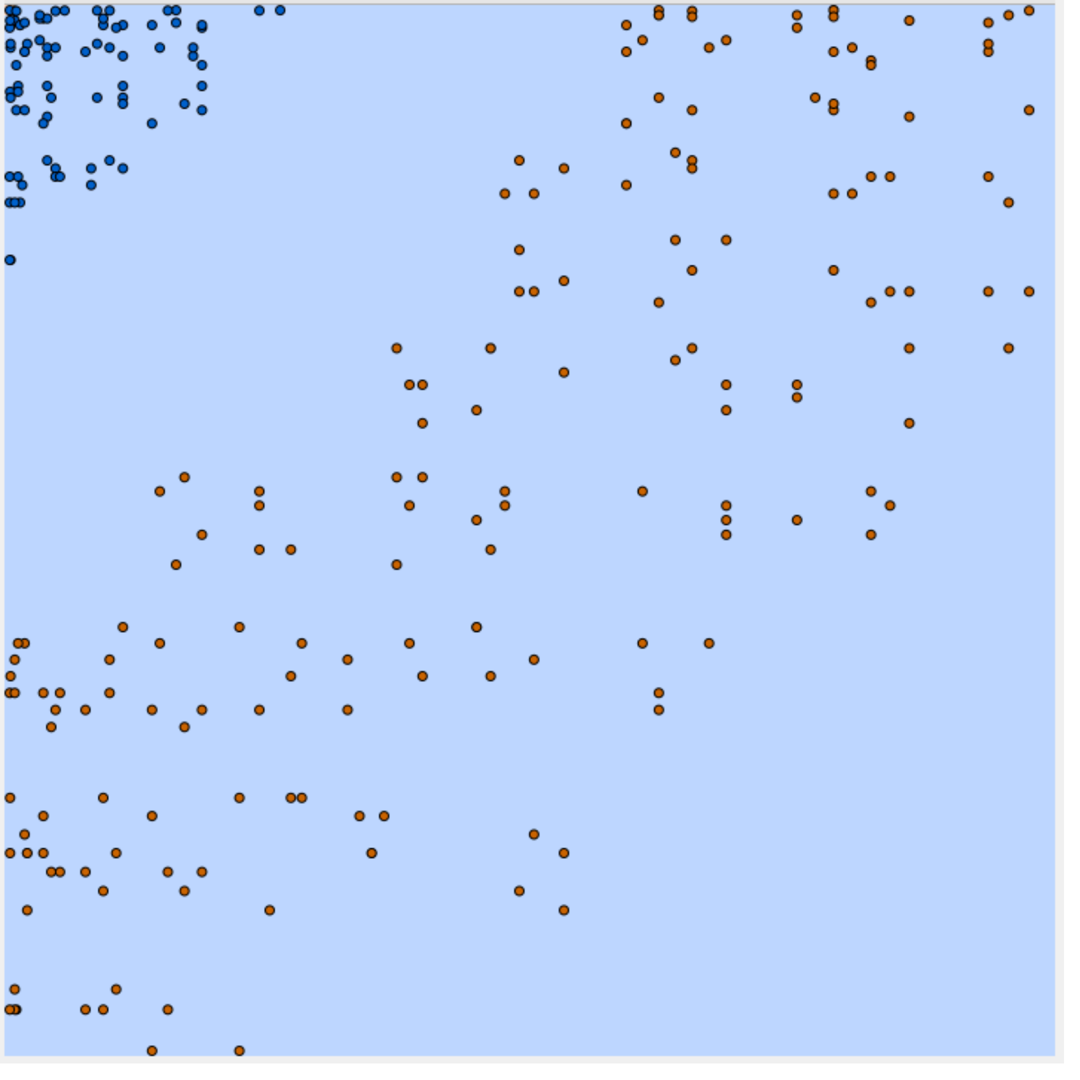


Kernel Classifier

Data transformation

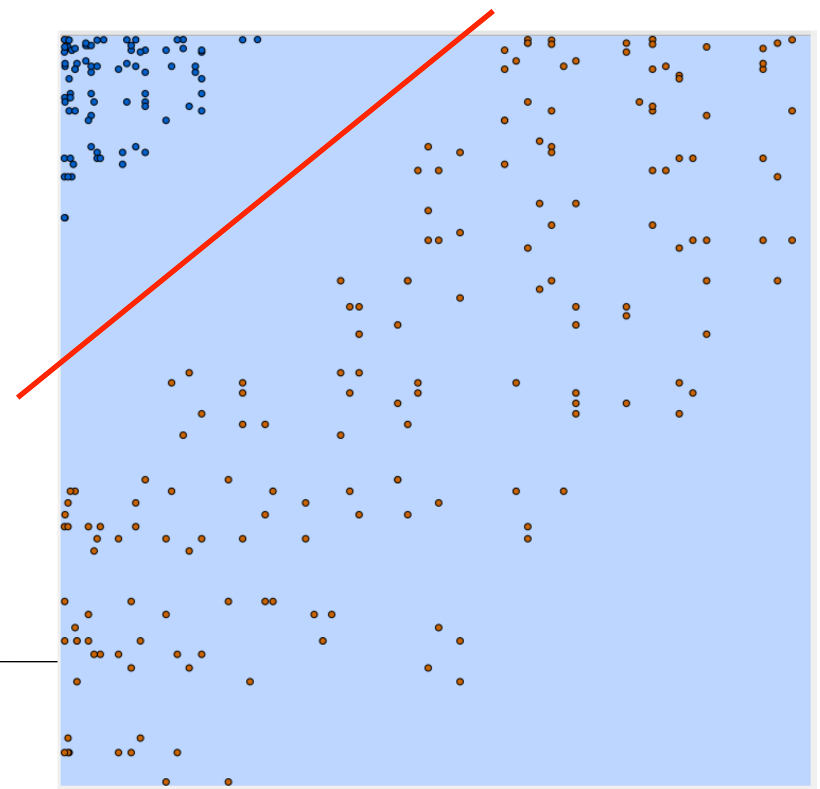
- Let's see what happens if we square every *x* and *y* value
- A point at (-1, 2) in XY-space will now be at (1, 4) in X^2Y^2-space
- If we do this for all data points and plot them again, the result will look like:

Data transformation



Data transformation

- All examples belonging to one category has now moved to the lower left corner
- It is now possible to divide the categories with a straight line!



Data transformation

- So, if we can find a transformation to a space where the data can be divided by a straight line we can use the linear classifier on the transformed data
- The problem is that in many real-world datasets it can be very difficult to find the right transformation
- Simply calculating the square of each value doesn't work for all datasets
- The classifier must find the unique transformation for each dataset!

The Kernel Trick

- Searching for the right transformation is not possible
- There are an endless number of possible transformations, and testing them all takes too long time
- Luckily we have something called the *kernel trick*, which works on any algorithm that uses dot-products for closeness
- This includes our linear classifier!

The Kernel Trick

- We can replace the dot-product function with a new function,
- … that returns what the dot-product would have been if the data had first been transformed to a higher dimensional space
- In practice there are only a few transformations used
- The probably most common one is the *radial-basis function*

Radial-basis function

- The radial-basis function is similar to the dot-product in that it takes two vectors as in parameters and returns a value
- It is however not linear, and therefore can divide more complex spaces
- The RBF function looks like this:

$$rbf = e^{-\gamma \cdot \sum_0^n (v1_i - v2_i)^2}$$

The gamma parameter can be adjusted to get the best separation for a data set

RBF in code

```
double RBF (Instance i1, Instance i2, double gamma)
  //Find squared distance between i1 and i2
  double sq_dist = 0
  for (int a : numAttributes)
    sq_dist += pow(i1[a] – i2[a], 2)

  //Calculate RBF value
  double rbf = pow(E, -gamma * sq_dist)

  return rbf
```

The Kernel Trick

- Now we need a function that calculates the distances from the average points in the transformed space
- We can’t do this, since we don’t know the locations of the points in the transformed space
- This is where the kernel tricks comes in:
  - Averaging a set of vectors and taking the dot-product of the average with vector A
  - … gives the same result as:
  - Averaging the dot-products of vector A with every vector in the set

The Kernel Trick

- So, instead of calculating the dot-product between example X and the average for a category,
- … we can calculate the radial-basis function between X and every other example belonging to the category,
- … and then average the result

The algorithm

```
int classify (Instance i)
  //Define variables
  float sum0, sum1, count0, count1

  //Iterate over all training instances
  //and calculate RBF values
  for (Instance t : trainingset)
    if (t.category == C0)
      sum0 += RBF(i, t, gamma)
      count0++
    if (t.category == C1)
      sum1 += RBF(i, t, gamma)
      count1++

  //Calculate y-value
  y = (1/count0)*sum0 - (1/count1)*sum1 + offset

  //Check sign for result
  if (y > 0) return C0
  else return C1
```

The algorithm in code

- The algorithm uses an *offset* value.
- Calculating this is quite time consuming,
- … so we should calculate it once during the training step and feed it to the classify step each time we want to classify a new example
- The code for doing this looks like:

Calculate offset

```
float calc_offset ()
  //Define lists
  List<Instance> l0, l1

  //Divide the training dataset for each class
  for (Instance t : trainingset)
    if (t.category == C0)
      l0.add(t)
    if (t.category == C1)
      l1.add(t)

  //Define variables
  float sum0, sum1

  //Calculate RBF values for C0
  for (Instance i01 : l0)
    for (Instance i02 : l0)
      sum0 += RBF(i01, i02, gamma)

  //Calculate RBF values for C1
  for (Instance i11 : l1)
    for (Instance i12 : l1)
      sum1 += RBF(i11, i12, gamma)

  //Calculate offset
  offset = (1/(l1.size^2))*sum1 – (1/(l0.size^2))*sum0
```

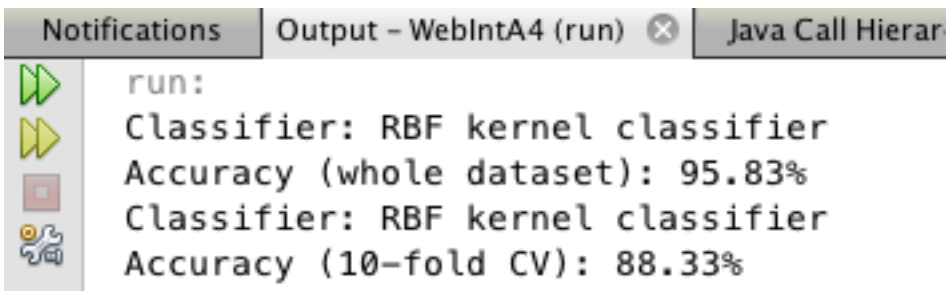
Non-linear Kernel Classifier

- The result is a non-linear kernel classifier
- It can divide categories that are not linearly separable

- So, how good is it?

Testing it

- We train and test the RBF classifier on the Flame dataset
- Result:



Better than before!

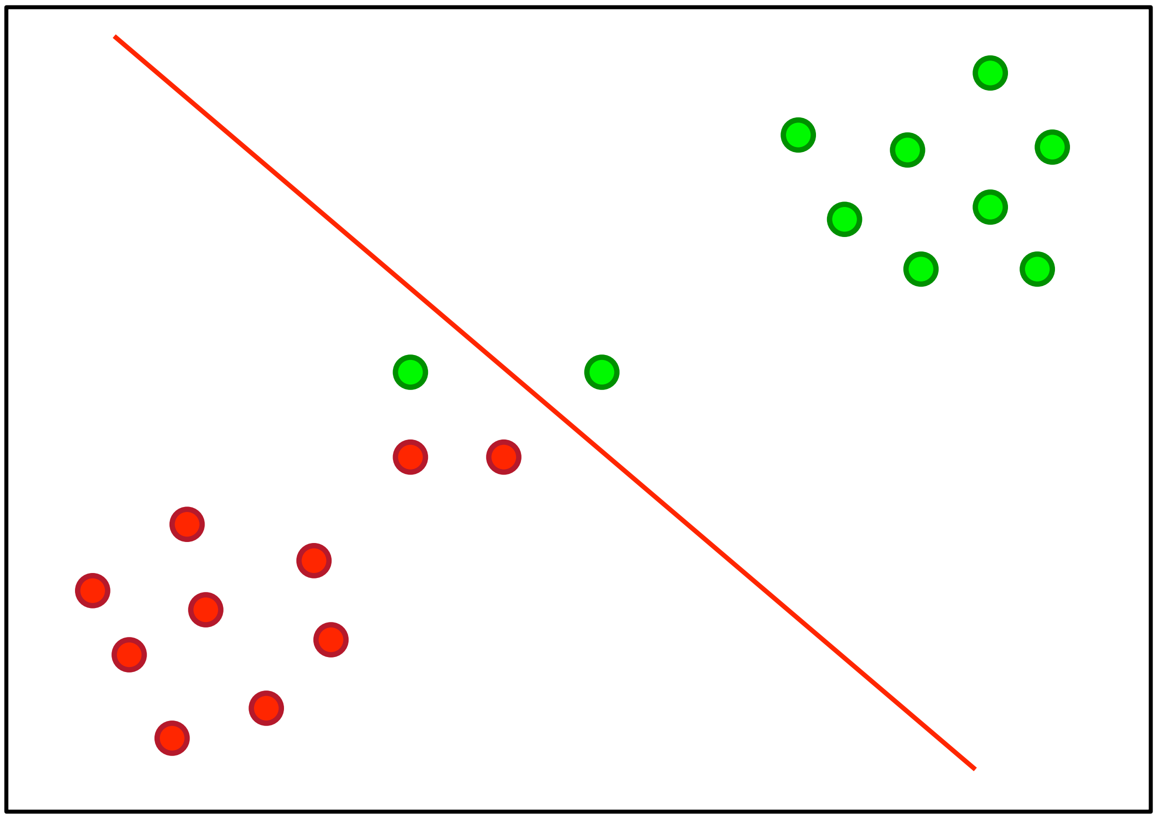
Multiclass RBF classification

- Still uses binary classification (two categories)
- The multiclass problem is reduced to a number of multiple binary classification problems
- We need a strategy to decide which binary combination that "wins"
- We will not dig further into this in this lecture

Support Vector Machines

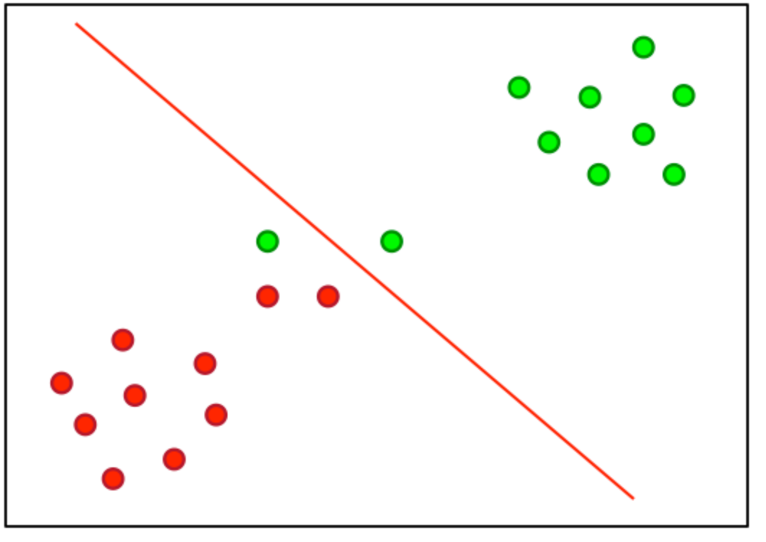
Support-Vector Machine

- Consider the following data:



Support-Vector Machines

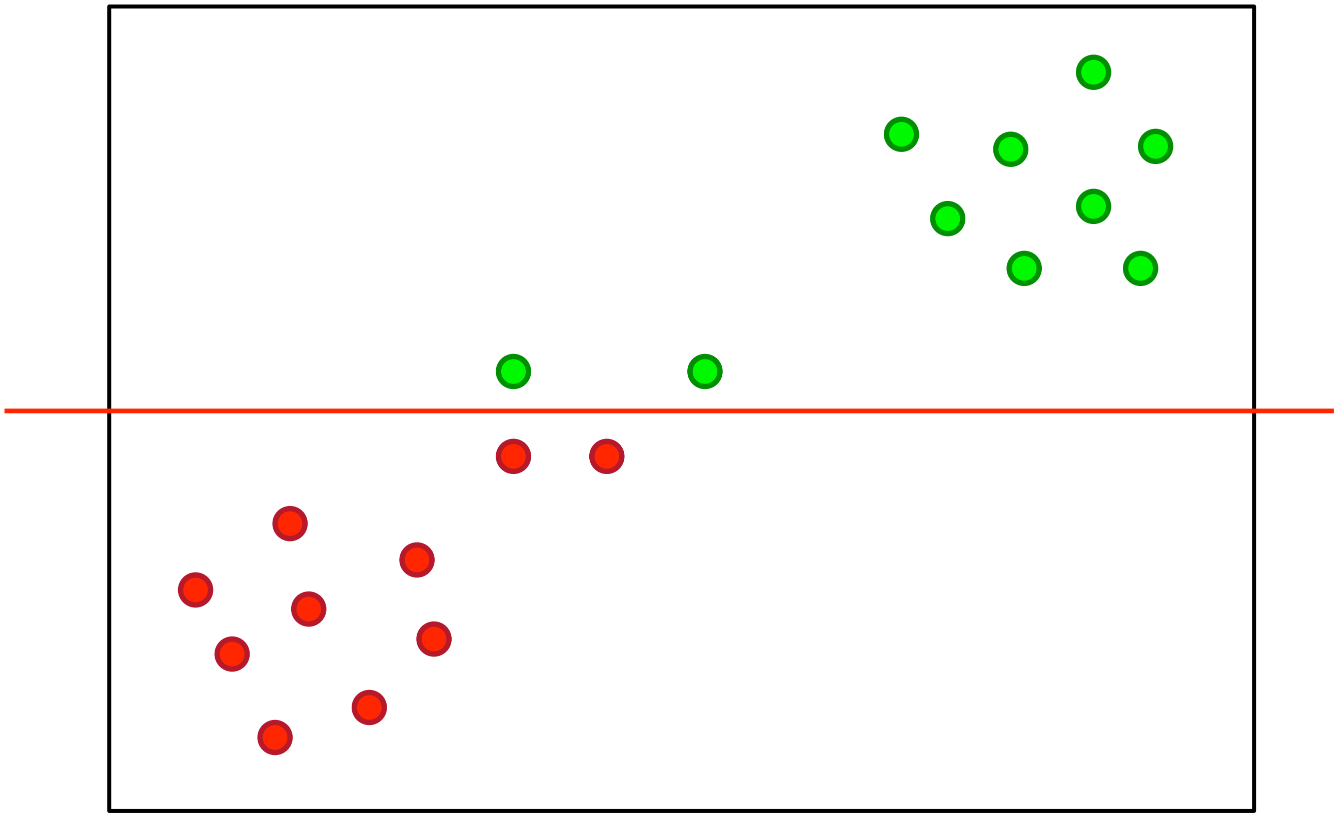
- The line is the dividing line using averages of categories
- One example is misclassified since it is on the wrong side of the dividing line
- In this example, most examples are far away from the line and is therefore not relevant for classification



Support-Vector Machines

- This is a problem for both a linear or kernel method classifier
- To solve this, we must use a Support-Vector Machine
- The work by finding the line that is as far away as possible from each of the categories
- This line is called the *maximum-margin hyperplane*:

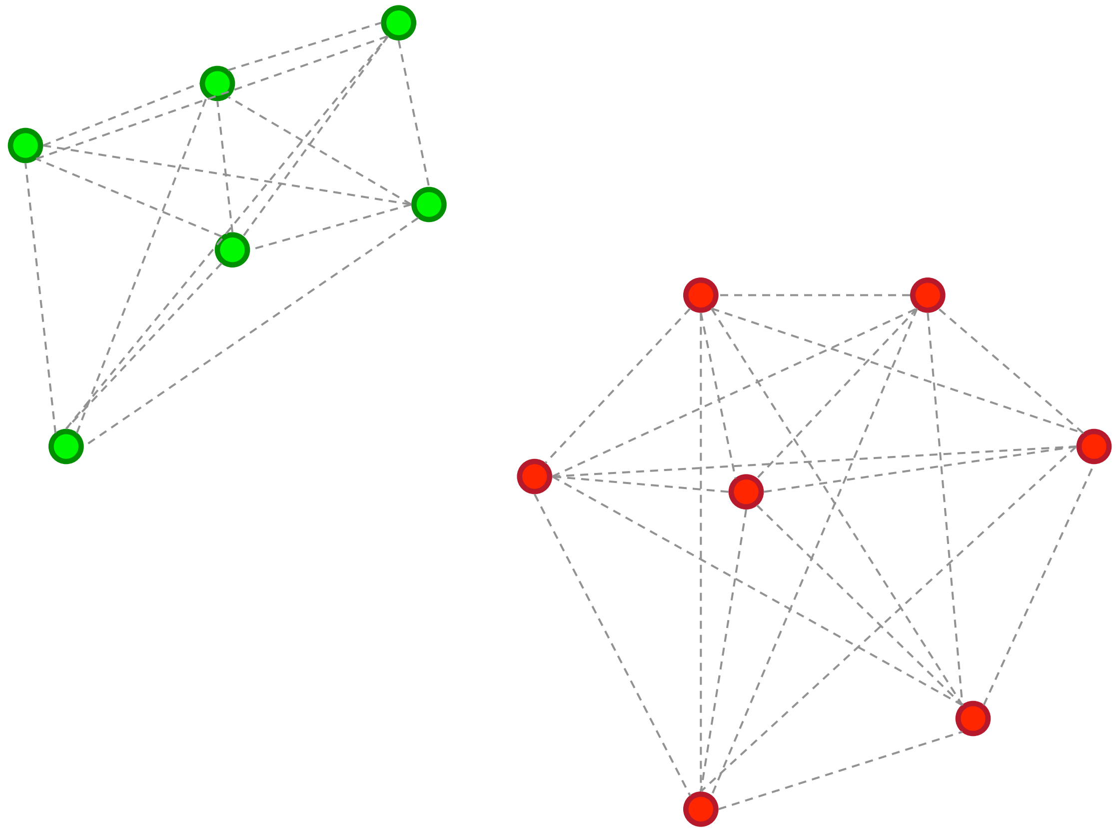
Maximum-margin hyperplane



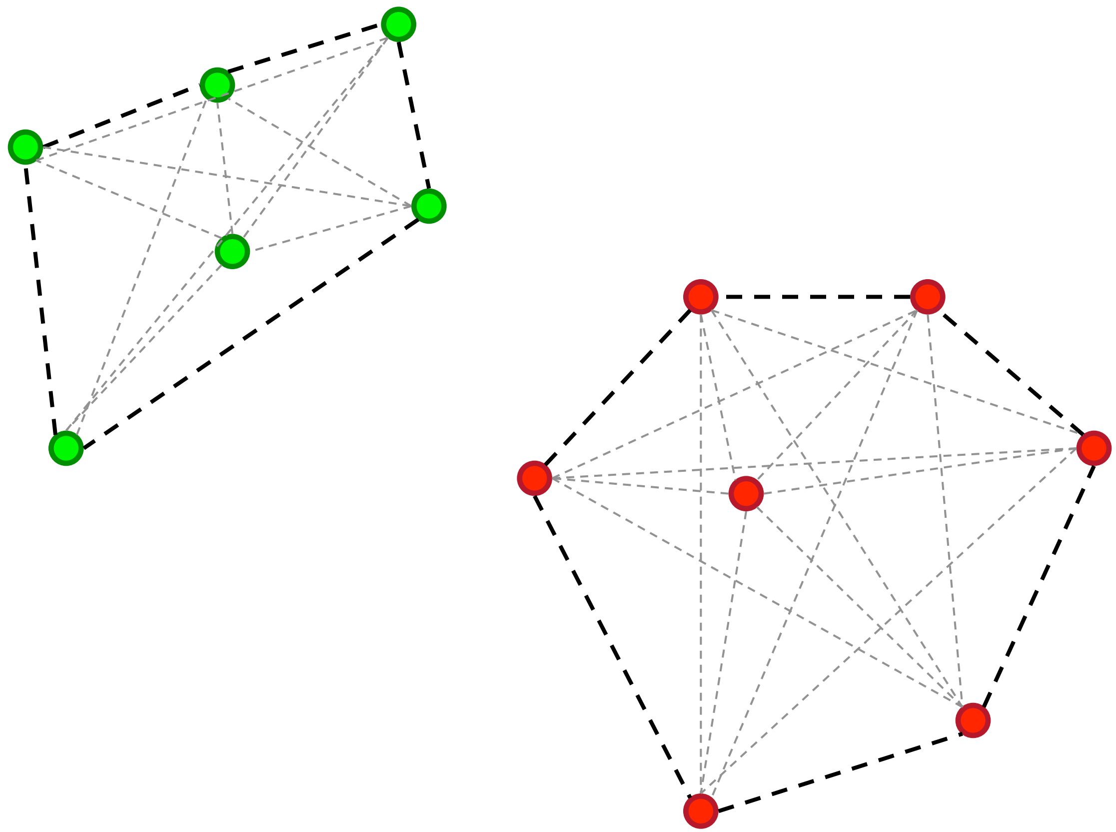
Finding the Maximum-margin hyperplane

- Conceptually, finding the maximum-margin hyperplane is done by:
  - Draw imaginary lines between all examples of a category
  - Repeat for all categories
  - The outer lines are called the convex hull
  - It is defined as the tightest polygon enclosing the examples in a category
  - The hyperplane is placed exactly between the convex hulls of the two categories

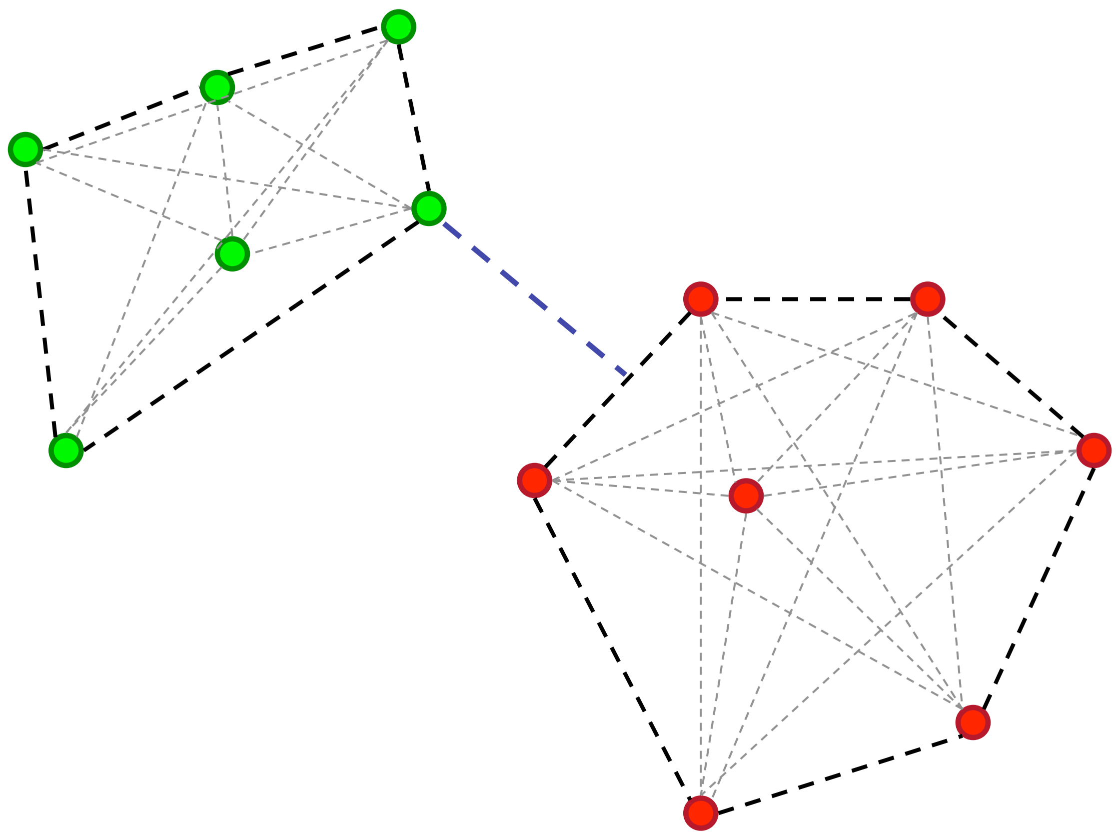
Draw imaginary lines



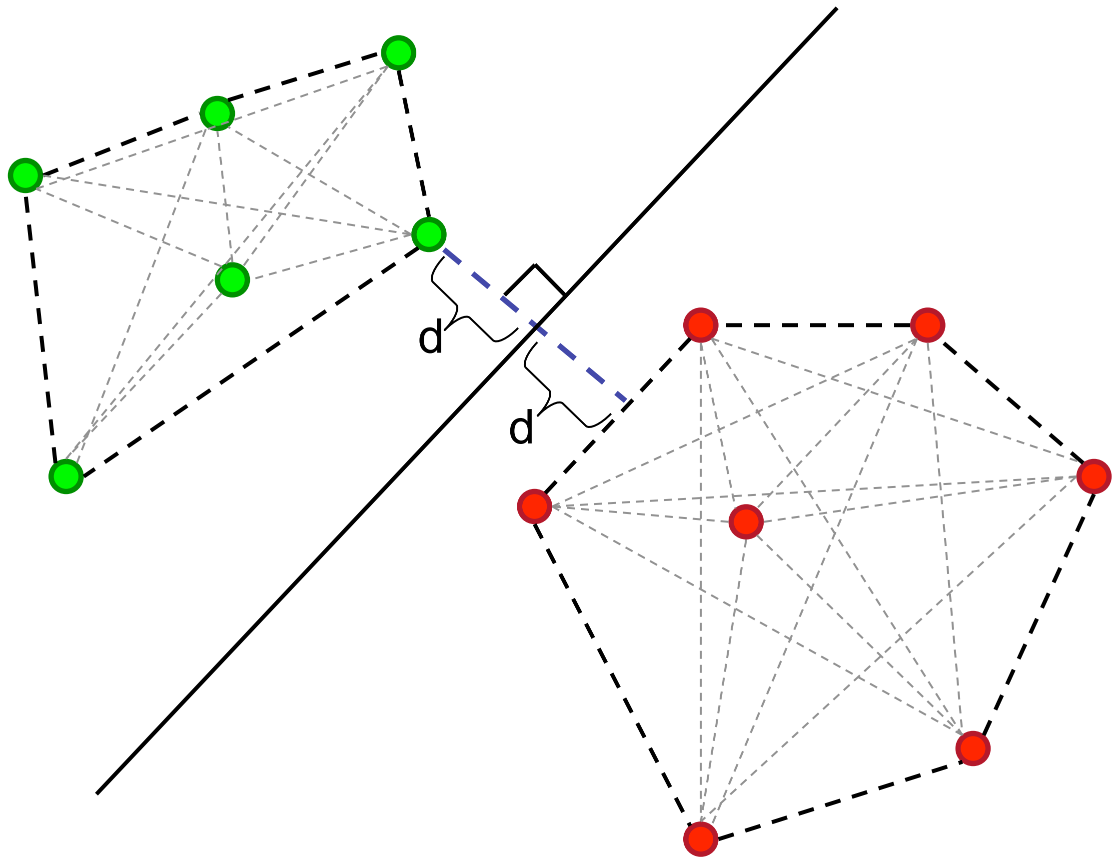
Find the convex hulls



Find the shortest line between the hulls



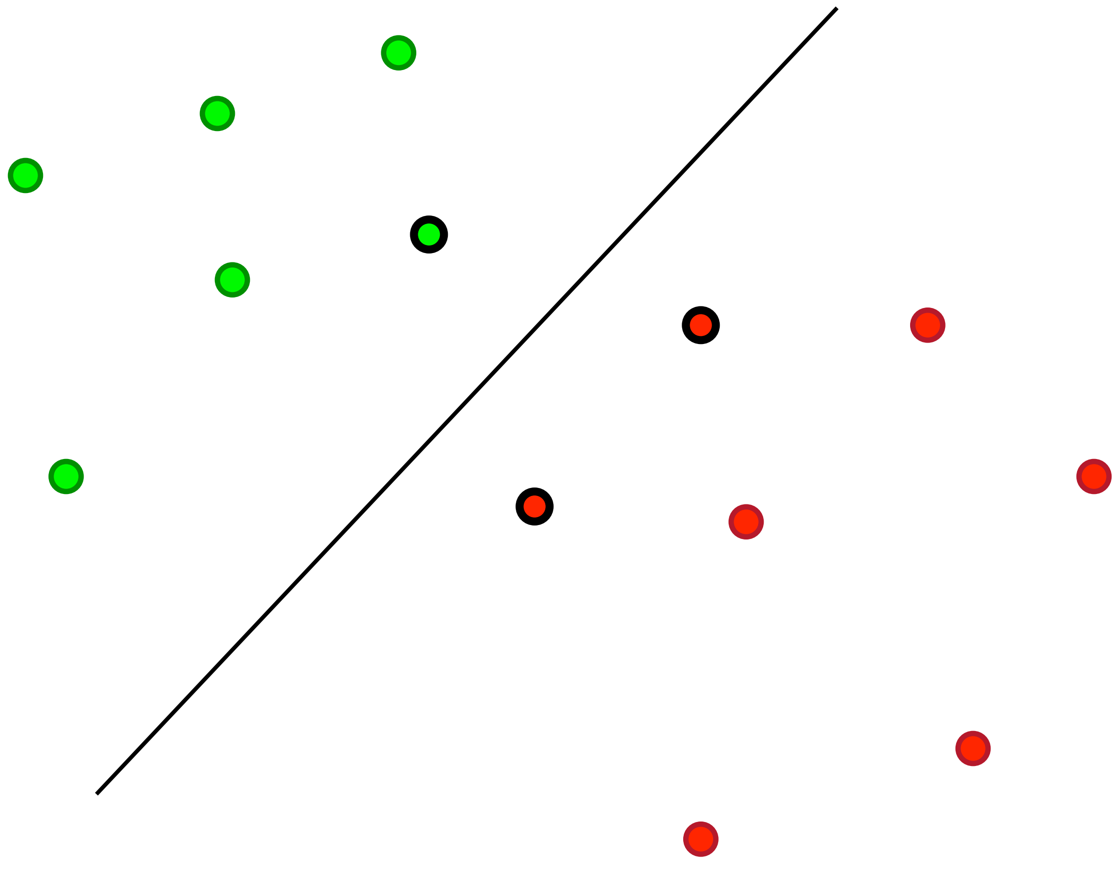
Place the hyperplane between the hulls



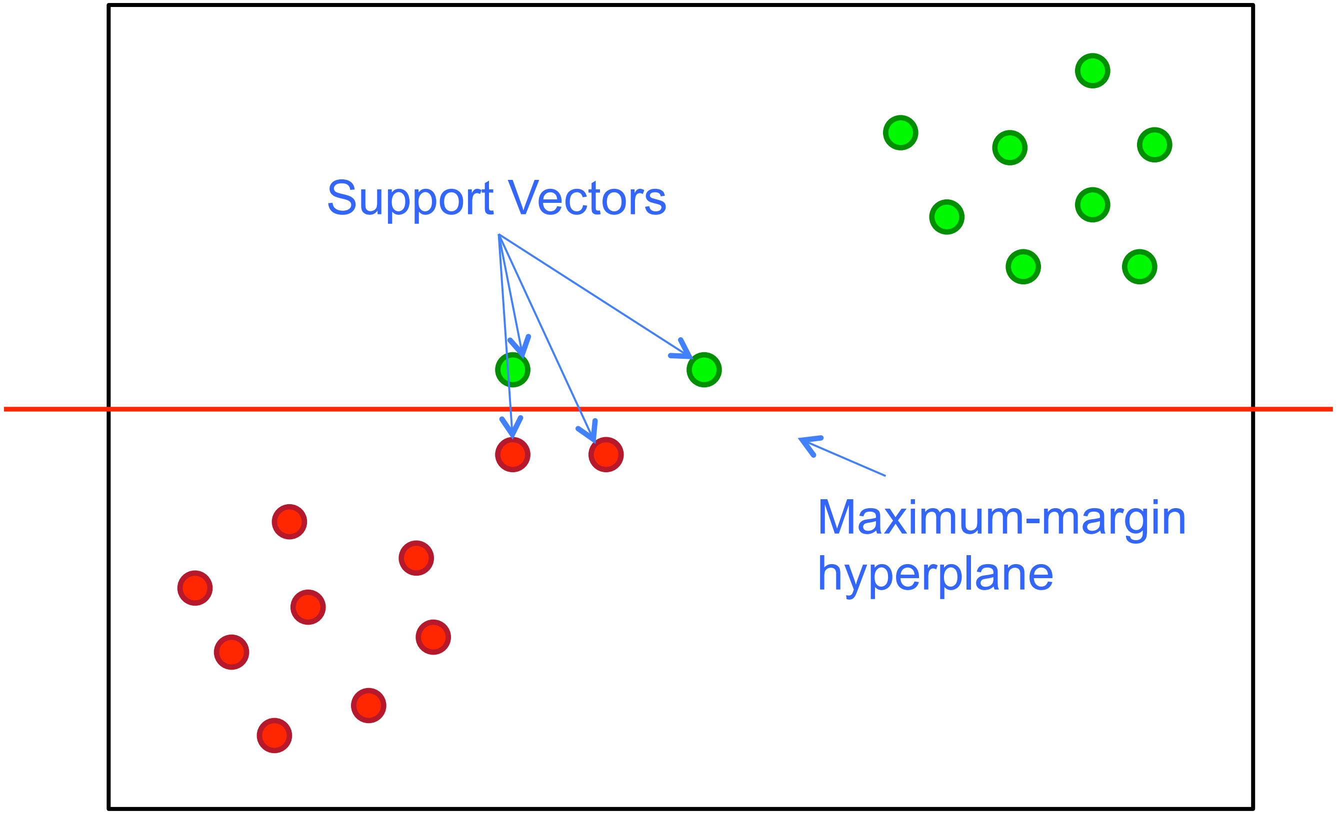
Support Vectors

- As can be seen in the figure, we don't need all examples to define the hyperplane
- We only need the closest examples for each category
- These are called the Support Vectors:

Support Vectors



Back to the example



Support Vector Machines

- Algorithms for finding the maximum-margin hyperplane are very complex
- In this course, we will learn how to use a very common library for Support Vector Machines:
  - libsvm
  - https://github.com/cjlin1/libsvm

Using libsvm

- The first thing to do in the training step is to convert the dataset to the data structures used by libsvm:

```
//Convert data set to LibSVM data structures.
//Data is added as svm_node objects in a svm_problem object.
int n = data.noInstances();
svm_problem prob = new svm_problem();
prob.y = new double[n];
prob.l = n;
prob.x = new svm_node[n][data.noAttributes() - 1];

for (int i = 0; i < data.noInstances(); i++)
{
    Instance inst = data.getInstance(i);

    //Attributes
    double[] vals = inst.getAttributeArrayNumerical();
    prob.x[i] = new svm_node[data.noAttributes() - 1];

    for (int a = 0; a < data.noAttributes() - 1; a++)
    {
        svm_node node = new svm_node();
        node.index = a;
        node.value = vals[a];
        prob.x[i][a] = node;
    }

    prob.y[i] = inst.getClassAttribute().numericalValue();
}
```

Using libsvm

- After converting the data, training the model is simple:

```
//Defines SVM parameters
//If these are incorrect, the classifier will give
//bad results
svm_parameter param = new svm_parameter();
param.probability = 1;
param.gamma = 10.0;
param.nu = 0.5;
param.C = 100;
param.svm_type = svm_parameter.C_SVC;
param.kernel_type = svm_parameter.RBF;
param.cache_size = 20000;
param.eps = 0.001;
```

Using libsvm

- Classifying an example also involves some data conversion:

```
//Convert instance to value array
double[] vals = i.getAttributeArrayNumerical();
int no_classes = data.noClassValues();

//Convert the instance to libsvm data structures
svm_node[] nodes = new svm_node[vals.length];
for (int a = 0; a < vals.length; a++)
{
    svm_node node = new svm_node();
    node.index = a;
    node.value = vals[a];
    nodes[a] = node;
}
```

Using libsvm

- Classifying the examples is then simple:

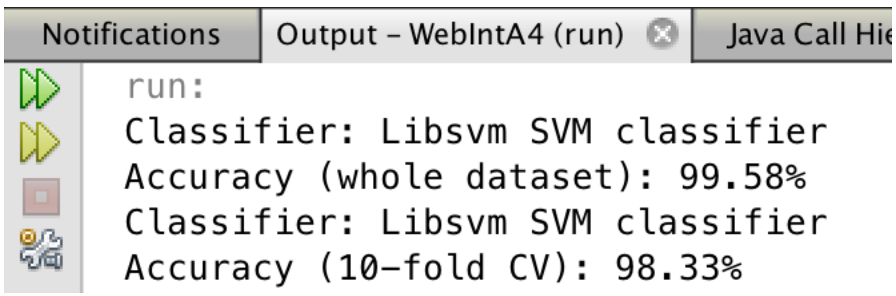
```java
//Define some libsvm stuff
int[] labels = new int[no_classes];
svm.svm_get_labels(model,labels);
double[] prob_estimates = new double[no_classes];

//Classify the instance
double cVal = svm.svm_predict_probability(model, nodes, prob_estimates);

//Return predicted class value
return new Result(cVal);
```

Testing it

- We train and test the model on the Flame dataset
- Result:



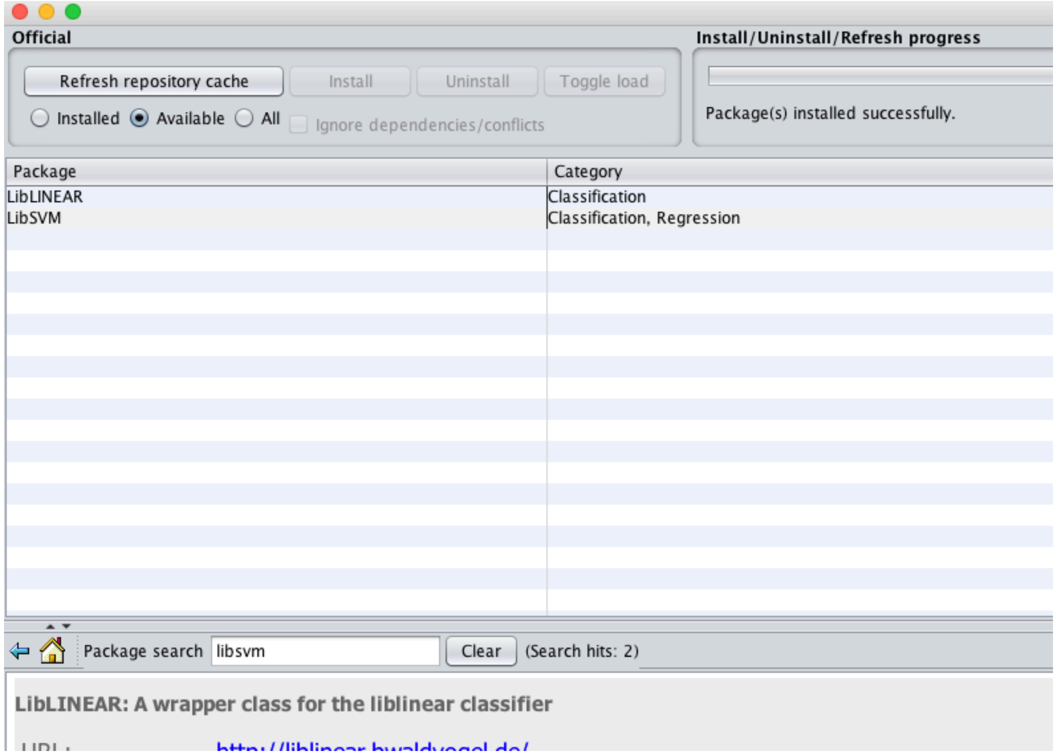
Best result!

When to use SVMs

- Support Vector Machines are very powerful classifiers which have successfully been used for a number of complex tasks:
  - Classifying facial expressions
  - Detecting intruders using datasets from the military
  - Predicting the structure of proteins from their DNA sequences
  - Handwriting recognition
- Finding good parameters can however be tricky, and using wrong parameters can result in very bad accuracy
- Which parameters to use depends on the dataset

Weka

- Weka uses libsvm for its SVM classifier
- The library is not included in the Weka package, so you need to install it in the package manager



Weka result

Classifier output

Correctly Classified Instances	239	99.5833 %
Incorrectly Classified Instances	1	0.4167 %
Kappa statistic	0.991	
Mean absolute error	0.023	
Root mean squared error	0.0775	
Relative absolute error	4.9667 %	
Root relative squared error	16.1147 %	
Total Number of Instances	240	

R

- R also supports SVM
- It is part of the machine learning package Caret
- R uses csv format (comma separated values) with or without header

R script

```
#Load the ML library
library(caret)

#Read the dataset
dataset <- read.csv(”flame.csv")

#setup 10-fold cross validation
control <- trainControl(method="cv", number=10)
metric <- "Accuracy"

#Train model
set.seed(7)
svm <- train(class~., data=dataset, method="svmRadial",
               metric=metric, trControl=control)

#Print result
print(svm)
```

R result

```
Support Vector Machines with Radial Basis Function Kernel

240 samples
  2 predictor
  2 classes: 'C0', 'C1'

No pre-processing
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 216, 216, 216, 216, 217, 216, ...
Resampling results across tuning parameters:

  C     Accuracy   Kappa
  0.25  0.9958333  0.9909091
  0.50  0.9873188  0.9725064
  1.00  0.9873188  0.9725064
```

Kernel Methods and SVMs

Dr. Johan Hagelbäck

johan.hagelback@lnu.se

http://aiguy.org