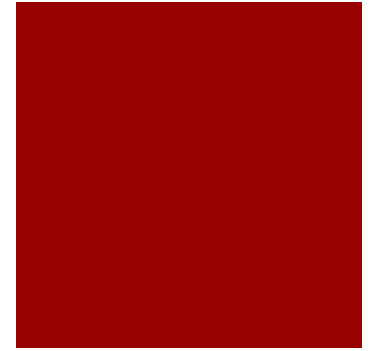


# A practical guide to Support Vector Machine and the Kernel Based Learning Platform (KeLP)

Danilo Croce  
University of Roma, Tor Vergata

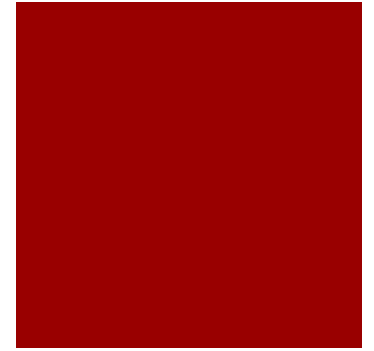
Deep Learning 2025

# Support Vector Machines



- Let us consider a binary classification problem where examples are represented in an “input” space  $X \subseteq \mathcal{R}^n$  and the output space  $Y = \{-1, 1\}$ 
  - Training set  $S = \{(x_1, y_1), \dots, (x_l, y_l)\} \in (X \times Y)$
- We want to derive a relation binding  $X$  and  $Y$
- In a classification problem, we want to induce a decision function  $f$  that, given a new example  $x$ , produces a label -1 or +1, depending on the assigned class

# Support Vector Machines



- A binary classifier is can be implemented considering the function
- $f: X \subseteq \mathcal{R}^n \rightarrow \mathcal{R} :$ 
  - if  $f(x) \geq 0$             +1
  - if  $f(x) < 0$             - 1
- And the final classification function is:
  - $\text{sign}(f(x))$

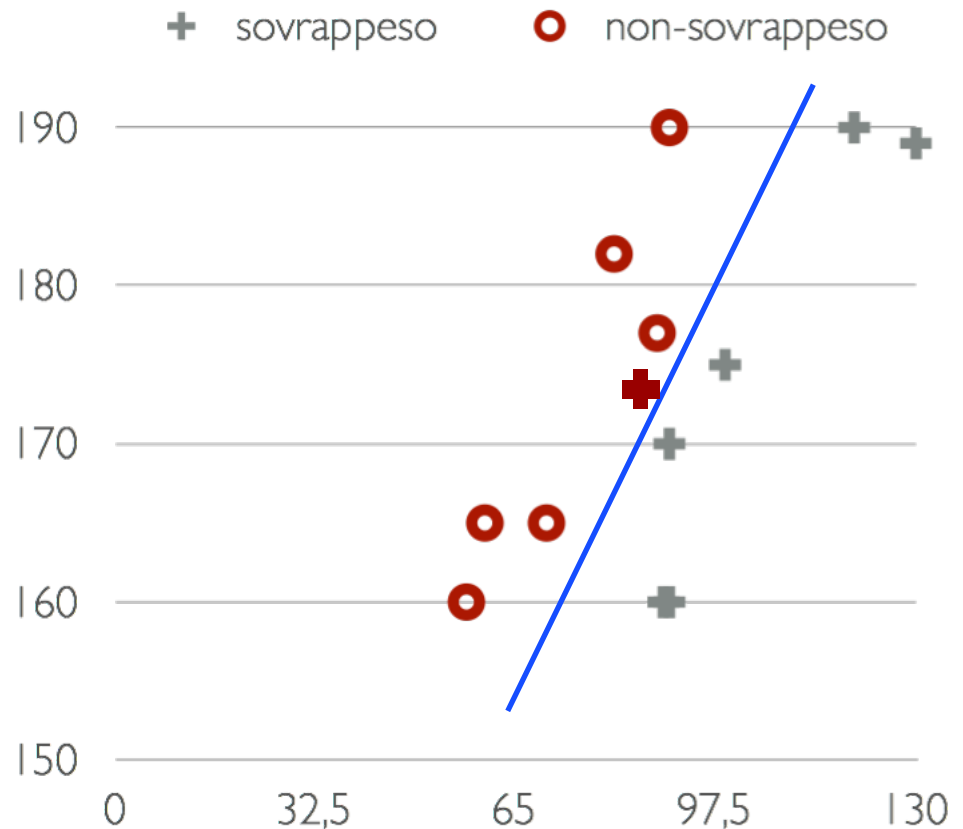
# SVM Optimization: trade-off between training error and margin



The soft-margin SVM allows classification errors in the training set

- The regularization parameter  $C$  need to appropriately chosen
- A very high value of  $C$  corresponds to the hard margin SVM

$$\begin{aligned} \text{Minimize:} \quad & \frac{1}{2} \|\vec{w}\|^2 + C \sum_k \xi_k \\ \text{subject to:} \quad & \forall k : y_k [\vec{w} \cdot \vec{x}_k + b] \geq 1 - \xi_k \end{aligned}$$



## SVM OPTIMIZATION: Adjusting the cost of false positives vs. false negatives

- As we can deal with unbalanced numbers of positive and negative examples, two different cost factors  $C_+$  and  $C_-$  are employed.

$$\begin{array}{ll} \text{Minimize:} & \frac{1}{2} \|\vec{w}\|^2 + C_+ \sum_{i:y_i=1} \xi_i + C_- \sum_{j:y_j=-1} \xi_j \\ \text{subject to:} & \forall k : y_k [\vec{w} \cdot \vec{x}_k + b] \geq 1 - \xi_k \end{array}$$

- They allow to adjust the cost of false positives vs. false negatives

# A practical example

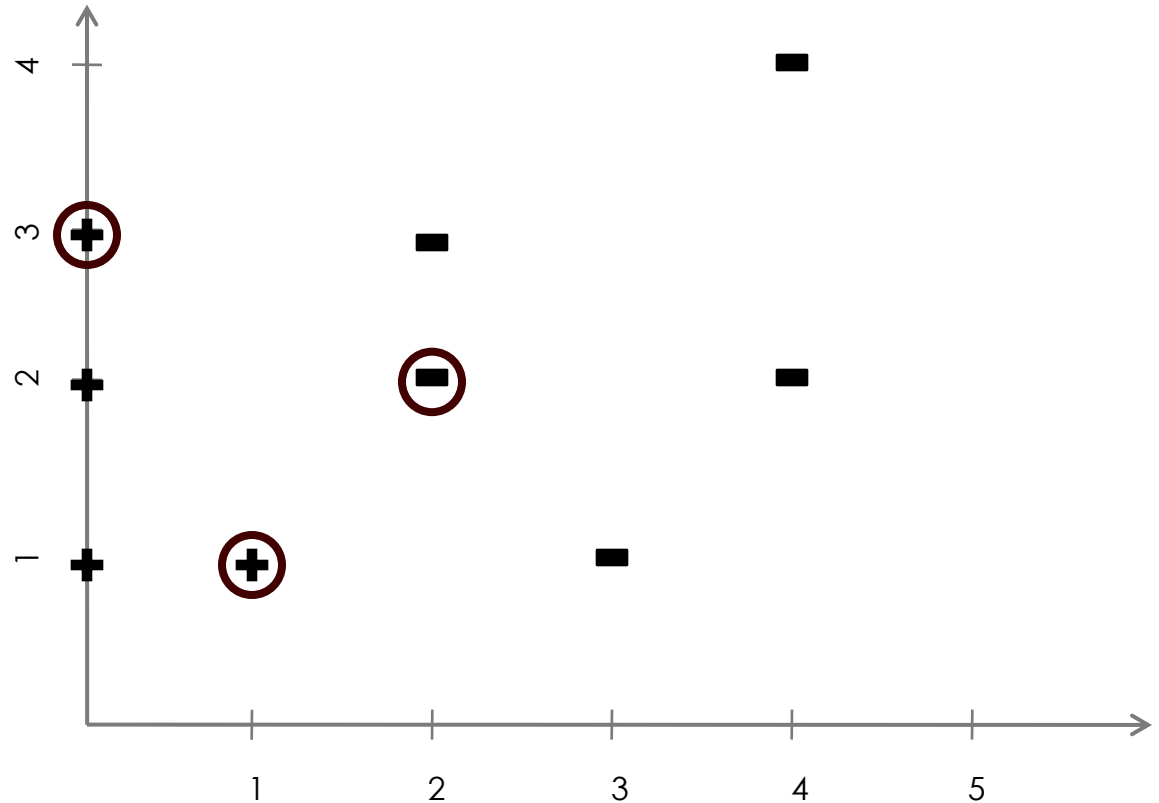


## Positive Ex.

a1 (1,1)  
a2 (0,1)  
a3 (0,2)  
a4 (0,3)

## Negative Ex.

b1 (3,1)  
b2 (2,2)  
b3 (2,3)  
b4 (4,2)  
b5 (4,4)



We apply the SVM learning algorithms



	$\alpha_i$	COORD
$sv_1$	0.222	(0,3)
$sv_2$	0.889	(1,1)
$sv_3$	-1.111	(2,2)
$b$	3	

# A practical example



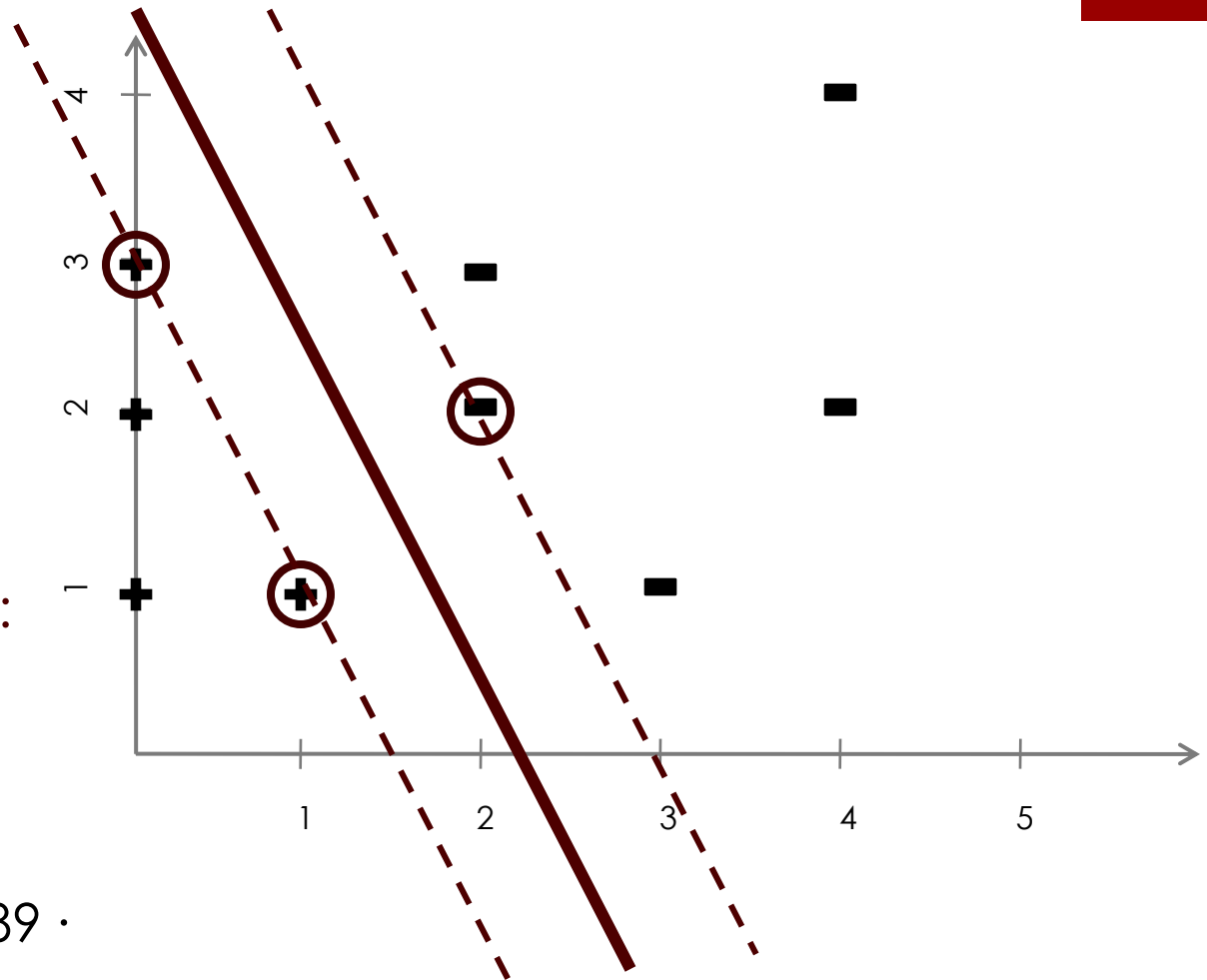
	$\alpha_i$	COORD
$sv_1$	0.222	(0, 3)
$sv_2$	0.889	(1, 1)
$sv_3$	-1.111	(2, 2)
<b>b</b>	3	

Hyperplane Equation:  
 $w \cdot x + b = 0$

$$w = \sum \alpha_i \cdot sv_i =$$

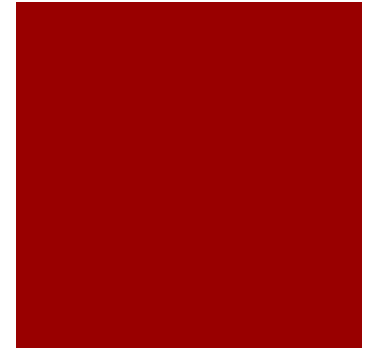
$$w = 0.222 \cdot (0, 3) + 0.889 \cdot (1, 1) - 1.111 \cdot (2, 2)$$

$$w = (-1.3334, -0.6667)$$



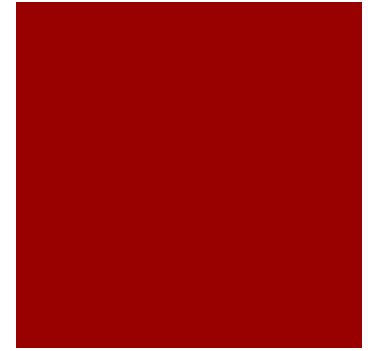
$$-1.3334 \cdot x_1 - 0.6667 \cdot x_2 + 3 = 0$$

# A **K**ernel-based **L**earning **P**latform



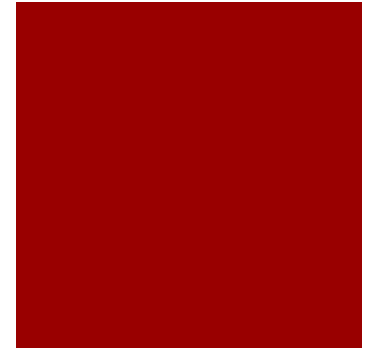
- **KeLP** is a Java open source Machine Learning platform focusing on Kernel machines
  - <http://sag.art.uniroma2.it/demo-software/kelp/kelp-installation/>
- Kernel are decoupled from learning algorithms, through the definition of specific interfaces
  - Support for the following learning tasks, e.g. Classification, Regression, Learning over sequences, Clustering...
  - Several learning algorithms are implemented for
    - **Batch Learning**: SVM, Pegasos, ...
    - **Online Learning**: Perceptron, Passive-Aggressive, ...

# Kelp and Maven



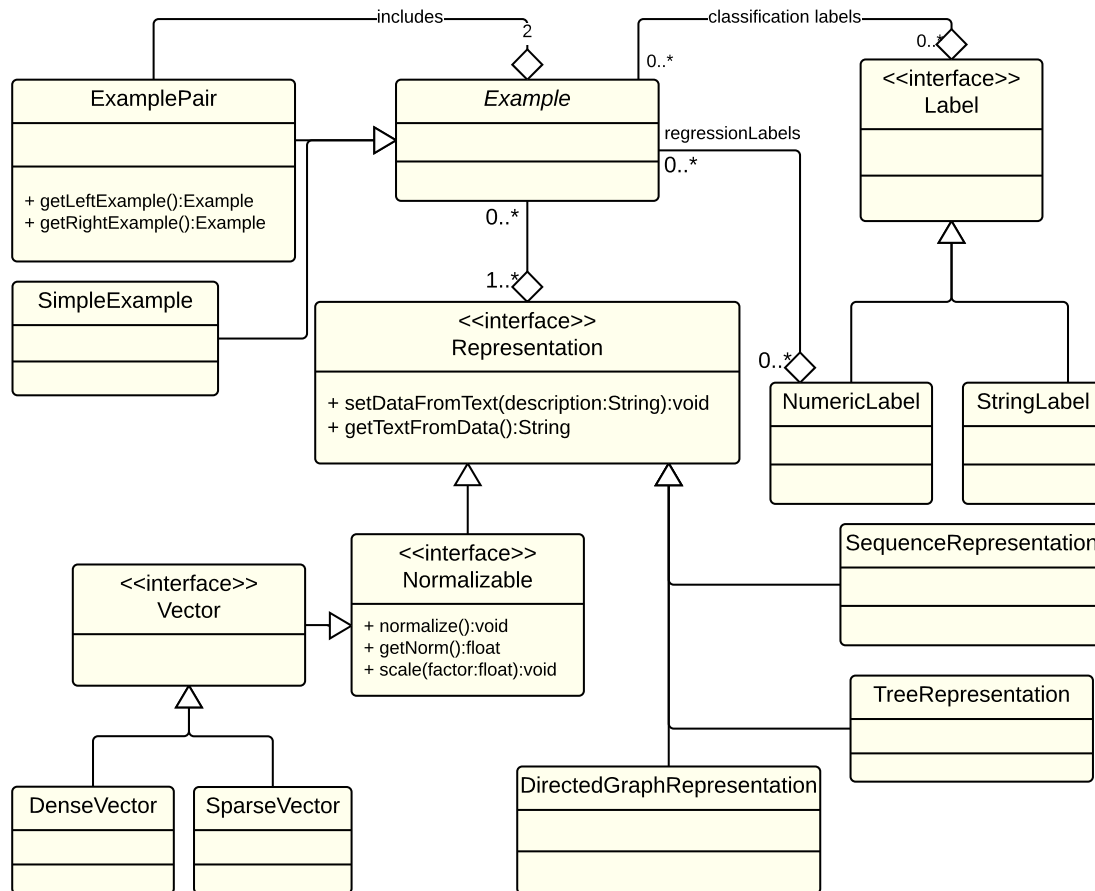
- You can import the KeLP library via Maven
  - Instructions are available at:  
<http://sag.art.uniroma2.it/demo-software/kelp/kelp-installation/>
- The source code is open-source and you can download it from  
<https://github.com/SAG-KeLP>
- KeLP is divided in four java projects
  - `kelp-core`: it contains the core interfaces and classes for algorithms, kernels and representations. (It contains the stands SVM)
  - `kelp-additional-kernels`: it contains additional kernel functions, such as the Tree Kernels.
  - `kelp-additional-algorithms`: it contains additional learning algorithms
  - `kelp-full`: it aggregates via Maven all the above projects.

## Examples/Representation/Algorithms and Java Objects



- Each example is stored in the `Example` object and it is characterized by
  - A set of `Labels` reflecting one or more classes
  - A list of `Representations` each identified by a `String`
- Examples are collected in `Datasets`
- Several `LearningAlgorithms` have been implemented
  - `BinaryCSvmClassification` implements the SVM learning algorithm seen in previous lessons

# A Class Diagram of main objects



# Input Data



- A dataset can be stored in a textual file (this is not true for graphs)
  - Each row of a dataset represents an `Example` according to the following format  
`label1 ... labelN | Btype1:name1 | descrip. | Etype1 | | Btype2:name2 | descrip. | Etype2 |`
- The list of strings `label1 ... labelN` identifies the classes
- An example can be represented through multiple representations, whose `type` must be specified:
  - `V`: Vector (default is sparse vector)
  - `DV`: Dense Vector
  - `T`: Tree
  - `S`: String
  - `SQ`: Sequence
- Each representation has a `name` used by the kernel function for a proper selection
  - A representation starts with a tab `B` (e.g., `BV`) and is closed with a tag `E` (e.g., `EV`)

# Let us consider the Question Classification task...



- Question classification consists in assigning a question to a class reflecting the intention of the question.

Example: “*What is the width of a football field?*” → Number

- A QC dataset is available at:

<http://cogcomp.cs.illinois.edu/Data/QA/QC/>

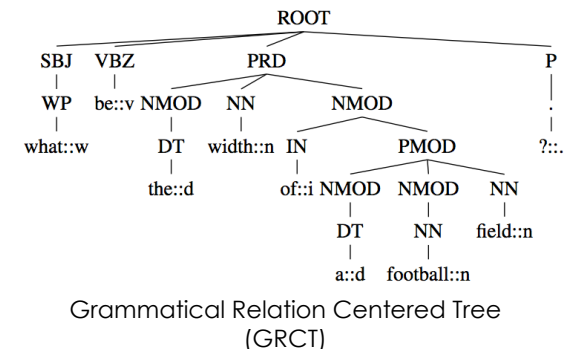
- Train dataset: 5,452 questions
- Test dataset: 500 questions
- Two settings:
  - Coarse-Grained: 6 classes ← We will focus on this setting
  - Fine-grained: 50 classes

[1] Xin Li, Dan Roth, Learning Question Classifiers. COLING'02, Aug., 2002.

# Multiple ways to represent a question

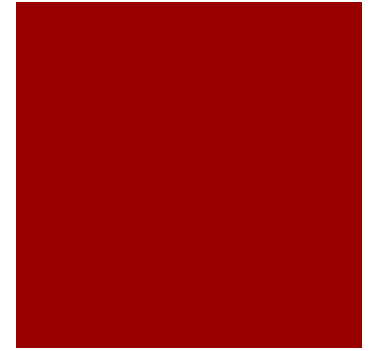
NUM |BV:bow| \_what\_\_w:1.0 \_a\_\_d:1.0 \_field\_\_n:1.0 \_football\_\_n:1.0  
\_the\_\_d:1.0 \_be\_\_v:1.0 \_width\_\_n:1.0 \_of\_\_i:1.0 \_?\_\_.:1.0 |EV| |BT:grct|  
(SYNT##root(POS##WP(LEX##what::w))(SYNT##cop(POS##VBZ(LEX##be::v)))(SYNT##nsu  
bj(SYNT##det(POS##DT(LEX##the::d)))(POS##NN(LEX##width::n))(SYNT##prep\_of(SYN  
T##det(POS##DT(LEX##a::d)))(SYNT##nn(POS##NN(LEX##football::n)))(POS##NN(LEX#  
#field::n))))|ET|  
|BS:quest| What is the width of a football field ?|ES|

- **NUM** is the class assigned to the question
- |BV:bow| \_what\_\_w:1.0 \_a\_\_d:1.0 ... |EV|  
it is a vector (V) called bow (do you remember the Bag-of-words representation?)
- |BT:grct| (SYNT##root(POS##WP(LEX##what::w)) ... |EV|  
it is a tree (T) derived from the dependency parsing of the question written in parenthetic form
- |BS:quest| What is the width of a football field ?|ES|  
It is a string (S). It is not used by kernel functions but it can be used to comment examples.



see Danilo Croce, Alessandro Moschitti, Roberto Basili: *Structured Lexical Similarity via Convolution Kernels on Dependency Trees*. In proceedings of EMNLP 2011: pages 1034-1046

# Handling datasets



- Given a dataset in a file, we can load the dataset with

```
String datasetFilePath="qc_train.klp"
SimpleDataset trainingSet = new SimpleDataset();
trainingSet.populate(datasetFilePath);
```
- We can access the examples and representations

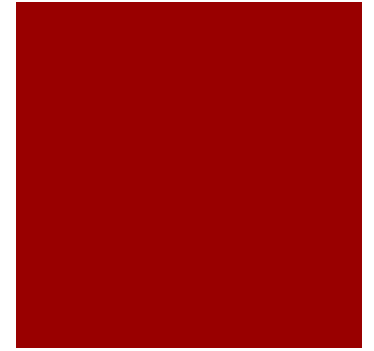
```
for(Example e: trainingSet.getExamples()){
    Representation rep = e.getRepresentation("bow");
}
```
- We can shuffle the dataset

```
SimpleDataset shDataset = trainingSet.getShuffledDataset();
```
- We can split the dataset in two datasets according to a split rate

```
float splitRate=0.8f;
SimpleDataset[] split = trainingSet.split(splitRate);
```
- Check if an example is associated to a class

```
StringLabel stringLabel = new StringLabel("NUM");
boolean isNum = e.isExampleOf(stringLabel);
```

# Kernels



- KeLP implements the following kernels seen in the previous lessons

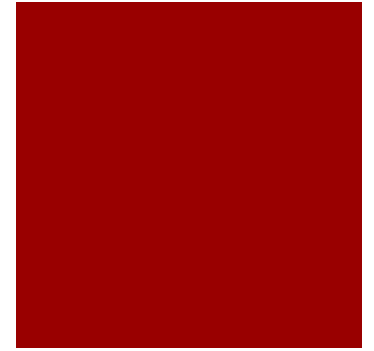
- Linear Kernel

```
String vectorRepName = "bow";  
Kernel linearKernel = new LinearKernel(vectorRepName);
```

- Polynomial Kernel

```
String vectorRepName = "bow";  
int exp=2;  
Kernel linearKernel = new LinearKernel(vectorRepName );  
Kernel polynKernel = new PolynomialKernel(exp, linearKernel);
```

# Kernels (2)



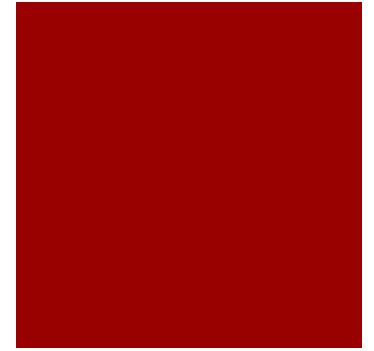
- Tree kernel (Vishwanathan and Smola, 2003)

```
String treeRepresentationName = "grct";  
float lambda = 0.4f;  
Kernel tkgrct = new SubSetTreeKernel(lambda,  
treeRepresentationName);
```

- Linear combination

```
String vectorRepresentationName = "bow";  
String treeRepresentationName = "grct";  
float lambda = 0.4f;  
  
Kernel linearKernel = new LinearKernel(vectorRepresentationName);  
Kernel tkgrct = new SubSetTreeKernel(lambda,  
treeRepresentationName);  
  
LinearKernelCombination combination = new LinearKernelCombination();  
combination.addKernel(0.7, linearKernel);  
combination.addKernel(0.3, tkgrct);
```

# Kernels (2)



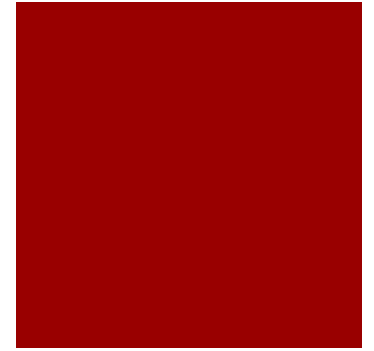
## ■ Kernel Normalization

```
Kernel linearKernel = new LinearKernel(vectorRepName);  
Kernel normLinearKernel = new NormalizationKernel(linearKernel);
```

## ■ Kernel Normalization and Combination

```
Kernel linearKernel = new LinearKernel(vectorRepName);  
Kernel normLinearKernel = new NormalizationKernel(linearKernel);  
  
Kernel treeKernel = new SubSetTreeKernel(lambda, treeRepName);  
Kernel normTreeKernel = new NormalizationKernel(treeKernel);  
  
LinearKernelCombination comb = new LinearKernelCombination();  
comb (0.7, normLinearKernel );  
comb (0.3, normTreeKernel );
```

# Binary Learning Algorithms

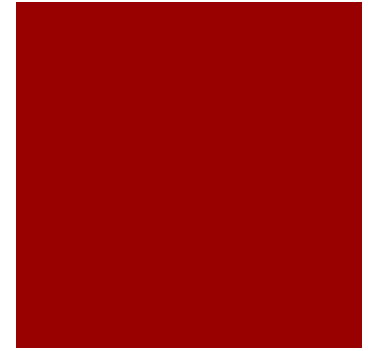


```
// define the positive class
StringLabel positiveClass = new StringLabel("+1");
// instantiate a learning algorithm
BinaryCSvmClassification learningAlgo = new
BinaryCSvmClassification();
// indicate to the learner what is the positive class
learningAlgo.setLabel(positiveClass);
// set the regularization parameters
learningAlgo.setCp(c);
learningAlgo.setCn(c);

// set the kernel function
Kernel linearKernel = new LinearKernel("repr_name");
learningAlgo.setKernel(linearKernel);

// learn and get the prediction function
learningAlgo.learn(trainingSet);
Classifier classifier = learningAlgo.getPredictionFunction();
```

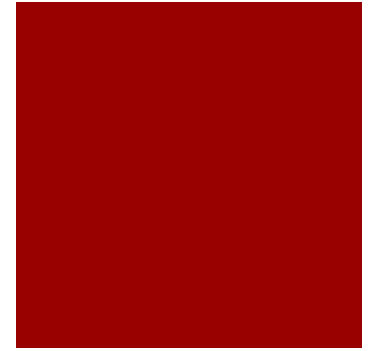
# From binary to multi-class classifiers



- When multiple classes are involved, we can combine several binary classifiers to build a multi-class classifier
- We adopt the **One-VS-All classification** schema
  - We adopt in competition as many classifiers as involved classes
- **Training:** at turn, a binary classifier is trained over the training set, with the set of examples of a class considered as positive examples
  - The remaining examples are considered negative examples
- **Test:** the example is classified with each binary classifier and the class associated to the classifier with the maximum classification score is selected

# Multiclass classification function

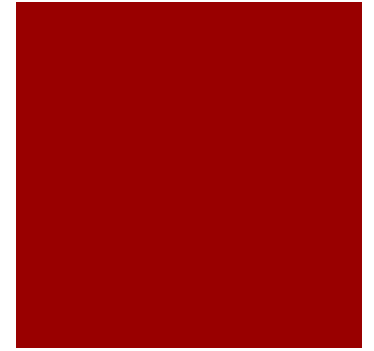
## The One Vs All Schema



```
List<Label> labels = trainingSet.getClassificationLabels();
// instantiate the basic binary learning algorithm.
// NO need of setting the labels
BinaryCSvmClassification baseAlgo = new BinaryCSvmClassification();
// set the regularization parameters
baseAlgorithm.setCp(c);
baseAlgorithm.setCn(c);
// set the kernel function
Kernel linearKernel = new LinearKernel("repr_name");
baseAlgorithm.setKernel(linearKernel);

OneVsAllLearning ovaLearning=new OneVsAllLearning();
// set the binary classification function
ovaLearning.setBaseAlgorithm(baseAlgorithm);
// set the targeted classes
ovaLearning.setLabels(labels);
// learn and get the prediction function
ovaLearning.learn(trainingSet);
// get the classification function
OneVsAllClassifier ovaCl = ovaLearning.getPredictionFunction();
```

# Caching kernel evaluations

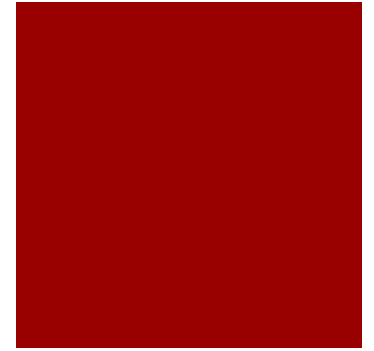


- The running time of some kernel-based learning algorithm can be almost  $O(n^2)$
- These algorithms evaluate the kernel function between two examples multiple times
- To reduce the learning time we can save some kernel computations

```
KernelCache cache=new FixIndexKernelCache(cacheSize);  
usedKernel.setKernelCache(cache);
```

- Useful in the training phase
  - In testing it is not useful for binary classification but it can be useful for Multi-class classification
  - Be careful when using the cache during training over datasets with hundred of thousands examples
- To switch the cache off: `usedKernel.disableCache();`

# Prediction Functions



## ■ Binary Classifier

```
ClassificationOutput p = classifier.predict(testExample);  
float getClassificationScore=p.getScore(positiveClass);  
if(getClassificationScore>0)  
    ...
```

## ■ Multi Class Classifier

```
ClassificationOutput pred = ovaClassifier.predict(testExample);  
List<Label> predictedClasses = pred.getPredictedClasses();  
Label predictedLabel = predictedClasses.get(0);  
    ...
```

# Evaluators

- To support the evaluation phase, KeLP implements evaluators for:
  - The binary scenario
  - The multi-class scenario

```
//Building the evaluation function
```

```
BinaryClassificationEvaluator evaluator = new  
BinaryClassificationEvaluator(positiveClass);
```

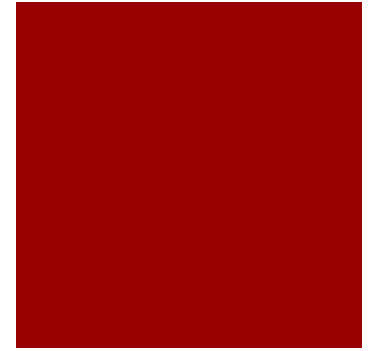
```
// Classify examples and compute the accuracy
```

```
for (Example e : testSet.getExamples()) {  
    ClassificationOutput p = classifier.predict(e);  
    evaluator.addCount(e, p);  
}
```

```
// Get evaluation metrics
```

```
float accuracy = evaluator.getAccuracy();  
float precision = evaluator.getPrecision();  
float recall = evaluator.getRecall();  
float f1 = evaluator.getF1();
```

# Evaluators for the Multi-class classification schema

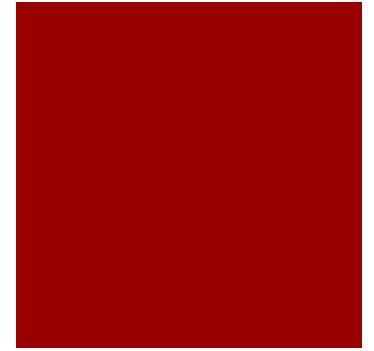


```
//Building the evaluation function
List<Label> labels = trainingSet.getClassificationLabels();
MulticlassClassificationEvaluator evaluator = new
    MulticlassClassificationEvaluator(labels);

// Classify examples and compute the accuracy
for (Example e : testSet.getExamples()) {
    ClassificationOutput p = classifier.predict(e);
    evaluator.addCount(e, p);
}

// Get evaluation metrics
float accuracy = evaluator.getAccuracy();
for (Label label : labels) {
    float precision = evaluator.getF1For(label);
    float recall = evaluator.getF1For(label);
    float f1 = evaluator.getF1For(label);
}
```

# Saving/Loading the model



- Once the classifier has been learned it can be saved into a file

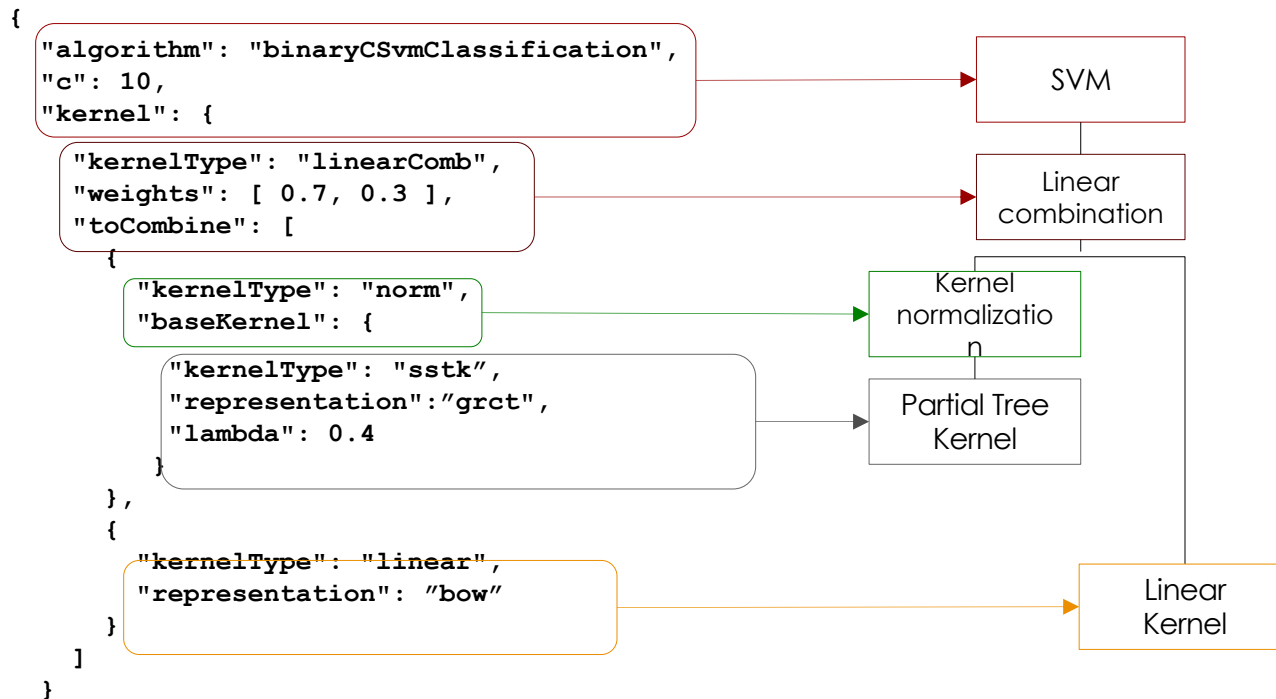
```
JacksonSerializerWrapper serializer = new JacksonSerializerWrapper();  
serializer.writeValueOnFile(classifier, "classifier_file_name.klp");
```

- And it can be also loaded

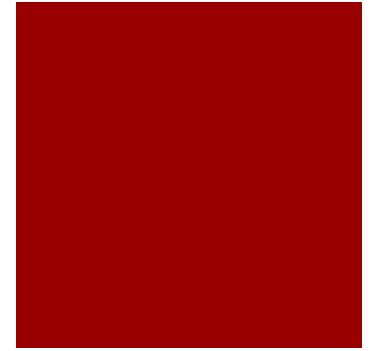
```
File inputFile=new File("classifier_file_name.klp");  
Classifier classifier=serializer.readValue(inputFile, Classifier.class);
```

- *But also kernel functions and learning algorithms can be serialized on file*
- *It means that you can build and tune the kernel/classifier without reading a single line of JAVA code*

# A learning algorithm in Json



# Homework



- In the folder you will find a README with the instructions to compile and execute the JAVA code
- For Homework you are required to:
  1. parameterize the kernel functions with a 5-fold schema
  2. learn the classifier maximizing the F1 metrics for the HUMAN class
  3. evaluate the classifiers without a direct use of the evaluators

# Homework – A Glimpse into Future Lessons



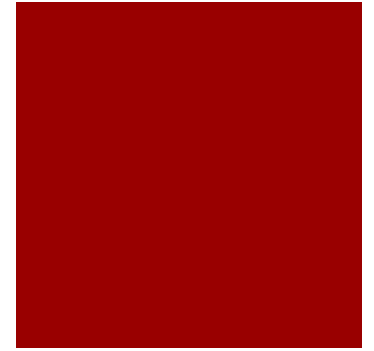
We will soon explore how to encode sentences using **neural networks**.

This allows us to obtain **dense embeddings** for full sentences, such as questions.

## Embeddings from BERT

- For this assignment, we provide datasets where each question is enriched with an embedding obtained using a pre-trained BERT model (msmarco\_distilbert):
  - qc\_train\_with\_embeddings.klp
  - qc\_test\_with\_embeddings.klp
- These dense vector representations can be used just like other representations in KeLP.

# Combining Embeddings with Kernels



You can experiment with:

- `ova_lin_embeddings.klp`: A linear kernel over the BERT embeddings
- `ova_lin_embeddings_norm.klp`: A normalized linear kernel over the embeddings
- `ova_comb_lin-bow_lin_embeddings.klp`: A combination of linear kernel on Bag-of-Words and on BERT embeddings

This allows you to compare traditional and neural representations within the same learning framework.