KeLP: Kernel-based Learning Platform
Official Guide of the version 2.0.2
Draft version

Contents

1 KeLP: a Kernel-based Learning Platform **3**

2 An overview of the library **5**
- 2.1 Importing KeLP via Maven 6

3 Data Structures **9**
- 3.1 Existing Representations 9
- 3.2 Input Format 11
- 3.3 Advanced Topics on Data 14
  - 3.3.1 Defining new Representations 14
  - 3.3.2 Manipulating Data 14

4 Kernels **17**
- 4.1 Kernel Organization 17
- 4.2 Available Kernels and their JSON Definition 19
  - 4.2.1 Direct Kernels 20
- 4.3 Kernel Compositions 24
  - 4.3.1 Kernel Combinations 26
  - 4.3.2 Kernel On Pairs 27
- 4.4 Advanced Topics on Kernels 29
  - 4.4.1 Defining new Kernels 29
  - 4.4.2 Speeding up Kernel Machines through Caching Mechanisms . 31

5 Learning Algorithms **35**
- 5.1 Learning Algorithms Organization 35
- 5.2 Available Learning Algorithms and their JSON Definition 38
  - 5.2.1 Batch Learning Algorithms 39
  - 5.2.2 Online Learning Algorithms 43
  - 5.2.3 Clustering Algorithms 47
  - 5.2.4 Linearization Functions 47
  - 5.2.5 Meta-learning Algorithims 48
- 5.3 Evaluators 51
- 5.4 Advanced Topics on Learning Algorithms 53
  - 5.4.1 Defining new Learning Algorithms 53

6 Sample Code **59**
6.1 Classification example 59
6.2 Usage of ExperimentUtils facilities 60
6.3 Instantiation from JSON 61

1

KeLP: a Kernel-based Learning Platform

Most of the existing Machine Learning (ML) platforms assume that instances are represented as vectors in a feature space, e.g., [Joachims(1999),Hall *et al.*(2009),Chang and Lin(2011)], that must be defined beforehand. For example, in Natural Language Processing (NLP) the definition of a feature space often requires a complex feature engineering phase. Let us consider any NLP task in which syntactic information is crucial, e.g., *Boundary Detection* in Semantic Role Labeling [Carreras and Màrquez(2005)]. Understanding which syntactic patterns should be captured is non-trivial and usually the resulting feature vector model is a poor approximation. Instead, a more natural approach is operating directly with the parse tree of sentences. Kernel methods [Shawe-Taylor and Cristianini(2004)] provide an efficient and effective solution, allowing to represent data at a more abstract level, while their computation still looks at the informative properties of them. For instance, Tree Kernels take in input two syntactic parse trees, and compute a similarity measure by looking at the shared sub-structures.

In this guide, KeLP, a Java kernel based learning platform is presented. It supports the implementation of Kernel-based learning algorithms, as well as kernel functions over generic data representations, e.g., vectorial data or discrete structures, such as graphs, trees, and sequences. The framework has been designed to decouple data structures, kernel functions and learning algorithms in order to maximize the re-use of existing functionalities: as an example, a new kernel can be included inheriting existing algorithms, and vice versa. KeLP supports XML and JSON serialization of kernel functions and algorithms, enabling the agile definition of kernel-based learning systems without writing additional lines of code. KeLP can effectively tackle a wide variety of learning problems, including classification, regression and clustering.

2
An overview of the library

KeLP is a machine learning library completely written in Java. The Java language has been chosen in order to be compatible with the Java world, as it is the most used language in production environments[1]. Moreover, in NLP/IR many tools are based on the Java language, such as Stanford CoreNLP[2], OpenNLP[3] or Lucene[4]. KeLP is released as open source software under the Apache 2.0 license and the source code is available on the github platform[5]. Furthermore, it is released in packages through Maven. A detailed documentation of KeLP with helpful examples and use cases is available on the website of the Semantic Analytics Group[6] of the University of Roma, Tor Vergata.

KeLP has been developed in different Maven packages according to a modularization aimed at logically separating the different components of the library. This allows users to include only the needed modules. KeLP is currently composed of the following 4 packages:

- `kelp-core`: it contains the infrastructure of abstract classes and interfaces to work with KeLP. Furthermore, some implementations of algorithms, kernels and representations are included, to provide a base operative environment.
- `kelp-additional-kernels`: it contains several kernel functions that extend the set of kernels made available in the kelp-core project. Moreover, this project implements the specific representations required to enable the applica-

[1] http://www.tiobe.com/tiobe_index
[2] http://nlp.stanford.edu/software/corenlp.shtml
[3] https://opennlp.apache.org/
[4] http://lucene.apache.org/
[5] https://github.com/SAG-KeLP
[6] http://sag.art.uniroma2.it/demo-software/kelp/

tion of such kernels. In this project the following kernel functions are considered: Sequence kernels, Tree kernels and Graphs kernels.

- `kelp-additional-algorithms`: it contains several learning algorithms extending the set of algorithms provided in the kelp-core project, e.g., the C-Support Vector Machine or ν-Support Vector Machine learning algorithms. In particular, advanced learning algorithms for classification and regression can be found in this package. The algorithms are grouped in: (*i*) Batch Learning, where the complete training dataset is supposed to be entirely available during the learning phase; (*ii*) Online Learning, where individual examples are exploited one at a time to incrementally acquire the model.

- `kelp-full`: this is the complete package of KeLP. It aggregates the previous modules in one jar. It contains also a set of fully functioning examples showing how to implement a learning system with KeLP. Batch learning algorithm as well as Online Learning algorithms usage is shown here. Different examples cover the usage of standard kernel, Tree Kernels and Sequence Kernel, with caching mechanisms.

2.1 Importing KeLP via Maven

All the KeLP packages are released under Maven in our repositories.

Apache Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a projects build, reporting and documentation from a central piece of information.

For now, a custom repository is used to distribute the maven packages for the whole platform. To use KeLP it is necessary to specify the repository where the platform is located. In the pom.xml of a Maven project, KeLP repositories can be specified with the following piece of code:

```
<repositories>
  <repository>
    <id>kelp_repo_snap</id>
    <name>KeLP Snapshots Repository</name>
    <releases>
      <enabled>false</enabled>
      <updatePolicy>always</updatePolicy>
      <checksumPolicy>warn</checksumPolicy>
    </releases>
    <snapshots>
      <enabled>true</enabled>
```

```
      <updatePolicy>always</updatePolicy>
      <checksumPolicy>fail</checksumPolicy>
    </snapshots>
    <url>
      http://sag.art.uniroma2.it:8081/artifactory/kelp-snapshot/
    </url>
  </repository>
  <repository>
    <id>kelp_repo_release</id>
    <name>KeLP Stable Repository</name>
    <releases>
      <enabled>true</enabled>
      <updatePolicy>always</updatePolicy>
      <checksumPolicy>warn</checksumPolicy>
    </releases>
    <snapshots>
      <enabled>false</enabled>
      <updatePolicy>always</updatePolicy>
      <checksumPolicy>fail</checksumPolicy>
    </snapshots>
    <url>
      http://sag.art.uniroma2.it:8081/artifactory/kelp-release/
    </url>
  </repository>
</repositories>
```

A full version of KeLP is available on maven in the package kelp-full. It includes all the modules that compose the library. To import the 2.0.2 version of kelp-full, you can use the following Maven dependency.

```
<dependencies>
      <dependency>
            <groupId>it.uniroma2.sag.kelp</groupId>
            <artifactId>kelp-full</artifactId>
            <version>2.0.2</version>
      </dependency>
</dependencies>
```

Maven allows also to import a subset of the packages. For example, if you need only to work with tree kernel functions, you can import only the kelp-additional-kernels package in your project. For example, to import the 2.0.2 version of this package, the dependency specification to be added in your pom.xml is:

```
<dependencies>
      <dependency>
            <groupId>it.uniroma2.sag.kelp</groupId>
```

```
            <artifactId>kelp-additional-kernels</artifactId>
            <version>2.0.2</version>
      </dependency>
</dependencies>
```

For a complete example on how to import KeLP packages via Maven, please refer to the pom.xml that is contained in the kelp-full project, that can be found on GitHub[7].

[7] https://github.com/SAG-KeLP/kelp-full

3

Data Structures

In KeLP a machine learning instance is modeled by the class `Example`. As illustrated in Figure 3.1 there are two implementations of such class:

- `SimpleExample`: it models an individual machine learning instance. It is suitable for most of the scenarios.
- `ExamplePair`: it models machine learning instance naturally structured into pairs, such as *question-answer* in Question Answering or *text-hypothesis* in Textual Entailment.

Every example has a set of `StringLabels` for classification tasks and a set of `NumericLabels` for regression problems. Multiple labels associated to a single example allow for tackling multi-label classification or multi-variate regression tasks. An example can have no label, as in the case of clustering problems.

Furthermore, an example is composed by a set of representations. This allows to model data instances from different viewpoints, and to perform a joint learning model, where multiple representations are exploited at the same time (i.e., using kernel combinations).

3.1 Existing Representations

KeLP supports both vectorial and structured data to model learning instances. For example, `SparseVector` can host a Bag-of-Words model, while `DenseVector` can represent data derived from low dimensional embeddings. They are both available into `kelp-core`, allowing to have the standard environment provided by most of the existing machine learning platforms.

`kelp-additional-kernels` provides structure representations:

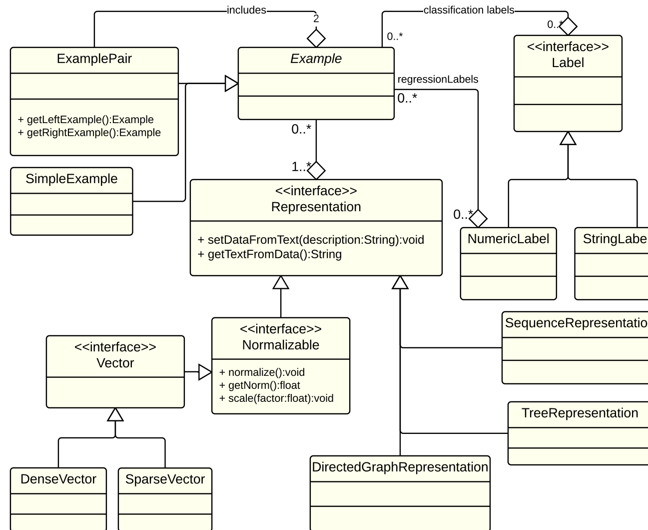

Figure 3.1: KeLP data class diagram

- `TreeRepresentation`: it models a tree structure that can be employed for representing syntactic trees.
- `SequenceRepresentation`: it models a sequence that can be employed for representing, for example, sequences of words;
- `DirectedGraphRepresentation`: it models a directed unweighted graph structure, i.e, any set of nodes and directed edges connecting them. Edges do not have any weight or label.

The nodes in these three structured representations are associated with a generic structure (`StructureElement`) containing the node label and possible additional information. For example, in a TreeRepresentation of a sentence leaves can represent lexical items. Nodes can thus model lexical items, where the node label represent the word, and the additional information can be constituted by a vector representation of a word, as the ones produced by Word Embedding methods.

Finally a `StringRepresentation` is included to store a plain text, useful for associating a comment to an example.

3.2 Input Format

The dataset input format for KeLP takes inspiration from the SvmLight/LibSVM formalism, extending it in order to deal with multiple labels and multiple representations. We did not exploit JSON cause it would have implied a lot of overhead.

A dataset is generally represented in a text file, where each row represents an example, that has one of the following two forms:

label$_1$ ... label$_N$ |B*type*$_1$:*name*$_1$| *description* |E*type*$_1$| |B*type*$_2$:*name*$_2$| *description* |E*type*$_2$| ...

label$_1$ ... label$_N$ |<| *rightExample* |,| *leftExample*|>|

The former row refers to a `SimpleExample` while the latter describes an `ExamplePair`, where *leftExample* and *rightExample* recursively have the form of one of these two formalisms.

Each example starts with a list of labels (highlighted in red) separated by a white space. A label can be a simple string in the case of a classification label, or can have the form *propertyName:value* (for instance height:10) in the case of regression values. Note that an isolated number will be considered a classification label.

In the `SimpleExample` case, after the labels parts, a list of representations begins. In the previous example there are two representations, highlighted in blue and green. Each representation must be included between a *begin of representation* sequence of the form |B*type:name*| and an *end of representation* sequence of the form |E*type*| where *type* is an identifier of the representation class (e.g., V for `SparseVector`) and *name* is an identifier for that specific representation (e.g., BoW for a bag-of-words representation). If no name is specified for a representation, it will be identified by its position within the sequence (i.e., the third representation will be automatically named 3). The name identifies uniquely a representation for an example and it is necessary to support examples having multiple representations of the same class.

Each representation has its own formalism:

- `DenseVector`. Its type identifier is *DV* and its textual description is a sequence of numbers separated by a white space (or a comma, or a semicolon). For instance:

  |BDV:lsa|10;89;0.4;-43;19;-9.3 |EDV|

- `SparseVector`. Its type identifier is *V* and its textual description is a sequence of *featureName:featureValue* pairs separated by a white space (i.e., the same formalism of SVMlight and LibSVM, but featureName is not forced to be a number, i.e., it can be a generic string). For instance:

|BV:bow| KeLP:0.33 is:0.33 amazing:0.33 |EV|

- `StringRepresentation`. Its type identifier is *S* and its textual description is a simple text. For instance:

|BS:comment| KeLP is amazing |ES|

The structured representations have nodes whose content is a `StructureElement`. Its textual format is a pair *type##content*, where *type* identifies a the specific implementation of the class `StructureElement`, while *content* is a text defining the parameters of the structure element. Every implementation of `StructureElement` has its own *content* formalism. For instance:

- `LexicalStructureElement`: Its type identifier is *LEX* and its content has the form *word::part-of-speech*, as in *LEX##KeLP::n*;
- `PosStructureElement`: Its type identifier is *POS* and its is a simple part-of-speech symbol, as in *POS##NN*;
- `SyntacticStructureElement`: Its type identifier is *SYNT* and its is a simple syntactic symbol (e.g., a constituent, a chunk, or a syntactic dependency), as in *SYNT##VP*;
- `CompositionalStructureElement`: Its type identifier is *COMP* and its content has the form *<head,modifier>*, as in *COMP##<tool,useful>*;
- `UntypedStructureElement`: Its type identifier is *NOTYPE* and its content is a generic text, as in *NOTYPE##KeLP*.

  This is the default `StructureElement` that is instantiated when the *type* information is missing (and the separator *##* is missing too); for instance the text *KeLP* is automatically instantiated as an `UntypedStructureElement`.

The `StructureElement` formalism is employed in the formats of the three structured representations:

- `SequenceRepresentation`. Its type identifier is *SQ* and its textual description is a sequence of structured elements in round brackets, as in:

|BSQ:sequence| (LEX##KeLP::n) (LEX##is::v) (LEX##amazing::j) |ESQ|

- `TreeRepresentation`. Its type identifier is *T* and its textual description must be in the Penn Treebank[1] notation, where each node label must respect the `StructureElement` formalism. For instance[2]:

  |BT:constTree| (S (NP (NNP KeLP)) (VP (VBZ is) (ADJP (JJ amazing)))) |ET|

- `DirectedGraphRepresentation`. Its type identifier is *G* and its textual description ...

|BG:graph| |EG|

Given a file written in the KeLP format, it can be loaded by simply calling the `populate` method:

```
SimpleDataset dataset = new SimpleDataset();
dataset.populate("datasetPath.klp");
```

Alternatively, it is possible to pass to the populate function an instance of `Datase-tReader`, allowing to read different data formats. Currently, KeLP supports the CSV data format and the LibSVM/SvmLight formats.

```
/*
 * Loading data in CSV format
 */
SimpleDataset dataset = new SimpleDataset();
String path = "datasetPath.csv";
String representationName = "featureVector";
boolean skipFirstLine = true; //in case of header
LabelPosition position = LabelPosition.LAST_COLUMN;
CsvDatasetReader csvReader = new CsvDatasetReader(path,
    representationName, skipFirstLine, position);
dataset.populate(csvReader);
```

```
/*
 * Loading data in LibSVM/SVMLight format
 */
SimpleDataset dataset = new SimpleDataset();
String path = "datasetPath.libSvm";
String representationName = "featureVector";
LibsvmDatasetReader libSvmReader = new LibsvmDatasetReader(path,
    representationName);
dataset.populate(libSvmReader);
```

[1] `http://www.cis.upenn.edu/~treebank/`
[2] in the example the compact format of the `UntypedStructureElement` is adopted

3.3 Advanced Topics on Data

3.3.1 Defining new Representations

A new representation can be added by creating a class that implements the interface `Representation`. The programmer has to implement two specific methods, `getTextFromData` and `setDataFromText`, which are necessary for serializing the Representation in a string format and deserializing it. The decision on which textual format using for describing the representation to be added is left to the programmer, i.e., a JSON/XML format is not imposed. An empty constructor must be included. Optionally, the class can be annotated with `@JsonTypeName` in order to specify an alternative type name to be used during the serialization/deserialization mechanism, otherwise the class name will be automatically used. These simple passages automatically enrich the KeLP data format allowing to load datasets containing the added representation.

If a norm can be defined on the representation to be added (i.e., the representation is a vector, a matrix or a higher order tensor), then the `Normalizable` interface can be implemented, enabling some useful preprocessing operations on datasets like feature scaling or data normalization (see section 3.3.2). In this case the following methods must be implemented:

- `getSquaredNorm`: returns the squared norm of this vector;
- `normalize`: scales the representation in order to have a unit norm in the explicit feature space;
- `scale`: multiplies each element of the representation by a given coefficient.

3.3.2 Manipulating Data

KeLP is a general-purpose machine learning platform and does not cover any feature extraction aspect. However it provides some simple data preprocessing features to manipulate the input data. Specific operations on data can be defined by implementing the `Manipulator` interface. Instances of such class can be then passed to the method `manipulate` of the class `Dataset` in order to perform the manipulation operations to the whole dataset.

Currently, the following implementations of the class `Manipulator` are available:

- `NormalizationManipolator`: it scales the selected representations in order to be a unit vector in its explicit feature space. This can be useful when

the orientation of the feature vectors is meaningful, while their magnitude is not relevant;

- `StandardizationManipulator`: It standardizes the feature values of a vectorial representation. Let x_i be the value of the i-th feature whose mean and standard deviation are μ_i and σ_i respectively. Then the standardized value is $\hat{x_i} = (x_i - \mu_i)/\sigma_i$. This operation is useful in order to map all the feature to a similar range.

- `VectorConcatenationManipulator`: it allows to concatenate vectors into a new `SparseVector` representation. It is useful when a linear approach must be applied to multiple vectorial representations;

- `PairSimilarityExtractor`: it analyzes an `ExamplePair` extracting some similarity scores between the left and the right examples of the pair. The extracted similarity scores are stored in a `DenseVector` that is added to the representations set of the `ExamplePair` to be manipulated.

- `TreePairRelTagger`: given an `ExamplePair` whose left and right examples contain `TreeRepresentations`, it performs the *REL* tagging described in [Filice *et al.*(2015)].

4 Kernels

Kernel methods are a powerful class of algorithms for pattern analysis that, exploiting the so called kernel functions, can operate in an implicit high-dimensional feature space without explicitly computing the coordinates of the data in that space. Most of the existing machine learning platforms provide kernel methods that operate only on vectorial data. On the contrary, KeLP has the fundamental advantage that there is no restriction on a specific data structure, and kernels directly operating on vectors, sequences, trees, graphs, or other structured data can be defined.

Furthermore, another appealing characteristic of KeLP is that kernels can be composed and combined in order to create richer similarity metrics in which different information from different `Representations` can be simultaneously exploited.

4.1 Kernel Organization

As shown in figure 4.1, KeLP completely supports the composition and combination mechanisms providing three abstractions of the `Kernel` class:

- `DirectKernel`: in computing the kernel similarity it operates directly on a specific `Representation`. For instance `LinearKernel` works over `Vector` representations, or tree kernels operate on `TreeRepresentations`;
- `KernelComposition`: it enriches the kernel similarity provided by any another `Kernel`. Some KeLP implementations are `PolynomialKernel`, or `RBFKernel`;
- `KernelCombination`: it combines different `Kernels` in a specific function. Some KeLP implementations are `LinearKernelCombination`, or `KernelMultiplication`;

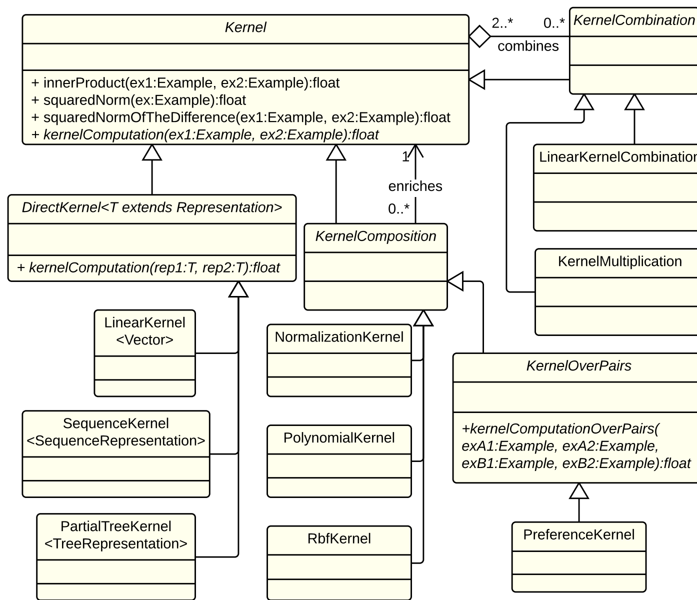

Figure 4.1: KeLP kernel class diagram

Moreover, the class `KernelOnPairs` models kernels operating on `ExamplePairs`: all kernels discussed in [Filice *et al.*(2015)] implement this class.

4.2 Available Kernels and their JSON Definition

One of the most useful features of KeLP is the possibility to serialize/deserialize kernels and learning algorithms in a readable JSON/XML format. This is useful for instantiating an algorithm or a kernel without writing a single line of Java code, i.e., the algorithm description can be provided in JSON to an interpreter that will instantiate it. Listing 4.1 reports a JSON example of a kernel-based Support Vector Machine operating in a one-vs-all schema, where a kernel linear combination between a normalized Partial Tree Kernel and a linear kernel is adopted. As the listing shows kernels and algorithms can be easily composed and combined in order to create new training models.

Listing 4.1: A JSON example.

```
{"algorithm" : "oneVsAll",
  "baseAlgorithm" : {
    "algorithm" : "binaryCSvmClassification",
    "c" : 10,
    "kernel" : {
      "kernelType" : "linearComb",
      "weights" : [1,1],
      "toCombine" : [
        {
          "kernelType" : "norm",
          "baseKernel" : {
            "kernelType" : "ptk",
            "mu" : 0.4,
            "lambda" : 0.4,
            "representation" : "parseTree"
          }
        },
        {
          "kernelType" : "linear",
          "representation" : "Bag-of-Words"
        }
      ]
    }
  }
}
```

Every kernel implementation has a specific JSON name, which is usually an abbreviation of the Java class name, such as `linearComb` for the `LinearKernelCombination`. The parameter `kernelType` in the JSON listing specifies the type of the kernel to be instantiated; for example, `"kernelType" : "ptk"` states that the kernel is an instance of `PartialTreeKernel`.

4.2.1 Direct Kernels

The implementations of the class `DirectKernel` directly compute the kernel similarity between two representations of a specific type, such as `Vector` in the case of `LinearKernel`. An example can have multiple representations of the same type, and the parameter `representation` univocally identifies the one on which the direct kernel has to operate. Referring to the listing 4.1, `"representation":"Bag-of-Words"` states that the linear kernel will operate on the `Vector` named *Bag-of-Words*[1]. KeLP currently implements the following `DirectKernels`:

Linear Kernel

It executes the dot product between two `Vector` representations.

JAVA CLASS: `LinearKernel`
MAVEN PROJECT: `kelp-core`
JSON TYPE: `linear`

Sequence Kernel

It is a convolution kernel between sequences. The algorithm corresponds to the recursive computation presented in [Bunescu and Mooney(2005)]

JAVA CLASS: `SequenceKernel`
MAVEN PROJECT: `kelp-additional-kernels`
JSON TYPE: `seqk`
PARAMETERS:

- `maxSubseqLeng`: it is the maximum length of common subsequences considered by the kernel
- `lambda`: the gap penalty

SubTree Kernel

A SubTree Kernel is a convolution kernel that evaluates the tree fragments shared between two trees. The considered fragments are subtrees, i.e., a node and its complete descendancy. For more details see [Vishwanathan and Smola(2002)].

[1] obviously it is expected that the input examples have a vector associated to such name, as in `-1 |BV:Bag-of-Words| ....|EV| |BT:parseTree| ... |ET|`.

JAVA CLASS: SubTreeKernel
MAVEN PROJECT: kelp-additional-kernels
JSON TYPE: stk
PARAMETERS:

- includeLeaves: whether the leaves must be involved in the kernel computation. Regardless its value, two subtrees are matched even if their leaves differ (but the other nodes must match). When it is true, matching leaves contribute to the kernel function; this corresponds to adding a bag-of-words similarity to the kernel function.
- lambda: the decay factor (associated to the height of a subtree)
- deltaMatrix: it is the data structure, where the kernel stores the results of the delta operations, i.e., the results of the intermediate operations in the recursive process of the convolution kernel. By default it is a 400x400 StaticDeltaMatrix, i.e., it is possible to calculate the kernel operation for input trees having up to 400 nodes.

SubSet Tree Kernel

A SubSetTree Kernel, a.k.a. Syntactic Tree Kernel, is a convolution kernel that evaluates the tree fragments shared between two trees. The considered fragments are subset-trees, i.e., a node and its partial descendancy (the descendancy can be incomplete in depth, but no partial productions are allowed; in other words, given a node either all its children or none of them must be considered). For further details see [Collins and Duffy(2001)].
JAVA CLASS: SubSetTreeKernel
MAVEN PROJECT: kelp-additional-kernels
JSON TYPE: sstk
PARAMETERS:

- includeLeaves: whether the leaves must be involved in the kernel computation. Regardless its value, two sub-set trees are matched even if their leaves differ (but the other nodes must match). When it is true, matching leaves contribute to the kernel function; this corresponds to adding a bag-of-words similarity to the kernel function.
- lambda: the decay factor (associated to the height of a sub-set tree)
- deltaMatrix: it is the data structure, where the kernel stores the results of the delta operations, i.e., the results of the intermediate operations in the recursive

process of the convolution kernel. By default it is a 400x400 `StaticDelta-Matrix`, i.e., it is possible to calculate the kernel operation for input trees having up to 400 nodes.

Partial Tree Kernel

A Partial Tree Kernel is a convolution kernel that evaluates the tree fragments shared between two trees. The considered fragments are partial trees, i.e., a node and its partial descendancy (the descendancy can be incomplete, i.e., a partial production is allowed). For further details see [Moschitti(2006)].
JAVA CLASS: `PartialTreeKernel`
MAVEN PROJECT: `kelp-additional-kernels`
JSON TYPE: `ptk`
PARAMETERS:

- `includeLeaves`: whether the leaves must be involved in the kernel computation. Regardless its value, two sub-set trees are matched even if their leaves differ (but the other nodes must match). When it is true, matching leaves contribute to the kernel function; this corresponds to adding a bag-of-words similarity to the kernel function.
- `lambda`: the decay factor (associated to the length of a production, including gaps)
- `mu`: the decay factor (associated to the height of a partial tree)
- `terminalFactor`: multiplicative factor to scale up/down the leaves contribution
- `maxSubseqLeng`: Maximum length of common subsequences considered in the recursion. It reflects the maximum branching factor allowed to the tree fragments.
- `deltaMatrix`: it is the data structure, where the kernel stores the results of the delta operations, i.e., the results of the intermediate operations in the recursive process of the convolution kernel. By default it is a 400x400 `StaticDelta-Matrix`, i.e., it is possible to calculate the kernel operation for input trees having up to 400 nodes.

Smoothed Partial Tree Kernel

A Smoothed Partial Tree Kernel (SPTK) is a convolution kernel that evaluates the tree fragments shared between two trees, [Croce *et al.*(2011)]. The considered fragments are partial trees (as for the Partial Tree Kernel) whose nodes are identical or similar according to a node similarity function: the contribution of the fragment pairs in the overall kernel thus depends on the number of shared substructures, whose nodes contribute according to such a metrics. This kernel is very flexible as the adoption of node similarity functions allows the definition of more expressive kernels, such as the Compositionally Smoothed Partial Tree Kernel [Annesi *et al.*(2014)]. Some examples of the usage of SPTK and node similarity function is reported in Section 6.
JAVA CLASS: `SmoothedPartialTreeKernel`
MAVEN PROJECT: `kelp-additional-kernels`
JSON TYPE: `sptk`
PARAMETERS:

- `lambda`: the decay factor (associated to the length of a production, including gaps)
- `mu`: the decay factor (associated to the height of a partial tree)
- `terminalFactor`: multiplicative factor to scale up/down the leaves contribution
- `maxSubseqLeng`: Maximum length of common subsequences considered in the recursion. It reflects the maximum branching factor allowed to the tree fragments.
- `deltaMatrix`: it is the data structure, where the kernel stores the results of the delta operations, i.e., the results of the intermediate operations in the recursive process of the convolution kernel. By default it is a 400x400 `StaticDeltaMatrix`, i.e., it is possible to calculate the kernel operation for input trees having up to 400 nodes.
- `StructureElementSimilarityI`: this interface allows the implementation of similarity functions between tree nodes. Each node contains a `StructureElement` reflecting the node information: a similarity function should return a similarity score that is a kernel itself, otherwise it is not guaranteed the convergence of several learning algorithms, such as Support Vector Machine. At the moment of writing two similarity function can be used within a SPTK.
  - The `ExactMatchingStructureElementSimilarity` is a similarity function returning 1 when the nodes are the same and 0 otherwise; by using this function, similar results w.r.t. the PTK should be obtained.

  - The `LexicalStructureElementSimilarity` is a similarity function used within syntactic trees derived from dependency/constituency parsing [Croce *et al.*(2012)]: according to this function, nodes reflecting syntactic or grammatical (i.e., pos-tag) information must be the same to contribute, while the similarity between lexical nodes is measured within a Word Space (e.g., [Croce and Previtali(2010)]) where words are represented as vectors and where the cosine distance reflects the semantic relatedness.

- `similarityThreshold` is a threshold applied to the similarity function, also used to speed up the kernel computation: node pairs whose score is below this threshold are ignored in the evaluation. We recommend to set this threshold to 0.001 in combination with the `LexicalStructureElementSimilarity` and the `ExactMatchingStructureElementSimilarity` functions.

Shortest Path Kernel

The Shortest Path Kernel associates a feature to each pair of nodes of one graph. The feature name corresponds to pair of node labels while the value is the length of the shortest path between the nodes in the graph. The complexity of the kernel is $\mathcal{O}(n^4)$, where n is the number of nodes in a graph. Further details can be found in [Borgwardt and Kriegel(2005)].
JAVA CLASS: `ShortestPathKernel`
MAVEN PROJECT: `kelp-additional-kernels`
JSON TYPE: `shortestPath`

4.3 Kernel Compositions

As mentioned in the beginning of this Section, kernels can be composed resulting in a new, valid kernel function. The composition of kernels is supported in KeLP through a specific abstract class called `KernelComposition`, that can be extended by a new kernel to realize the composition operation. Such operation is based on the computation provided by any other `Kernel` function, i.e., it enriches the kernel similarity computed by any other kernel. KeLP is designed to allow the composition of multiple levels of kernels. The `KernelComposition` abstract class contains two implemented methods, that are `getBaseKernel` and `setBaseKernel`, respectively the accessor methods for the base kernel that is composed by the current kernel.

In KeLP some available implementations of `KernelComposition` functions are:

Polynomial Kernel

The Polynomial Kernel is a kernel that implicitly works in a features space where all the polynomials of the original features are available. As an example, a 2nd degree polynomial kernel applied over a linear kernel over a vector representation will also consider in the computation the pairs of features. This can be easy demonstrated by expanding the following formula $poly_K(x, y) = \big(aK(x, y) + b\big)^d$, for $d = 2$.
JAVA CLASS: `PolynomialKernel`
MAVEN PROJECT: `kelp-core`
JSON TYPE: `poly`
PARAMETERS:

- `degree`: the degree of the polynomial;
- `a`: the *a* parameter in $poly_K(x, y) = \big(aK(x, y) + b\big)^d$;
- `b`: the *b* parameter in $poly_K(x, y) = \big(aK(x, y) + b\big)^d$.

RBF Kernel

The Radial Basis Function (RBF) Kernel computes the kernel function in an infinite dimensional implicit feature space. It enriches another kernel according to the following formula $RBF_K(x, y) = e^{-\gamma\|x-y\|^2_{\mathcal{H}_K}}$, where: $\|a\|_{\mathcal{H}_K}$ is the norm of a in the kernel space $\mathcal{H}_K$ generated by a base kernel K. $\|x - y\|^2_{\mathcal{H}_K}$ can be computed as $\|x - y\|^2_{\mathcal{H}_K} = \|x\|^2_{\mathcal{H}_K} + \|y\|^2_{\mathcal{H}_K} - 2K(x, y) = K(x, x) + K(y, y) - 2K(x, y)$.
JAVA CLASS: `RbfKernel`
MAVEN PROJECT: `kelp-core`
JSON TYPE: `rbf` PARAMETERS:

- `gamma`: the gamma parameter of the Gaussian kernel, according to $RBF_K(x, y) = e^{-\gamma\|x-y\|^2_{\mathcal{H}_K}}$.

Normalization Kernel

The Normalization Kernel allows to normalize a generic kernel, according to the formula $norm_K(x, y) = \frac{K(x,y)}{\sqrt{(K(x,x) \cdot K(y,y))}}$

JAVA CLASS: `NormalizationKernel`
MAVEN PROJECT: `kelp-core`
JSON TYPE: `norm`

4.3.1 Kernel Combinations

Kernel functions can be combined obtaining new valid kernel functions. The combination of kernel functions is supported in KeLP through a specific abstract class called `KernelCombination`, that can be extended by a new kernel to realize the desired combination operation. The combination operation is based on the computation provided by other base Kernel functions that are combined according to the specific function. The `KernelCombination` class contains two implemented methods, `getToCombine` and `setToCombine`, i.e., the accessor methods for the lists of the kernel to be combined. In KeLP some available implementations of the `KernelCombination` class are:

Linear Kernel Combination

The Linear Combination of Kernel function is computed as the weighted sum of kernel values, that are made available by the list of base kernels of the combination. The weights can be optionally normalized (by calling the method `normalizeWeights`, such that their sum is 1. The final kernel is then computed as the weighted linear combination of kernels K_i with $i = 1, \ldots, n$: $\sum_{i \leqslant n} c_i K_i$.
JAVA CLASS: `LinearKernelCombination`
MAVEN PROJECT: `kelp-core`
JSON TYPE: `linearComb`
PARAMETERS:

- `weights`: the weights associated to the linear combination.

Kernel Multiplication

The multiplication of Kernel functions is computed as the multiplication of kernel values that are computed starting from the base kernels. The `KernelMultiplication` executes the combination of kernels K_i with $i = 1, \ldots, n$ according to: $\prod_{i \leqslant n} K_i$.
JAVA CLASS: `KernelMultiplication`
MAVEN PROJECT: `kelp-core`
JSON TYPE: `multiplication`

4.3.2 Kernel On Pairs

Kernel on Pairs is an abstract class that serves as a generic interface for kernels operating on ExamplePairs, applying a simpler kernel BK to the elements within the pairs. As in `KernelComputation`, the simpler kernel corresponds to the class parameter `baseKernel`, that can be accessed using the methods `getBaseKernel` and `setBaseKernel`.

Preference Kernel

In the learning to rank scenario, the preference kernel [Shen and Joshi(2003)] compares two pairs of ordered objects $p_a = \langle a_1, a_2 \rangle$ and $p_b = \langle b_1, b_2 \rangle$:

$$PK(p_a, p_b) = BK(a_1, b_1) + BK(a_2, b_2) - BK(a_1, b_2) - BK(a_2, b_1) \quad (4.1)$$

where BK is a generic kernel operating on the elements of the pairs. The underlying idea is to evaluate whether the first pair $\langle a_1, a_2 \rangle$ aligns better to the second pair in its regular order $\langle b_1, b_2 \rangle$ rather than to its inverted order $\langle b_2, b_1 \rangle$.
JAVA CLASS: `PreferenceKernel`
MAVEN PROJECT: `kelp-core`
JSON TYPE: `preference`

Uncrossed Pairwise Sum Kernel

This kernel compares two pairs of ordered objects $p_a = \langle a_1, a_2 \rangle$ and $p_b = \langle b_1, b_2 \rangle$, summing the contributions of the single element similarities:

$$PK(p_a, p_b) = BK(a_1, b_1) + BK(a_2, b_2) \quad (4.2)$$

where BK is a generic kernel operating on the elements of the pairs. It has been used in learning scenarios where the elements within a pair have different roles, such as *text* and *hypothesis* in Recognizing Textual Entailment [Filice *et al.*(2015)], or *question* and *answer* in Question Answering [Severyn *et al.*(2013)].
JAVA CLASS: `UncrossedPairwiseSumKernel`
MAVEN PROJECT: `kelp-core`
JSON TYPE: `uncrossedPairwiseSum`

Uncrossed Pairwise Product Kernel

This kernel compares two pairs of ordered objects $p_a = \langle a_1, a_2 \rangle$ and $p_b = \langle b_1, b_2 \rangle$, multiplying the contributions of the single element similarities:

$$PK(p_a, p_b) = BK(a_1, b_1) \cdot BK(a_2, b_2) \tag{4.3}$$

where BK is a generic kernel operating on the elements of the pairs. It has been used in learning scenarios where the elements within a pair have different roles, such as *text* and *hypothesis* in Recognizing Textual Entailment [Filice *et al.*(2015)], or *question* and *answer* in Question Answering [Severyn *et al.*(2013)].
JAVA CLASS: `UncrossedPairwiseProductKernel`
MAVEN PROJECT: `kelp-core`
JSON TYPE: `uncrossedPairwiseProduct`

Pairwise Sum Kernel

This kernel compares two pairs of objects $p_a = \langle a_1, a_2 \rangle$ and $p_b = \langle b_1, b_2 \rangle$, summing the contributions of all pairwise similarities between the single elements:

$$PK(p_a, p_b) = BK(a_1, b_1) + BK(a_2, b_2) + BK(a_1, b_2) + BK(a_2, b_1) \tag{4.4}$$

where BK is a generic kernel operating on the elements of the pairs. It has been used in symmetric tasks, such as Paraphrase Identification [Filice *et al.*(2015)], where the elements within a pair are interchangeable.
JAVA CLASS: `PairwiseSumKernel`
MAVEN PROJECT: `kelp-core`
JSON TYPE: `pairwiseSum`

Pairwise Product Kernel

This kernel compares two pairs of objects $p_a = \langle a_1, a_2 \rangle$ and $p_b = \langle b_1, b_2 \rangle$, summing the contributions of the two possible pairwise alignments:

$$PK(p_a, p_b) = BK(a_1, b_1) \cdot BK(a_2, b_2) + BK(a_1, b_2) \cdot BK(a_2, b_1) \tag{4.5}$$

where BK is a generic kernel operating on the elements of the pairs. It has been used in symmetric tasks, such as Paraphrase Identification [Filice *et al.*(2015)], where the elements within a pair are interchangeable.

JAVA CLASS: `PairwiseProductKernel`
MAVEN PROJECT: `kelp-core`
JSON TYPE: `pairwiseProduct`

Best Pairwise Alignment Kernel

This kernel compares two pairs of objects $p_a = \langle a_1, a_2 \rangle$ and $p_b = \langle b_1, b_2 \rangle$, evaluating the best pairwise alignment:

$$SM(p_a, p_b) = softmax\Big(BK(a_1, b_1) \cdot BK(a_2, b_2), BK(a_1, b_2) \cdot BK(a_2, b_1)\Big) \quad (4.6)$$

where BK is a generic kernel operating on the elements of the pairs, and *softmax* is a function put in place of the max operation, which would cause SM not to be a valid kernel function (i.e., the resulting Gram matrix can violate the Mercer's conditions). In particular, *softmax*$(x_1, x_2) = \frac{1}{c}\log(\exp(cx_1) + \exp(cx_2)$ (c=100 is accurate enough). The Best Pairwise Alignment Kernel has been used in symmetric tasks, such as Paraphrase Identification [Filice *et al.*(2015)], where the elements within a pair are interchangeable.
JAVA CLASS: `BestPairwiseAlignmentKernel`
MAVEN PROJECT: `kelp-core`
JSON TYPE: `bestPairwiseAlignment`

4.4 Advanced Topics on Kernels

4.4.1 Defining new Kernels

As discussed in Section 4, kernels are organized into four main abstractions, i.e., `DirectKernel`, `KernelComposition`, `KernelCombination`, and `KernelOnPairs`. Therefore, when implementing a new kernel, the first step is understanding which abstract class we must extends. In this guide, we will describe how to implement a new `DirectKernel` and a new `KernelComposition`; the extensions to the other kernel types is straightforward.

Implementing a Direct Kernel: the Linear Kernel Example We are now assuming that the Linear Kernel LK is not available in KeLP, and that we need to implement it from scratch. The linear kernel is simply the dot product between two vectors $\vec{x_i}$

and $\vec{x_j}$, i.e., $LK(\vec{x_i}, \vec{x_i}) = \vec{x_i} \cdot \vec{x_j}$. Then, in implementing LinearKernel, we need to extend a `DirectKernel` over `Vector`. Optionally the class can be annotated with `@JsonTypeName` in order to specify an alternative type name to be used during the serialization/deserialization mechanism.

```
@JsonTypeName("linear")
public class LinearKernel extends DirectKernel {
```

To make the JSON serialization/deserialization mechanism work, an empty constructor must be defined:

```
        public LinearKernel() {
        }
```

Finally the `kernelComputation` method must be implemented:

```
        @Override
        protected float kernelComputation(Vector repA, Vector repB) {
                return repA.innerProduct(repB);
        }
```

Implementing a Kernel Composition: the RBF Kernel Example We are now assuming that the RBF Kernel RBF is not available in KeLP, and that we need to implement it from scratch. As described in Section 4.3, the RBF Kernel be generalized to any so that its computation depends on the result provided by a base kernel K:

$$RBF_K(x, y) = e^{-\gamma \|x-y\|^2_{\mathcal{H}_K}}$$

where: $\|a\|_{\mathcal{H}_K}$ is the norm of a in the kernel space $\mathcal{H}_K$ generated by a base kernel K. Therefore, the `RbfKernel` must extend `KernelComposition`. Optionally the class can be annotated with `@JsonTypeName` in order to specify an alternative type name to be used during the serialization/deserialization mechanism.

```
@JsonTypeName("rbf")
public class RbfKernel extends KernelComposition {
```

To make the JSON serialization/deserialization mechanism work, an empty constructor must be defined and all the kernel parameters must be associated to the corresponding getter and setter methods. In this case, `gamma` is the only parameter and the corresponding `getGamma` and `setGamma` methods must be implemented.

```
        private float gamma;

        public RbfKernel() {
```

```
	}

	/**
	 * @return the gamma
	 */
	public float getGamma() {
		return gamma;
	}

	/**
	 * @param gamma the gamma to set
	 */
	public void setGamma(float gamma) {
		this.gamma = gamma;
	}
```

Finally the `kernelComputation` method must be implemented containing the specific kernel similarity logic.

```
	@Override
	protected float kernelComputation(Example exA, Example exB) {
		float innerProductOfTheDiff = this.baseKernel
				.squaredNormOfTheDifference(exA, exB);
		return (float) Math.exp(-gamma *
		    innerProductOfTheDiff);
	}
```

4.4.2 Speeding up Kernel Machines through Caching Mechanisms

When adopting advanced kernel functions, such as convolution kernels, the kernel computation is the most computationally expensive part of both training and test phases. Usually the same kernel operation is required multiple times, therefore, a caching mechanism is required to dramatically speed up the algorithm. KeLP provides two kinds of cache:

- `SquaredNormCache` stores the squared norms of the instances in the RKHS, i.e., $K(x, x)$.
- `KernelCache` stores the kernel computations between instances, i.e., $K(x, y)$

A large cache allows to save many computations, however it may use a lot of memory: given a dataset of n instances, it can be useful to store in cache all the possible pairwise kernel computations, i.e., a total of $\frac{n(n+1)}{2}$ values. Both cache types are available

in KeLP with different implementations, which are optimized for different learning scenarios. All the caches are addressed using the `ExampleId`, which is a unique identifier that is associated to each `Example` during its instantiation (each example receives an `ExampleId` corresponding to the number of examples created so far).

`SquaredNormCache`: Several learning algorithms, such as the Passive Aggressive [Crammer *et al.*(2006)] during its updating procedure, explicitly require the computation of the squared norm of the training examples in the RKHS, i.e., $K(x, x)$. Furthermore, the squared norm in the RKHS of a base kernel appears in the formula of many kernel compositions, such as the Kernel Normalization described in section 4.3. Thus, caching these values is beneficial. The `SquaredNormCache` is the interface for a cache of the squared norms in the RKHS and its usage is shown in below:

```
PartialTreeKernel ptk = new PartialTreeKernel(0.4f, 0.4f, 1,
    "constTree");
NormalizationKernel normK = new NormalizationKernel(ptk);
int instances = 1000;
SquaredNormCache normCache = new FixIndexSquaredNormCache(instances);
ptk.setSquaredNormCache(SquaredNormCache);
```

The `SquaredNormCache` is implemented in two variants:

- `FixIndexSquaredNormCache`: It has a fix dimension that must be specified during the instantiation phase. Every example is statically assigned to a fix position in the cache that depends on its `ExampleId`. Collisions (i.e., multiple examples pointing at the same cache position) can occur: if two or more examples are assigned to the same position, only one can be stored in the cache. It is an optimal solution for storing norms when the cache size is large enough to contain all the examples in the dataset, or, more in general, when the examples to be stored have consecutive `exampleIds`.

- `DynamicIndexSquaredNormCache`: It has a fix dimension that must be specified during the instantiation phase. When the cache is full a last recently used strategy is applied for eliminating some entries. It is slightly slower and heavier than a `FixIndexSquaredNormCache` with the same size, however it is a better choice when only a subset of the examples should be stored.

In general, if the purpose is to cache all the possible squared norm values, then the best choice is `FixIndexSquaredNormCache`. In the next section it will be shown a particular situation where the `DynamicIndexSquaredNormCache` is more convenient.

KernelCache: There are several situations where the same kernel operations are required multiple times. Some examples are:

- during the training process of several algorithms, including SVM, or multi-epoch online learning algorithms;
- when binary classifiers are combined for solving a multi-class problems;
- when a n-fold cross validation is performed;
- during a tuning stage, when the same kernel is employed in different learning algorithms.

Therefore using a proper caching mechanism is crucial, especially when the adopted kernel is computationally expensive, as in the case of convolution kernels. The listing below illustrates how to equip a kernel with a `KernelCache`:

```
UncrossedPairwiseProductKernel pairwiseK = new
    UncrossedPairwiseProductKernel(normK, false);
int examplePairs = 1000;
KernelCache kernelCache = new DynamicIndexKernelCache(examplePairs);
pairwiseK.setKernelCache(kernelCache);
```

The `SquaredNormCache` is implemented in three variants:

- `FixIndexKernelCache`: it stores all the pairwise kernel computations of a set of n examples, where n is a fix dimension that must be specified during the instantiation phase. Basically it stores a symmetric matrix where every example is statically assigned to a fix row/column that depends on its `ExampleId`. Collisions (i.e., multiple examples pointing at the same row/column) can occur: if two or more examples are assigned to the same index, only one can be stored in the cache. It is an optimal solution for storing kernel computations when the cache size is large enough to avoid collisions, or, more in general, when the examples to be stored have consecutive `exampleId`s.

- `DynamicIndexKernelCache`: it stores all the pairwise kernel computations of a set of n examples, where n is a fix dimension that must be specified during the instantiation phase. When the cache is full a last recently used strategy is applied for eliminating some entries. It is slightly slower and heavier than a `FixIndexKernelCache` with the same size, however it is a better choice when only the pairwise kernel computations of a subset of the examples should be stored.

- `StripeKernelCache`: Given a dataset, this cache stores kernel computations in "stripes", i.e., whole rows of the complete Gram Matrix. In other words given a subset S of the examples in the dataset D, the cache is able to store all the kernel computations between any example in S and any example in D. This kind of cache is efficient when there is a fix split between the training set and test set, therefore no kernel operations are required between the instances in the testset. Another efficient usage is in the training phase of particular binary learning algorithms, such as the SVM implementation proposed in [Chang and Lin(2011)], that are optimized for reducing the required number of kernel operations to few "stripes" of the kernel gram matrix. When the number of stripes is exceeded, they are removed according to a FIFO policy.

To clarify the differences between the `FixIndex` versions and the `DynamicIndex` counterparts of the `SquaredNormCache` and `KernelCache`, an example is hereafter proposed. Assume instances have the form of an `ExamplePair` where the left and right parts are `SimpleExamples` having a `TreeRepresentation`. In this case, for each example, three `exampleIds` will be assigned: one to the left part, one to the right part, and finally one to the overall `ExamplePair`. The listings above define the following kernel over pairs: $N^{+}_{PTK}(p_a, p_b) = N_{PTK}(a_1, b_1) + N_{PTK}(a_2, b_2) = \frac{PTK(a_1,b_1)}{\sqrt{PTK(a_1,a_1)PTK(b_1,b_1)}} + \frac{PTK(a_2,b_2)}{\sqrt{PTK(a_2,a_2)PTK(b_2,b_2)}}$, where each kernel is the sum of the normalized PTK between the left trees and the normalized PTK between the right trees. In this case, for each overall kernel computation, four squared norms in the RKHS of the PTK are required (i.e., $PTK(x, x)$), and caching them can significantly reduce the computational time. Given a dataset of n example pairs, a total of $2n$ trees occur. A $2n$ size `DynamicIndexSquaredNormCache` can store all the resulting $2n$ squared norms. Instead, the `FixIndexSquaredNormCache` requires a $3n$ size to avoid collisions, as the total number of `exampleIds` is $3n$, considering the overall `ExamplePairs`.

For the same reason, if a `KernelCache` is needed for the overall kernel, it is better to use a `DynamicIndexKernelCache` than a `FixIndexKernelCache`, as the former requires only a size n, i.e., the total number of `ExamplePairs`, while the latter needs $3n$ to avoid collisions[2].

[2]in this case the size of a kernel cache expresses the total number of examples whose pairwise computations can be simultaneously stored, and not the actual memory occupation, which is quadratic of such number.

5
Learning Algorithms

In Machine Learning, a wide plethora of learning algorithms have been defined for different purposes, and many variations of the existing ones as well as completely new learning methods are often proposed. KeLP provides a large number of learning algorithms[1] that can be used to tackle a variety of learning scenarios, including (multi-class, multi-label) classification, regression and clustering.

Many algorithms are implemented as a linear version to favor efficiency and allow to explore very large datasets; other algorithms have a kernel-based implementation that exploits the expressiveness and efficacy of kernel-methods. Furthermore KeLP provides both Online Learning (e.g., Perceptron-like learners) and Batch Learning models (e.g., Support Vector Machines). In Batch Learning, the complete training dataset is supposed to be entirely available during the learning phase, and all the examples are exploited at the same time (usually optimizing a global objective function). In Online Learning individual examples are exploited one at a time to incrementally acquire the model.

5.1 Learning Algorithms Organization

KeLP proposes a comprehensive set of interfaces to support the implementation of new algorithms. Figure 5.1 illustrates a simplified class diagram of the algorithms in KeLP . Three main objects can be identified:

- `LearningAlgorithm`: It contains the learning procedure that is applied on training data to generate a `PredictionFunction`; for instance LibLinear-LearningAlgorithm is a binary classification learning algorithm which imple-

[1] All the algorithms are completely re-implemented in Java and they do not wrap any external library

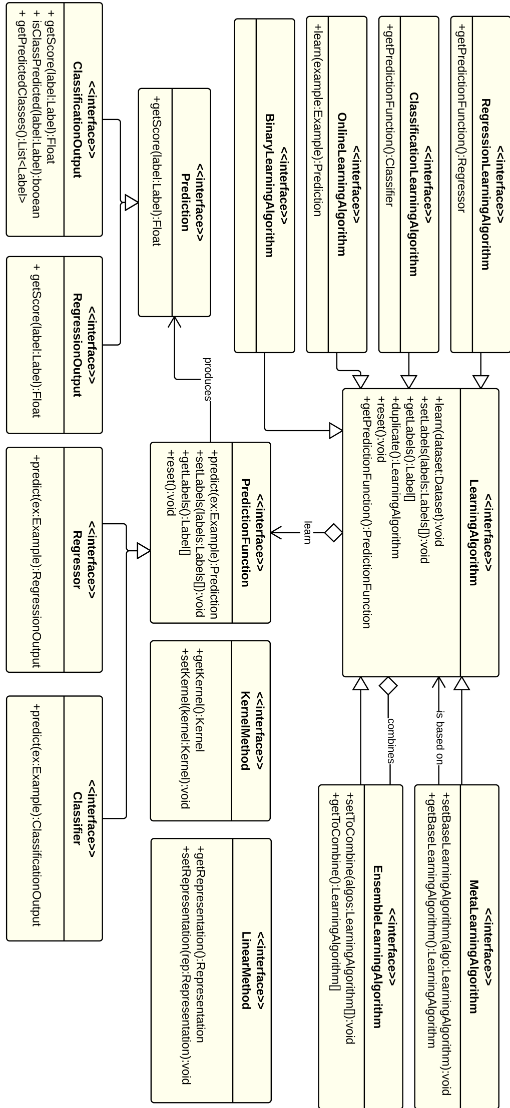

Figure 5.1: KeLP algorithm class diagram

ments the linear SVM algorithm described in [Fan *et al.*(2008)]: given a training dataset it finds the best separating hyperplane between the two classes.

- `PredictionFunction`: It contains the logic to produce a `Prediction` on a new data instance. For example, `BinaryLinearClassifier` classifies a new instance performing a dot product with the classification hyperplane.

- `Prediction`: it is the output associated to a test data. For instance, BinaryMarginClassifierOutput is the predicted label and score associated to a test data on a binary classification task.

Among the `LearningAlgorithm` interfaces, the most relevant are:

- `ClassificationLearningAlgorithm`: for algorithms that learn from labelled data how to classify new instances. Some of the current implementations are C-SVM, ν-SVM [Chang and Lin(2011)] and Pegasos [Shalev-Shwartz *et al.*(2007)].

- `OnlineLearningAlgorithm`: it support the definition of online learning algorithms, like the Perceptron [Rosenblat(1958)] or the Passive Aggressive algorithms, [Crammer *et al.*(2006)].

- `RegressionLearningAlgorithm`: for algorithms that learn from labelled data a regression function. For instance, ϵ-SVR [Chang and Lin(2011)], Linear and Kernelized Passive Aggressive Regression [Crammer *et al.*(2006)].

- `MetaLearningAlgorithm`: for learning approaches that rely on another algorithm. We implemented Multi-classification schemas, e.g., One-VS-One, One-VS-All and Multi-label classification, as well as budgeted policies of online learning algorithms like Randomized Budget Perceptron [Cesa-Bianchi and Gentile(2006)] and Stoptron [Orabona *et al.*(2008)].

- `EnsembleLearningAlgorithm`: for algorithms that combine other algorithms in order to provide a more robust solution.

Algorithms exploiting kernel functions must implement `KernelMethod`, while algorithms operating directly in an explicit feature space must implement `LinearMethod`.

In addition, the `ClusteringAlgorithm` interface models algorithms that are able to organize a dataset into clusters. Currently KeLP implements kernel-based clustering from [Kulis *et al.*(2005)].

5.2 Available Learning Algorithms and their JSON Definition

This sections provides a description of the learning algorithms implemented within KeLP , grouped according to the following criteria:

- in Section 5.2.1 batch learning algorithms for classification and regression tasks are reported;
- in Section 5.2.2 online learning algorithms for classification and regression tasks are reported;
- in Section 5.2.3 algorithms for clustering are reported;
- in Section 5.2.4 algorithms for linearizing a complex RKHS, such as the Nystrom method, are reported;
- in Section 5.2.5 meta learning algorithms are reported to implement complex schemas, such as One-Vs-All and One-Vs-One multi-class classifiers; moreover, several extensions to online algorithms are reported (e.g., the MultiEpoch online learning algorithm); some budgeted methods are included to scale up online kernel-based learning over large datasets.

For each learning algorithm the corresponding Java class, KeLP project, JSON Type and main parameters are reported.

Learning algorithms and representations. Each learning algorithm in KeLP operates on specific representations reflecting the examples. In kernel-based method the targeted representations are specified directly into the adopted kernel functions. In learning methods operating directly on the explicit spaces the name of the representations must be expressed within the parameters of the algorithm implementations.

Fairness and learning algorithms. In many real-world classification domains, the examples are not equally distributed among the classes, and dealing with very imbalanced datasets can make Machine Learning algorithms produce a weak classification hypothesis that largely penalizes the less frequent classes. In KeLP most of learning algorithms implement a strategy called **fairness** that consists of directly modifying the learning algorithm in order to emphasize the contribution of the less frequent examples. In [Morik *et al.*(1999)] the original SVM formulation is slightly modified splitting the empirical risk term into a positive part and a negative one. Each example is associated

to parameters C_+ and C_- that substitute the original regularization parameter C. We adopted in KeLP the strategy proposed in [Filice *et al.*(2014)] to guarantee the fairness among positive and negative examples both in batch and online learning algorithms for classification tasks: the parameters C_+ and C_- are chosen so that the potential total cost of the false positive equals the potential total cost of the false negative, i.e., the ratio C_+/C_- is equal to the ratio between the number of negative and positive training examples.

5.2.1 Batch Learning Algorithms

The following batch learning algorithms are implemented.

Binary C-SVM Classification

It is the KeLP implementation of C-Support Vector Machine learning algorithm [Cortes and Vapnik(1995)]. It is a learning algorithm for binary classification and it relies on kernel functions. It is a Java porting of the library LIBSVM v3.17, written in C++ [Chang and Lin(2011)]
JAVA CLASS: `BinaryCSvmClassification`
MAVEN PROJECT: `kelp-core`
JSON TYPE: `binaryCSvmClassification`
PARAMETERS:

- `kernel`: The kernel function
- `label`: The label to be learned
- `cp`: The regularization parameter for positive examples
- `cn`: The regularization parameter for negative examples
- `useFairness`: A boolean parameter to force the fairness policy

Binary ν-SVM Classification

It is the KeLP implementation of the ν-Support Vector Machine learning algorithm [Schölkopf *et al.*(2000)]. It is a learning algorithm for binary classification and it relies on kernel functions. It is a Java porting of the library LIBSVM v3.17, written in C++ [Chang and Lin(2011)].
JAVA CLASS: `BinaryNuSvmClassification`
MAVEN PROJECT: `kelp-core`
JSON TYPE: `binaryNuSvmClassification`
PARAMETERS:

- `kernel`: The kernel function
- `label`: The label to be learned
- `nu`: The ν parameter, i.e., the percentage of training example to be used as Support Vectors
- `useFairness`: A boolean parameter to force the fairness policy

One Class Svm Classification

It is KeLP implementation of One-Class Support Vector Machine learning algorithm [Schölkopf *et al.*(2001)]. It is a learning algorithm for estimating the Support of a High-Dimensional Distribution and it relies on kernel functions. The model is acquired only by considering positive examples. It is useful in anomaly detection (a.k.a. novelty detection). It is a Java porting of the library LIBSVM v3.17, written in C++ [Chang and Lin(2011)]
JAVA CLASS: `OneClassSvmClassification`
MAVEN PROJECT: `kelp-core`
JSON TYPE: `oneClassSvmClassification`
PARAMETERS:

- `kernel`: The kernel function
- `label`: The label to be learned
- `nu`: The ν parameter, i.e., the percentage of training example to be used as Support Vectors

LibLinear SVM Classification

This class implements linear SVMs models trained using a coordinate descent algorithm [Fan *et al.*(2008)]. It operates in an explicit feature space (i.e., it does not rely on any kernel). This code has been adapted from the Java port of the original LIBLINEAR C++ sources.
JAVA CLASS: `LibLinearLearningAlgorithm`
MAVEN PROJECT: `kelp-additional-algorithms`
JSON TYPE: `liblinear`

PARAMETERS:

- `label`: The label to be learned

- `cp`: The regularization parameter for positive examples
- `cn`: The regularization parameter for negative examples
- `representation`: The identifier of the representation to be considered for the training step

Pegasos Classification

It implements the Primal Estimated sub-GrAdient SOlver (PEGASOS) for SVM. It is a learning algorithm for binary linear classification Support Vector Machines. It operates in an explicit feature space (i.e., it does not rely on any kernel). Further details can be found in [Shalev-Shwartz *et al.*(2007)].
JAVA CLASS: `PegasosLearningAlgorithm`
MAVEN PROJECT: `kelp-additional-algorithms`
JSON TYPE: `pegasos`

PARAMETERS:

- `label`: The label to be learned
- `lambda`: The regularization coefficient
- `iterations`: The number of iterations required from the learning algorithm
- `k`: The number of examples k that PEGASOS exploits in its mini-batch learning approach
- `representation`: The identifier of the representation to be considered for the training step

A Dual Coordinate Descent Learning Algorithm

the KeLP implementation of Dual Coordinate Descent (DCD) training algorithm for a Linear L^1 or L^2 Support Vector Machine for binary classification [Hsieh *et al.*(2008)].
JAVA CLASS: `DCDLearningAlgorithm`
MAVEN PROJECT: `kelp-additional-algorithms`
JSON TYPE: `dcd`

PARAMETERS:

- `label`: The label to be learned

- `dcdLoss`: The considered Loss function (L^1 or L^2)
- `useBias`: This boolean parameter determines the use of bias b in the classification function $f(x) = wx + b$. If usebias is set to `false` the bias is set to 0.
- `cp`: The regularization parameter for positive examples
- `cn`: The regularization parameter for negative examples
- `maxIterations`: The number of iterations required from the learning algorithm
- `representation`: The identifier of the representation to be considered for the training step

ϵ-Support Vector Regression

It implements the ϵ-Support Vector Regression learning algorithm. It is a learning algorithm for linear regression based on Support Vector Machines [Vapnik(1998)]. It relies on kernel functions. It is a Java porting of the library LIBSVM v3.17, written in C++ [Chang and Lin(2011)].
JAVA CLASS: `EpsilonSvmRegression`
MAVEN PROJECT: `kelp-core`
JSON TYPE: `epsilonSvmRegression`
PARAMETERS:

- `kernel`: The kernel function
- `pReg`: The regularization parameter for positive examples
- `c`: The regularization parameter

LibLinear Regression

This class implements linear SVM regression trained using a coordinate descent algorithm [Fan *et al.*(2008)]. It operates in an explicit feature space (i.e., it does not rely on any kernel). This code has been adapted from the Java port of the original LIBLINEAR C++ sources.
JAVA CLASS: `LibLinearRegression`
MAVEN PROJECT: `kelp-additional-algorithms`
JSON TYPE: `liblinearregression`
PARAMETERS:

- `p`: The ϵ in the loss function of SVR (default 0.1)
- `c`: The regularization parameter
- `representation`: The identifier of the representation to be considered for the training step

5.2.2 Online Learning Algorithms

Linear Passive Aggressive (PA) Classification

Online Passive-Aggressive Learning Algorithm for classification tasks (linear version, presented in [Crammer *et al.*(2006)] and extended in [Filice *et al.*(2014)]). Every time an example is misclassified it is added the current hyperplane, with the weight that solves the passive aggressive minimization problem.
JAVA CLASS: `LinearPassiveAggressiveClassification`
MAVEN PROJECT: `kelp-additional-algorithms`
JSON TYPE: `linearPA`
PARAMETERS:

- `label`: The label to be learned
- `loss`: The loss function to weight each misclassification
- `policy`: The updating policy applied by the Passive Aggressive Algorithm when a miss-prediction occurs
- `cp`: The aggressiveness parameter for positive examples
- `cn`: The aggressiveness parameter for negative examples
- `useFairness`: A boolean parameter to force the fairness policy
- `representation`: The identifier of the representation to be considered for the training step

Kernel-based Passive Aggressive (PA) Classification

Online Passive-Aggressive Learning Algorithm for classification tasks with Kernels (presented in [Crammer *et al.*(2006)] and extended in [Filice *et al.*(2014)])). Every time an example is misclassified it is added as support vector, with the weight that solves the passive aggressive minimization problem.

JAVA CLASS: `KernelizedPassiveAggressiveClassification`
MAVEN PROJECT: `kelp-additional-algorithms`
JSON TYPE: `kernelizedPA`
PARAMETERS:

- `label`: The label to be learned
- `kernel`: The kernel function
- `loss`: The loss function to weight each misclassification
- `policy`: The updating policy applied by the Passive Aggressive Algorithm when a miss-prediction occurs
- `cp`: The aggressiveness parameter for positive examples
- `cn`: The aggressiveness parameter for negative examples
- `useFairness`: A boolean parameter to force the fairness policy

Linear Perceptron

The perceptron learning algorithm for classification tasks (linear version, presented in [Rosenblat(1958)]).
JAVA CLASS: `LinearPerceptron`
MAVEN PROJECT: `kelp-additional-algorithms`
JSON TYPE: `linearPerceptron`
PARAMETERS:

- `label`: The label to be learned
- `alpha`: The learning rate, i.e., the weight associated to misclassified examples during the learning process
- `margin`: The minimum distance from the hyperplane that an example must have in order to be not considered misclassified
- `unbiased`: This boolean parameter determines the use of bias b in the classification function $f(x) = wx + b$. If usebias is set to `true` the bias is set to 0.
- `representation`: The identifier of the representation to be considered for the training step

Kernel-based Perceptron

The perceptron learning algorithm for classification tasks (Kernel machine version, presented in [Rosenblat(1958)]).
JAVA CLASS: `KernelizedPerceptron`
MAVEN PROJECT: `kelp-additional-algorithms`
JSON TYPE: `kernelizedPerceptron`
PARAMETERS:

- `kernel`: The kernel function
- `label`: The label to be learned
- `alpha`: The learning rate, i.e., the weight associated to misclassified examples during the learning process
- `margin`: The minimum distance from the hyperplane that an example must have in order to be not considered misclassified
- `unbiased`: This boolean parameter determines the use of bias b in the classification function $f(x) = wx + b$. If usebias is set to `true` the bias is set to 0.

Soft Confidence Weighted Classification

Implements Exact Soft Confidence-Weighted (SCW) algorithms, an on-line learning algorithm for binary classification [Wang *et al.*(2012)]. This class implements both the SCW-I and SCW-II variants.
JAVA CLASS: `SoftConfidenceWeightedClassification`
MAVEN PROJECT: `kelp-additional-algorithms`
JSON TYPE: `scw`
PARAMETERS:

- `label`: The label to be learned
- `scwType`: The type of SCW learning algorithm (SCW-I or SCW-II)
- `eta`: The probability of correct classification required for the updated distribution on the current instance
- `cp`: The regularization parameter for positive examples
- `cn`: The regularization parameter for negative examples

- `useFairness`: A boolean parameter to force the fairness policy
- `representation`: The identifier of the representation to be considered for the training step

Linear Passive Aggressive (PA) Regression

Online Passive-Aggressive Learning Algorithm for regression tasks (linear version, proposed in [Crammer *et al.*(2006)]).
JAVA CLASS: `LinearPassiveAggressiveRegression`
MAVEN PROJECT: `kelp-additional-algorithms`
JSON TYPE: `linearPA-R`
PARAMETERS:

- `policy`: The updating policy applied by the Passive Aggressive Algorithm when a miss-prediction occurs
- `c`: The aggressiveness parameter
- `eps`: The accepted distance between the predicted and the real regression values
- `representation`: The identifier of the representation to be considered for the training step

Kernel-based Passive Aggressive (PA) Regression

Online Passive-Aggressive Learning Algorithm for regression tasks (kernel-based version, proposed in [Crammer *et al.*(2006)]).
JAVA CLASS: `KernelizedPassiveAggressiveRegression`
MAVEN PROJECT: `kelp-additional-algorithms`
JSON TYPE: `kernelizedPA-R`
PARAMETERS:

- `kernel`: The kernel function
- `policy`: The updating policy applied by the Passive Aggressive Algorithm when a miss-prediction occurs
- `c`: The aggressiveness parameter
- `eps`: The accepted distance between the predicted and the real regression values

5.2.3 Clustering Algorithms

The following algorithms for Clustering have been implemented:

K-means

Implements the K-means Clustering Algorithm, that works on an Explicit feature Space
JAVA CLASS: `LinearKMeansEngine`
MAVEN PROJECT: `kelp-core`
JSON TYPE: `kmeans`
PARAMETERS:

- `k`: The number of expected clusters
- `maxIterations`: The maximum number of iterations
- `representation`: The identifier of the representation to be considered

Kernel-based K-means

Implements the Kernel Based K-means described in [Kulis *et al.*(2005)]
JAVA CLASS: `KernelBasedKMeansEngine`
MAVEN PROJECT: `kelp-additional-algorithms`
JSON TYPE: `kernelbased_kmeans`
PARAMETERS:

- `kernel`: The kernel function
- `k`: The number of expected clusters
- `maxIterations`: The maximum number of iterations

5.2.4 Linearization Functions

The following linearization functions have been implemented to linearized examples through linear representations, i.e., vectors:

Nystrom Method

This class implements the Nystrom Method to approximate the implicit space underlying a Kernel Function, thus producing a low-dimensional dense representation as discussed in [Williams and Seeger(2001)] and applied to some of the kernels presented

here in [Croce and Basili(2016)]. As an example, given a `Dataset` of examples represented through tree structures and a tree kernel function, this class allows deriving a linearized dataset at a given dimensionality.
JAVA CLASS: `NystromMethod`
MAVEN PROJECT: `kelp-additional-algorithms`
JSON TYPE: `nystrom`
PARAMETERS:

- `kernel`: The kernel function
- `landmarks`: The examples used as landmarks
- `rank`: The expected rank of the space representing the linearized examples

5.2.5 Meta-learning Algorithims

In this Section, meta learning algorithms are reported to implement complex schemas, such as One-Vs-All and One-Vs-One multi-class classifiers; moreover several extensions to online algorithms are reported (e.g., the MultiEpoch online learning algorithm); some budgeted methods are included to scale up online kernel-based learning over large datasets.

One-Vs-All strategy from multi-class classification

It is a meta algorithm that operates applying a One-Vs-All strategy over the base learning algorithm which is intended to be a binary learner. The One-Vs-All strategy will learn N different classifiers, where N is the number of classes involved in the dataset. In this strategy each classifier is learned by considering in turn the examples of a single class as positives, while all the other examples are considered as negative.
NOTE: the base learning algorithm must provide a duplicate method which properly works
JAVA CLASS: `OneVsAllLearning`
MAVEN PROJECT: `kelp-core`
JSON TYPE: `oneVsAll`
PARAMETERS:

- `baseAlgorithm`: The base learning algorithm which is intended to be a binary learner
- `labels`: The list of targeted classes

One-Vs-One strategy from multi-class classification

It is a meta algorithm that operates by applying a One-Vs-One strategy over the base learning algorithm which is intended to be a binary learner. This meta-algorithms will learn $N(N-1)/2$ classifiers, by comparing each class with all the others separately. The resulting classifier applies a voting strategy to perform the final decision. (N is the number of classes in the dataset)
NOTE: the base learning algorithm must provide a duplicate method which properly works
JAVA CLASS: `OneVsOneLearning`
MAVEN PROJECT: `kelp-core`
JSON TYPE: `oneVsOne`
PARAMETERS:

- `baseAlgorithm`: The base learning algorithm which is intended to be a binary learner
- `labels`: The list of targeted classes

Multi Epoch Online Learning

It is a meta learning algorithms for online learning methods. It performs multiple iterations (or *epochs*) on the training data.
JAVA CLASS: `MultiEpochLearning`
MAVEN PROJECT: `kelp-additional-algorithms`
JSON TYPE: `multiEpoch`
PARAMETERS:

- `baseAlgorithm`: The base *online* learning algorithm which is intended to apply for multiple iterations
- `epochs`: The number of iterations

Budgeted Online Learning: the Stoptron algorithm

It is a variation of the Stoptron proposed in [Orabona *et al.*(2008)]. Until the budget is not reached the online learning updating policy is the one of the baseAlgorithm that this meta-algorithm is exploiting. When the budget is full, the learning process ends.
JAVA CLASS: `Stoptron`
MAVEN PROJECT: `kelp-additional-algorithms`
JSON TYPE: `stoptron`
PARAMETERS:

- `label`: The label to be learned
- `baseAlgorithm`: The base *online* learning algorithm which is intended to apply in this budgeted approach
- `budget`: The maximum number of support vectors allowed in the budget

Budgeted Online Learning: the Randomized Budgeted Perceptron

A variant of the Randomized Budget Perceptron proposed in [Cavallanti *et al.*(2007)]. Until the budget is not reached the online learning updating policy is the one of the baseAlgorithm that this meta-algorithm is exploiting. When the budget is full, a random support vector is deleted and the perceptron updating policy is adopted.
JAVA CLASS: `RandomizedBudgetPerceptron`
MAVEN PROJECT: `kelp-additional-algorithms`
JSON TYPE: `randomizedPerceptron`
PARAMETERS:

- `label`: The label to be learned
- `baseAlgorithm`: The base *online* learning algorithm which is intended to apply in this budgeted approach
- `budget`: The maximum number of support vectors allowed in the budget

Budgeted Online Learning: Budgeted Passive Aggressive Classification

It is the implementation of the Budgeted Passive Aggressive Algorithm proposed in [Wang and Vucetic(2010)]. When the budget is full, the schema proposed in [Wang and Vucetic(2010)] to update examples (and weights) is adopted.
JAVA CLASS: `BudgetedPassiveAggressiveClassification`
MAVEN PROJECT: `kelp-additional-algorithms`
JSON TYPE: `budgetedPA`
PARAMETERS:

- `label`: The label to be learned
- `label`: The label to be learned
- `kernel`: The kernel function
- `loss`: The loss function to weight each misclassification

- `policy`: The updating policy applied by the Passive Aggressive Algorithm when a miss-prediction occurs
- `deletingPolicy`: The policy for the removal of examples from the budget before adding new examples. This can be
  - `BPA_S`: Budgeted Passive Aggressive Simple: when a new support vector must be added, one is removed and the weight of the other support vectors is kept unchanged
  - `BPA_1NN`: Budgeted Passive Aggressive Nearest Neighbor: when a new support vector must be added, one is removed and the weight of its nearest neighbor is adjusted
- `cp`: The aggressiveness parameter for positive examples
- `cn`: The aggressiveness parameter for negative examples
- `useFairness`: A boolean parameter to force the fairness policy
- `budget`: The maximum number of support vectors allowed in the budget

5.3 Evaluators

In Machine Learning is often necessary to evaluate the performances of a classification or regression function that is derived through a learning process. Generally, it means measuring performance indicators over a test dataset. These performances can be then used to decide whether the learning algorithm with its parameterization is good enough for the task of interests. This is a pattern that is repeated every time a new experiment is necessary.

Performance measures are the same for many different tasks, thus their computation can be easily standardized in order to support many scenarios. In KeLP, we provided the `Evaluator` abstract class, which serves as a base class for other performance evaluations classes. The `Evaluator` class contains a public implemented method, whose name is `getPerformanceMeasure(String, Object...)` that will access the internal class methods by means of Java reflection to return a performance measure. So for example, if a specific implementation of an evaluator offers the method to compute the accuracy, whose name is `getAccuracy()`, then the `getPerformanceMeasure` can be invoked as `getPerformanceMeasure("Accuracy")`. It serves as a general interface to retrieve the performance measures computed by a specific evaluators. Notice that an evaluator must contain methods whose name is

"get{MeasureName}" to be compliant to the getPerformanceMeasure mechanism. This implementation pattern is necessary to support the generic instantiation of an evaluator in the case automatic classes for experiments will be provided.

The Evaluator class contains 4 abstract methods that should be implemented by its sub-classes in order to respect the Evaluator contract. The four abstract methods of the Evaluator class are:

- addCount(Example, Prediction): it is the main interface with which an external program will call the evaluator to add the Prediction of an Example to the *counts* that will be adopted to compute the final performance measures. Notice that KeLP does not force to adopt any particular internal mechanisms for computing the performances;
- compute(): this method is called internally by getPerformanceMeasures method to force the computation of all the performances;
- clear(): it serves, eventually, to reset the evaluator;
- duplicate(): it should implement a method to duplicate the evaluator.

In KeLP some implementation of Evaluators are available: BinaryClassificationEvaluator, MulticlassClassificationEvaluator and RegressorEvaluator. They are intended to satisfy the major needs when dealing with binary classification tasks, multi-class classification tasks and regression tasks.

Binary Evaluation

This is an instance of an Evaluator and it allows to compute some common performance measures for binary classification tasks. In particular, this evaluator contains methods to compute: *Precision*, *Recall*, F_1 and *Accuracy*.
JAVA CLASS: BinaryClassificationEvaluator
MAVEN PROJECT: kelp-core

Multi-class Evaluation

This is an instance of an Evaluator and it is an extension of the BinaryClassificationEvaluator to the multi-class case. It computes *Precision*, *Recall* and F_1 for each class, and a global *Accuracy* measure.
JAVA CLASS: MulticlassClassificationEvaluator
MAVEN PROJECT: kelp-core

Regressor Evaluation

This is an instance of an `Evaluator` and it allows to compute some measure for regression tasks. In particular, this evaluator computes the *mean squared error*.
JAVA CLASS: `RegressorEvaluator`
MAVEN PROJECT: `kelp-core`

5.4 Advanced Topics on Learning Algorithms

5.4.1 Defining new Learning Algorithms

As discussed in Section 5.1, in KeLP , there is a comprehensive taxonomy for learning algorithms, that should be taken into account when implementing new algorithm. In this guide, we will describe how to implement a new linear algorithm for binary classification tasks. The concepts highlighted can be easily extended to implement different types of learning algorithms.

Implementing a new learning algorithm: the Pegasos Example We are now assuming that the Pegasos Learning Algorithm is not available in KeLP , and that we need to implement it from scratch. Pegasos is an efficient solver for linear SVM for binary classification tasks that uses a mini-batch approach; therefore, we need to implement `ClassificationLearningAlgorithm`, `BinaryLearningAlgorithm` and `LinearMethod`. Optionally, the class can be annotated with `@JsonTypeName` in order to specify an alternative name to be used during the JSON/XML serialization/deserialization mechanism.

```
@JsonTypeName("pegasos")
public class PegasosLearningAlgorithm implements LinearMethod,
    ClassificationLearningAlgorithm, BinaryLearningAlgorithm{
```

KeLP decouples learning algorithms from prediction functions, which are used to provide predictions on new data. Regarding `PegasosLearningAlgorithm`, the proper prediction function is `BinaryLinearClassifier`, that is already implemented in the platform. Then, we have to add a new corresponding parameter, i.e., `classifier`. An empty constructor must be defined and all the learning parameters must be associated to the corresponding getter and setter methods, in order to make the JSON serialization/deserialization mechanism work. In this case, the learning parameters are the regularization coefficient `lambda`, the number of iterations `T`, the number of examples `k` exploited during each mini-batch iteration.

```
        private BinaryLinearClassifier classifier;
```

```
        private int k = 1;
        private int iterations = 1000;
        private float lambda = 0.01f;

        /**
         * Returns the number of examples k that Pegasos exploits in
             its
         * mini-batch learning approach
         *
         * @return k
         */
        public int getK() {
                return k;
        }

        /**
         * Sets the number of examples k that Pegasos exploits in its
         * mini-batch learning approach
         *
         * @param k the k to set
         */
        public void setK(int k) {
                this.k = k;
        }

        /**
         * Returns the number of iterations
         *
         * @return the number of iterations
         */
        public int getIterations() {
                return iterations;
        }

        /**
         * Sets the number of iterations
         *
         * @param T the number of iterations to set
         */
        public void setIterations(int T) {
                this.iterations = T;
        }

        /**
         * Returns the regularization coefficient
         *
         * @return the lambda
         */
        public float getLambda() {
```

```
                return lambda;
        }

        /**
         * Sets the regularization coefficient
         *
         * @param lambda the lambda to set
         */
        public void setLambda(float lambda) {
                this.lambda = lambda;
        }

        public PegasosLearningAlgorithm(){
                this.classifier = new BinaryLinearClassifier();
                this.classifier.setModel(new BinaryLinearModel());
        }
```

According to the selected interfaces some specific methods have to be implemented. As any `LinearMethod` we need to implement `setRepresentation` and `getRepresentation`, which refer to the `String` identifier for the specific `Representation` the algorithm must exploit. Obviously, a corresponding parameter must created, i.e., representation

```
        private String representation;

        @Override
        public String getRepresentation() {
                return representation;
        }

        @Override
        public void setRepresentation(String representation) {
                this.representation = representation;
                BinaryLinearModel model = this.classifier.getModel();
                model.setRepresentation(representation);
        }
```

As any `BinaryLearningAlgorithm` we need to implement `getLabel` and `setLabel`, which refer to the label that must be considered as positive class. Obviously, a corresponding parameter must created, i.e., `label`. Moreover, to be compliant with the `LearningAlgorithm` interface, the methods `getLabels` and `setLabels` must be implemented that, for the special case of `BinaryLearningAlgorithm` must operated on a single entry `List`:

```
        private Label label;

        @Override
```

```
	public void setLabels(List<label> labels){
		if(labels.size()!=1){
			throw new IllegalArgumentException("Pegasos
			    algorithm is a binary method which can
			    learn a single Label");
		}
		else{
			this.label=labels.get(0);
			this.classifier.setLabels(labels);
		}
	}

	@Override
	public List<label> getLabels() {
		return Arrays.asList(label);
	}

	@Override
	public Label getLabel(){
		return this.label;
	}

	@Override
	public void setLabel(Label label){
		this.setLabels(Arrays.asList(label));
	}
```

Finally, as any `ClassificationLearningAlgorithm` we need to implement the following methods:

- `getPredictionFunction`: the proper prediction function must be returned, in this case the classifier object. It is a good practice to specialize the returning type, i.e., the method must return a `BinaryLinearClassifier` instead of a generic `Classifier`;
- `duplicate`: the instance of an algorithm must be created and returned, copying all the learning parameters (the state variables must not be copied, leaving them to their default value). It is a good practice to specialize the returning type, i.e., the method must return a `PegasosLearningAlgorithm` instead of a generic `LearningAlgorithm`;
- `reset`: it must set the algorithm to its default state, forgetting the learning process already conducted;
- `learn`: this method must implement the learning process of Pegasos.

```
    @Override
    public BinaryLinearClassifier getPredictionFunction(){
            return this.classifier;
    }

    @Override
    public PegasosLearningAlgorithm duplicate() {
            PegasosLearningAlgorithm copy = new
                PegasosLearningAlgorithm();
            copy.setK(k);
            copy.setLambda(lambda);
            copy.setIterations(iterations);
            copy.setRepresentation(representation);
            return copy;
    }

    @Override
    public void reset() {
            this.classifier.reset();
    }

    @Override
    public void learn(Dataset dataset) {
            if(this.getPredictionFunction().getModel()
            .getHyperplane()==null){
                    this.getPredictionFunction().getModel()
                    .setHyperplane(
                            dataset.getZeroVector(representation)
                    );
            }

            for(int t=1;t<=iterations;t++){

                    List<Example> A_t = dataset.getRandExamples(k);
                    List<Example> A_tp = new ArrayList<Example>();
                    List<Float> signA_tp = new ArrayList<Float>();
                    float eta_t = ((float)1)/(lambda*t);
                    Vector w_t =
                        this.getPredictionFunction().getModel()
                    .getHyperplane();

                    //creating A_tp
                    for(Example example: A_t){
                            BinaryMarginClassifierOutput
                                prediction =
                                this.classifier.predict(example);
                            float y = -1;
                            if(example.isExampleOf(label)){
```

```
                                        y=1;
                                }

                                if(prediction.getScore(label)*y<1){
                                        A_tp.add(example);
                                        signA_tp.add(y);
                                }
                        }
                        //creating w_(t+1/2)
                        w_t.scale(1-eta_t*lambda);
                        float miscassificationFactor = eta_t/k;
                        for(int i=0; i<A_tp.size(); i++){
                                Example example = A_tp.get(i);
                                float y = signA_tp.get(i);
                                this.getPredictionFunction().getModel()
                                .addExample(y*miscassificationFactor,
                                    example);
                        }

                        //creating w_(t+1)
                        float factor = (float) (1.0/Math.sqrt(lambda)/
                                Math.sqrt(w_t.getSquaredNorm()));
                        if(factor < 1){
                                w_t.scale(factor);
                        }

                }

        }
```

6 Sample Code

In this Section, we are going to provide examples of usage of some of the KeLP functionalities. The full code of these examples is available in the KeLP repositories, in particular in the kelp-full package in GitHub. Other examples are available in the KeLP official website.

6.1 Classification example

This example contains the starter code to learn a classifier and to use it. In the most general case, three main steps are necessary to induce a classifier and to use it.

First of all, a training and a test datasets should be loaded in memory.

```
SimpleDataset tr = new SimpleDataset();
tr.populate("<path_to_the_train_dataset_in_kelp_format>");

SimpleDataset testSet = new SimpleDataset();
testSet.populate("<path_to_the_test_dataset_in_kelp_format>");
```

Then, the learning process of a classifier must be set up, and the learn method must be invoked. Notice that the algorithm works on a specific representation, whose name is `REPR1`.

```
StringLabel positiveLabel = new StringLabel("+1");
LinearPassiveAggressiveClassification pa=new
    LinearPassiveAggressiveClassification();
// use the first representation
pa.setRepresentation("REPR1");
// indicate to the learner what is the positive class
pa.setLabel(positiveLabel);
// set an aggressiveness parameter for the PA
pa.setC(0.01f);
```

```
// Learn the model
pa.learn(tr);
```

Finally, the classification function is used to classify a new instance, whose prediction is added to the counts of a `BinaryClassificationEvaluator`.

```
BinaryClassificationEvaluator ev = new
    BinaryClassificationEvaluator(positiveLabel);
Classifier f = pa.getPredictionFunction();
List<Prediction> testPredictions = new ArrayList<Prediction>();
for (int i=0; i<testSet.getNumberOfExamples(); ++i) {
        ClassificationOutput p = f.predict(testSet.getNextExample());
        testPredictions.add(p);
        ev.addCount(test,p);
}
```

6.2 Usage of ExperimentUtils facilities

In ML it is often necessary to tune the classifier parameters or to make more reliable measures in an experiment via cross-validation. These activities are repetitive, and it is easy to extract patterns in code that can be re-used. In KeLP we provide a class for automatizing some of these activities, the class `ExperimentUtils`. First, this class contains a method `test(PredictionFunction, Evaluator, Dataset)` the will produce in output a `List<Prediction>` and will update the "counters" of the `Evaluator`. It serves to automatize the repetitive operations that are executed for classifying a test set. For example, let us consider the last code snippet of Section 6.1. It should be re-written by exploiting the `test` method in `ExperimentUtils` as:

```
BinaryClassificationEvaluator ev = new
    BinaryClassificationEvaluator(positiveLabel);
Classifier f = pa.getPredictionFunction();
List<Prediction> testPredictions = ExperimentUtils.test(f, ev,
    testSet);
```

Moreover, it contains methods that helps in making predictions with a n-fold cross validation strategy.

Let us consider the code example in Section 6.1. If we would like to make 5-fold cross measures it is possible to adopt the `ExperimentUtils` methods, such as:

```
nfold=5;
List<BinaryClassificationEvaluator> nfoldEv =
    ExperimentUtils.nFoldCrossValidation(nfold, pa, tr, ev);
```

And then retrieve for example the accuracy measures for each fold and compute mean and standard deviation:

```
import it.uniroma2.sag.kelp.utils.Math;

float[] values = new float[nfoldEv.size()];
for (int i = 0; i < ret.length; i++) {
        values[i] = nfoldEv.get(i).getPerfomanceMeasure("Accuracy");
}
float mean = Math.getMean(values);
double standardDeviation = Math.getStandardDeviation(values);
```

6.3 Instantiation from JSON

JSON serialization/deserialization mechanism in KeLP is intended to support the definition of learning architectures with textual configuration files. KeLP offers the possibility to specify in a JSON human-readable format all the kernels and algorithms that are implemented. This formalism is intended to enable deploy of new algorithms or kernels in production environments without the need of reloading the entire Java Virtual Machine, but only changing a specification file.

For example, let us consider the example of Section 6.1. The `LinearPassiveAggressiveClassification` algorithm can be described in JSON with the following code:

```
{
  "algorithm" : "linearPA",
  "label" : {
    "labelType" : "StringLabel",
    "className" : "1"
  },
  "policy" : "PA_II",
  "loss" : "RAMP",
  "cp" : 2.0,
  "fairness" : false,
  "representation" : "REPR1",
  "cn" : 2.0
}
```

and it can be loaded in memory with (assuming the JSON is saved in a file called jsonSpecification.klp):

```
JacksonSerializerWrapper serializer = new JacksonSerializerWrapper();
LinearPassiveAggressiveClassification pa =
        serializer.readValue(
```

```
        new File("jsonSpecification.klp"),
        LinearPassiveAggressiveClassification.class);
```

The same applies with a kernel function. For example, let us consider the following JSON code:

```
"kernel":{
        "kernelType":"poly",
        "degree":2,
        "kernelCache":{
                "cacheType":"fixIndex",
                "examplesToStore":6000
        },
        "baseKernel":{
                "kernelType":"linear",
                "representation":"REPR1"
        }
}
```

It is the JSON description of a composition kernel, i.e., 2-degree polynomial kernel, applied on top of a linear kernel that is applied on a representation whose name is `REPR1`.

Again, this JSON code can be loaded similarly to the previous example (assuming the JSON code is saved on a file called ”kernelSpecification.klp”):

```
JacksonSerializerWrapper serializer = new JacksonSerializerWrapper();
Kernel kernel =
        serializer.readValue(
        new File("kernelSpecification.klp"),
        PolynomialKernel.class);
```

Bibliography

[Annesi *et al.*(2014)] Annesi, P., Croce, D., and Basili, R. (2014). Semantic compositionality in tree kernels. In *Proc. of CIKM 2014*, pages 1029–1038, New York, NY, USA. ACM.

[Borgwardt and Kriegel(2005)] Borgwardt, K. M. and Kriegel, H.-P. (2005). Shortest-path kernels on graphs. In *Proceedings of the Fifth IEEE International Conference on Data Mining*, ICDM '05, pages 74–81, Washington, DC, USA. IEEE Computer Society.

[Bunescu and Mooney(2005)] Bunescu, R. C. and Mooney, R. J. (2005). Subsequence kernels for relation extraction. In *Proceedings of NIPS*, pages 171–178.

[Carreras and Màrquez(2005)] Carreras, X. and Màrquez, L. (2005). Introduction to the CoNLL-2005 Shared Task: Semantic Role Labeling. In *Proc. of CoNLL-2005*, pages 152–164, Ann Arbor, Michigan.

[Cavallanti *et al.*(2007)] Cavallanti, G., Cesa-Bianchi, N., and Gentile, C. (2007). Tracking the best hyperplane with a simple budget perceptron. *Mach. Learn.*, **69**(2-3), 143–167.

[Cesa-Bianchi and Gentile(2006)] Cesa-Bianchi, N. and Gentile, C. (2006). Tracking the best hyperplane with a simple budget perceptron. In *In Proc. of the 19th Annual Conference on Computational Learning Theory*, pages 483–498. Springer-Verlag.

[Chang and Lin(2011)] Chang, C.-C. and Lin, C.-J. (2011). LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, **2**, 27:1–27:27.

[Collins and Duffy(2001)] Collins, M. and Duffy, N. (2001). Convolution kernels for natural language. In *Proceedings of Neural Information Processing Systems (NIPS'2001)*, pages 625–632.

[Cortes and Vapnik(1995)] Cortes, C. and Vapnik, V. (1995). Support-vector networks. *Mach. Learn.*, **20**(3), 273–297.

[Crammer *et al.*(2006)] Crammer, K., Dekel, O., Keshet, J., Shalev-Shwartz, S., and Singer, Y. (2006). Online passive-aggressive algorithms. *J. Mach. Learn. Res.*, **7**, 551–585.

[Croce and Basili(2016)] Croce, D. and Basili, R. (2016). Large-scale kernel-based language learning through the ensemble nystrom methods. In *ECIR*.

[Croce and Previtali(2010)] Croce, D. and Previtali, D. (2010). Manifold Learning for the Semi-Supervised Induction of FrameNet Predicates: An Empirical Investigation. In *Proceedings of the 2010 Workshop on GEometrical Models of Natural Language Semantics*, GEMS '10, pages 7–16, Uppsala, Sweden.

[Croce *et al.*(2011)] Croce, D., Moschitti, A., and Basili, R. (2011). Structured lexical similarity via convolution kernels on dependency trees. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '11, pages 1034–1046, Stroudsburg, PA, USA. Association for Computational Linguistics.

[Croce *et al.*(2012)] Croce, D., Basili, R., Moschitti, A., and Palmer, M. (2012). Verb classification using distributional similarity in syntactic and semantic structures. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics: Long Papers - Volume 1*, ACL '12, pages 263–272, Stroudsburg, PA, USA. Association for Computational Linguistics.

[Fan *et al.*(2008)] Fan, R.-E., Chang, K.-W., Hsieh, C.-J., Wang, X.-R., and Lin, C.-J. (2008). Liblinear: A library for large linear classification. *J. Mach. Learn. Res.*, **9**, 1871–1874.

[Filice *et al.*(2014)] Filice, S., Castellucci, G., Croce, D., and Basili, R. (2014). Effective kernelized online learning in language processing tasks. In *ECIR*, pages 347–358.

[Filice *et al.*(2015)] Filice, S., Da San Martino, G., and Moschitti, A. (2015). Structural representations for learning relations between pairs of texts. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1003–1013, Beijing, China. Association for Computational Linguistics.

[Hall *et al.*(2009)] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). The weka data mining software: An update. *sigkdd explor.*, **11**(1).

[Hsieh *et al.*(2008)] Hsieh, C.-J., Chang, K.-W., Lin, C.-J., Keerthi, S. S., and Sundararajan, S. (2008). A dual coordinate descent method for large-scale linear svm. In *Proceedings of the ICML 2008*, pages 408–415, New York, NY, USA. ACM.

[Joachims(1999)] Joachims, T. (1999). Making large-scale SVM learning practical. In B. Schölkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods - Support Vector Learning*, pages 169–184. MIT Press.

[Kulis *et al.*(2005)] Kulis, B., Basu, S., Dhillon, I., and Mooney, R. (2005). Semi-supervised graph clustering: A kernel approach. In *Proceedings of the ICML 2005*, pages 457–464, New York, NY, USA. ACM.

[Morik *et al.*(1999)] Morik, K., Brockhausen, P., and Joachims, T. (1999). Combining statistical learning with a knowledge-based approach - a case study in intensive care monitoring. In *ICML*, pages 268–277, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Moschitti(2006)] Moschitti, A. (2006). Efficient convolution kernels for dependency and constituent syntactic trees. In *ECML*, pages 318–329, Berlin, Germany. Machine Learning: ECML 2006, 17th European Conference on Machine Learning, Proceedings.

[Orabona *et al.*(2008)] Orabona, F., Keshet, J., and Caputo, B. (2008). The projectron: a bounded kernel-based perceptron. In *Int. Conf. on Machine Learning*. IDIAP-RR 08-30.

[Rosenblat(1958)] Rosenblat, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psych. Review*, **65**, 386–408.

[Schölkopf *et al.*(2000)] Schölkopf, B., Smola, A. J., Williamson, R. C., and Bartlett, P. L. (2000). New support vector algorithms. *Neural Comput.*, **12**(5), 1207–1245.

[Schölkopf *et al.*(2001)] Schölkopf, B., Platt, J. C., Shawe-Taylor, J. C., Smola, A. J., and Williamson, R. C. (2001). Estimating the support of a high-dimensional distribution. *Neural Comput.*, **13**(7), 1443–1471.

[Severyn *et al.*(2013)] Severyn, A., Nicosia, M., and Moschitti, A. (2013). Building structures from classifiers for passage reranking. In *Proceedings of the 22Nd ACM International Conference on Conference on Information and Knowledge Management*, CIKM '13, pages 969–978, New York, NY, USA. ACM.

[Shalev-Shwartz *et al.*(2007)] Shalev-Shwartz, S., Singer, Y., and Srebro, N. (2007). Pegasos: Primal estimated sub–gradient solver for SVM. In *Proc. of ICML*.

[Shawe-Taylor and Cristianini(2004)] Shawe-Taylor, J. and Cristianini, N. (2004). *Kernel Methods for Pattern Analysis*. Cambridge University Press.

[Shen and Joshi(2003)] Shen, L. and Joshi, A. K. (2003). An svm based voting algorithm with application to parse reranking. In *Proceedings of the Seventh Conference on Natural Language Learning at HLT-NAACL 2003 - Volume 4*, CONLL '03, pages 9–16, Stroudsburg, PA, USA. Association for Computational Linguistics.

[Vapnik(1998)] Vapnik, V. N. (1998). *Statistical Learning Theory*. Wiley-Interscience.

[Vishwanathan and Smola(2002)] Vishwanathan, S. and Smola, A. J. (2002). Fast kernels on strings and trees. In *Proceedings of Neural Information Processing Systems*, pages 569–576.

[Wang *et al.*(2012)] Wang, J., Zhao, P., and Hoi, S. C. (2012). Exact soft confidence-weighted learning. In J. Langford and J. Pineau, editors, *Proceedings of the 29th International Conference on Machine Learning (ICML-12)*, pages 121–128, New York, NY, USA. ACM.

[Wang and Vucetic(2010)] Wang, Z. and Vucetic, S. (2010). Online passive-aggressive algorithms on a budget. *Journal of Machine Learning Research - Proceedings Track*, **9**, 908–915.

[Williams and Seeger(2001)] Williams, C. K. I. and Seeger, M. (2001). Using the nyström method to speed up kernel machines. In T. Leen, T. Dietterich, and V. Tresp, editors, *Advances in Neural Information Processing Systems 13*, pages 682–688. MIT Press.