How to use the 10-PPI-kernels-package

Domonkos Tikk, Illés Solt, Philippe Thomas

July 28, 2011

This document briefly summarizes how to use kernel methods provided in the on-line appendix of our paper [15]. The tools have been further elaborated to support subsequent papers (in preparation).

Platform All the below experiments have been performed under Linux. Most programs can be also compiled and run on other operating systems, but here we restrict our description to the Linux platform.
Required software:

- Subversion (see Appendix A.1 for setup instructions)
- PostgreSQL (database server, see Appendix A.3 for setup instructions, tested with versions 8.2 and 8.4)
- GNU make (tested with version 3.80)
- gcc (tested with version 4.4)
- GNU flex (tested with version 2.5)
- Java runtime and compiler (tested with version 1.6)
- ant (tested with version 1.7)
- Python (tested with version 2.6)
- Python NumPy (tested with version 1.3)

On a Ubuntu machine you could install/update all of the above by issuing:

```
sudo apt-get install subversion postgresql make sun-java6-bin sun-java6-jdk ant gcc flex python
    python-numpy
```

You could test the availability and version of all the above programs by observing the outputs of the following commands:

```
svn --version
psql --version
java -version
javac -version
ant -version
gcc --version
flex --version
python --version
python -c "import numpy; print numpy.version.version"
```

Required hardware:

- 4 GB RAM
- 80–100 GB hard-disk space, depending on how many parameter combinations are evaluated. (Alone the experiments with Kim's kernel require 40 GB space.)

Availability

- SVN: program codes, running scripts
- ZIP: corpora, learning formats, sample results of experiments

The ZIP files contain only such codes that can be generated by using the programs in the SVN repository. Therefore this can be used partly to jump over some steps of the entire process, such as e.g. the generation of the learning formats, and partly serves as samples the actual experiment may be compared with. The ZIP is provided in two parts. This has only technical reason to avoid files with size over 25MB.

When referring to a part of published code, we specify it sources as either SVN or ZIP.

Processing steps The processing pipeline is the following

1. Transform PPI-corpora into the appropriate learning format of the kernel-based classifier.
2. Learn a classification model on a given training set using the kernel, and then test the model on the corresponding test set.
3. Write the details of learning/testing into an evaluation file in the form of SQL `INSERT` statements.
4. Upload data into a database by executing the SQL `INSERT` statements of the evaluation file.
5. Store experimental results in the database, so that experiments can be evaluated.

This document discusses steps 1–4.

1 Running the benchmark

The SVN distribution includes several `Makefile`-s that execute the various programs and scripts.

The most important files are:

- `Makefile.config` – contains the definition of directories and variables used overall in the experiments (detailed in Section 1.1).
- `Corpora/Makefile` – downloads the corpora, makes necessary modifications and creates the learning format for syntactic tree based kernels and kBSPS kernel.
- `Corpora/Makefile.Fayruzov` – creates the learning format for the Fayruzov implementation [4] of Kim's kernels.
- `Corpora/APG/Makefile` – creates the learning format for the APG kernel [1].
- `Experiments/Makefile` – runs the experiments for all supported kernels (except the ones having separate Makefile, listed below); inserts the results of experiments into the database.

1.1 Configuration

`Makefile.config` contains all directory definition and variables that is needed to execute the entire package. Here we depict only those ones which may need to be set after installation.

```
baseDir=${HOME}/ppi-benchmark
```

`baseDir` is the root directory of the package. All other directories are set relative to `baseDir`. One may need to modify this, if the package `ppi-benchmark` is not downloaded directly into the `${HOME}` directory. Assuming you followed instructions in Appendix A.1, you do not have to modify this setting.

```
PSQLCONNECT=psql -h racer
```

Specifies the connection to the PostgreSQL database server. Change the name `racer` to the host computer where the Postgres DBMS resides, and ensure that the host can be connected without explicit authentication. (Please follow the instructions in Appendix A.3 or refer to the documentation of the `psql` command line client[1].) Assuming you followed instructions in Appendix A.3 and have a server running locally, set

```
PSQLCONNECT=psql -h localhost -U ppi
```

```
BENCHMARKCORPORA=AIMed BioInfer HPRD50 IEPA LLL
CORPORA=${BENCHMARKCORPORA}
KERNELS=ST SST PT SpT kBSPS APG cosine edit SL Kim
EXPTYPES=CV CC CL
```

In `CORPORA` the identifiers of each corpus in the experiment listed. For convenience, this is copied from `BENCHMARKCORPORA` to facilitate easier overriding. `KERNELS` and `EXPTYPES` contain the identifiers of all kernels and experimentation types, respectively. One may add here new corpora, kernels and experimentation types. In that case, the availability of the corpora, the learning-format and the kernel implementation has to be assured.

```
runSynTree=${kernelDir}/SVM-Light-1.5-to-be-released/trunk
runSpT=${kernelDir}/svm_light/branches/svm_light_spectrum_tree_kernel
runkBSPS=${kernelDir}/svm_light/trunk
```

Definition of the root directory of SVM executables. This is only used for C++ implementations. Java implementation uses `classpath` definition to locate the executable Java classes. These settings should be not be altered normally, unless you migrate your executables.

In addition, the configuration file contains decompression and download related targets.

2 Corpora

2.1 Original version

Five corpora are used in the experiments: AIMed, BioInfer, HPRD50, IEPA, and LLL. We use the derived version of these corpora created by Pyysalo's group [14], which unifies PPI-annotation and contains parses created with the Charniak–Lease parser [9] and transformed to collapsed Stanford format [3]. The files are in directory `/Corpora/Original` (ZIP). The corpora can be downloaded directly from `http://mars.cs.utu.fi/PPICorpora/GraphKernel.html`. This can be done also by

```
cd Corpora
make download
```

2.2 Syntax-Tree version

Syntax tree based kernels require syntax parses in the learning format. This conversion was done first by Peter Palaga [13], then improved by the authors. The conversion is described in Section 3 and in more detail in Appendix C–D. We provide the converted version in the directory `/Corpora/Syntax-Tree-Learning-Format` (ZIP).

2.3 Split

In order to insure the reproducibility and comparativeness of the experiments, we use the *de facto* standard document level splits of Airola [1].

The splits resides in `/Corpora/Splits` (SVN), under the respective corpus name. This version is used by syntax tree kernels, kBSPS, and Kim's kernels. APG kernel uses the same train-test splits but in a different format, which can be found in `/Corpora/splits-test-train` (SVN).

[1] `http://www.postgresql.org/docs/8.1/static/app-psql.html`

2.4 Creating the enriched XML format

We created a separate target to perform exclusively the enrichment of XML files with all necessary syntax parse and dependency parse information. To execute the target `generate-enriched-xml` the input XML file should be in `/Corpora/Original`, for instance `/Corpora/Original/Test.xml`. The XML file should be in Airola format and should contain the `sentence`, `entity`, `pair` elements under `corpus/document`. Additionally the value of `CORPORA` variable in `/Makefile.config` should be changed accordingly (e.g. to `Test`). After the preprocessing, the execution is done as

```
cd Corpora
make generate-enriched-xml
```

3 Learning formats

Various kernel methods work with different learning formats. The learning format depends on the

- The classification algorithm used; SVM or RLS; various SVM implementations (SVM^{light}, SVM-TK, LibSVM)
- Integration of kernel function into the classifier;
- The kernel itself.

We provide learning formats for all experiment types, namely cross-validation (CV), cross-learning (CL), and cross-corpus (CC) evaluations. In the subsequent subsections we describe the process of learning format creation for all learning softwares.

3.1 Creating the learning format SL kernel

The learning format of SL is created by calling the following target:

```
cd Corpora
make create-SL-LF
```

This creates all the necessary training data for CV, CC and CL evaluation, performed for all corpora in `CORPORA`.

3.2 Creating the learning format for the Moschitti and spectrum tree kernels

Learning format creation is performed in several steps.

Step 1 Creates the appropriate input format from the derived XML files for the syntactic parser.

Step 2 Performs syntactic parsing.

Step 3 Injects the syntactic parses into the derived XML files.

Step 4 Aligns the original sentence with the syntactic parses (specifies the character offsets in the raw text and the syntactic parse trees).

Step 5 Injects the results of the above alignment into the XML files.

Step 6 Creates the learning format for various evaluation scenarios (CV, CC, CL).

All steps can be executed via the provided `Makefile` in the `Corpora` directory (SVN). Below we describe its targets. The detailed description of each step can be found in Appendix C.

```
cd Corpora
make all
```

Runs the entire processing chain and creates learning format for all evaluation methods (CV, CC and CL). This also includes the creation of kBSPS learning format for technical reason. The original syntactic tree learning format does not include the identifier of each entity pair. This is copied from the kBSPS version.

```
make pre-steps
```

Runs the following three steps together: `download`, `repair-BioInfer` and `compile`.

```
make download
```

Download the corpora and normalize their names.

```
make repair-BioInfer
```

The original BioInfer corpus contains a minor tokenization error in sentence `BioInfer.d109.s0`, which is repaired by this code. Without this repair, Step 4 would crash with an error on BioInfer.

```
make compile
```

Compiles the Java code for the learning format transformation. Prior running it checks for java updates in the SVN.

```
make main-steps
```

Runs all the 6 steps described above for syntax tree kernels and the 6th step for kBSPS kernel (see Section 3.3).

```
make main-steps-syntree
```

Runs all the 6 steps described above for syntax tree kernels. Among them parsing (Step 2) takes the longest time, but on a 4-core machine the entire pipeline is ready within 10 minutes.

```
make post-steps
```

Perform the below detailed four targets: `clean`, `createCC`, `createCL` and `idVersion`.

```
make clean
```

Delete temporary files.

```
make createCC
```

Creates cross-corpus (CC) learning formats based on the CV learning format.

```
make createCL
```

Creates cross-learning (CL) learning formats based on the CC learning format.

```
make idVersion
```

Add the id of each learning instances to the learning format based on the kBSPS files.

3.3 Creating the learning format for kBSPS

The creation of learning format is simpler because the kernel uses the dependency graphs already available in the original learning script.

```
make all-kBSPS
```

Runs the entire processing chain for kBSPS kernel (that is indeed only the Step 6 from above, since syntax trees do not need to be injected into the learning format) and creates learning format for all evaluation methods (CV, CC and CL). This target can be executed on its own without preparing the syntax tree learning formats.

```
make pre-steps
```

As above in Section 3.2.

```
make main-steps-kBSPS
```

Creates the learning format for kBSPS.

```
make post-steps-kBSPS
```

This consists of only two targets, `createCC` and `createCL` which create the corresponding CC and CL evaluation files from CV files (in this order).

3.4 Creating the learning format for the APG kernel

The learning format of the APG kernel can be created by the `Corpora/APG/Makefile` as

```
cd Corpora/APG
make all
```

This requires that the enriched XML files (generated previously at syntax tree kernels) are available.

This calls the following three targets that belong to the appropriate experiment types: `generate-CV-data`, `generate-CL-data`, `generate-CC-data`.

3.5 Creating the learning format for Kim's kernels

The creation of learning format is performed in two main steps. The target for those steps are defined in `Corpora/Makefile.Fayruzov`. This file is imported into `Corpora/Makefile`. The steps are the followings:

```
cd Corpora
make parse-Fayruzov
```

Launch the Stanford parser to parse the available corpora. The target checks the availability of the necessary JAR file and grammar definition file. The target does not download automatically the required files, because they are embedded in larger packages which are not needed for the current task. Instead when the required files are missing, it returns an error message specifying the missing file and its location on the net.

```
make create-libSVM-LF
```

Calls the three targets to create the learning formats for CV, CC and CL evaluations.

4 Kernels and classifiers

For running experiments we provide `Experiments/Makefile`. The main target in the makefile is `experiment` that should be called as described by `make help`:

```
cd Experiment
make help
```

```
use 'make experiment Corpora="AIMed BioInfer HPRD50 IEPA LLL" Kernel="ST | SST | PT | SpT | kBSPS | APG |
    cosine | edit | SL | Kim" expType="CV | CC | CL"'
```

The target can be called with any subset of the available `Corpora`, with only a single kernel given in `Kernel` and a single experimentation type given in `expType`. The target accepts only those corpora, kernels and experimentation types that are given in `CORPORA`, `KERNELS`, and `EXPTYPES`, resp., defined in `Makefile.config`.

We provide an additional `experiment-test` target which runs only one experiment with the fastest setting, in order to check the functionality of the kernel.

4.1 Shallow linguistic (SL) kernel

The source code of the classifier is located under `Kernels/jsre/source` (SVN). This was obtained from Claudio Giuliano [5]. As its name suggests, the kernel uses only shallow linguistic features, but no syntax or dependency parses.

This kernel does not require any compilation step before running. If a forced rebuild of the Java executable is necessary, this can be done as:

```
cd Kernels/jsre/source
ant build
```

Experiments can be run using the `experiment` and `experiment-test` targets as specified above (page 6).

4.2 Syntax tree based kernels: ST, SST, and PT

The source code of the classifier, called SVM-Light V5.01-TK-1.5, can be found in `Kernels/SVM-Light-1.5-to-be-released/trunk` (SVN). It is an extension of SVM^{light} by Alessandro Moschitti [11, 12]. SVM^{light} is the SVM implementation of Torsten Joachims [6]. As of 31.05.2010, only the earlier version 1.2 of the SVM-Light-TK software is available online: http://dit.unitn.it/~moschitt/Tree-Kernel.htm. We received the pre-release version as the courtesy of the author. It contains the file `kernel.h` created by Moschitti, which implements 3 kernels: subtree (ST), subset tree (SST), and partial tree (PT) kernels. The various kernels can be selected via the `-F` option.

The kernel-based classifier can be compiled with

```
cd Kernels/SVM-Light-1.5-to-be-released/trunk
make all
```

that creates two executables, `svm_learn` and `svm_classify`.

The `experiment` and `experiment-test` targets call the `Experiments/ST-SST-PT/execute-syntree.sh` script with the appropriate setting. The script is detailed in Appendix B.

4.3 Spectrum tree (SpT) kernel

Spectrum tree kernel was implemented by Peter Palaga [13] based on the work of Kuboyama [8]. The kernel is integrated into the SVM^{light} package. It uses slightly different learning-format than the Moschitti version, therefore we treat it separately. The source code of the classifier can be found in `Kernels/svm_light/branches/svm_light_spectrum_tree_kernel` (SVN). The kernel function is embedded in the same way as in the Moschitti implementation, using the user-defined `kernel.h` file.

The kernel-based classifier can be compiled with

```
cd Kernels/svm_light/branches/svm_light_spectrum_tree_kernel
make all
```

that creates two executables, `svm_learn` and `svm_classify`.

The `experiment` and `experiment-test` targets call the `Experiments/SpT/execute-SpT.sh` script with the appropriate setting. The script is analogous to the `execute-syntree.sh` script detailed in Appendix B.

4.4 k-band shortest path spectrum (kBSPS) kernels

kBSPS kernel was invented by Peter Palaga [13]. The kernel is integrated into the SVM^{light} package. It uses a different learning-format than the Moschitti version: only the k-band shortest paths are passed to the classifier, where tokens (nodes) and edges (dependency types) are represented by integers, which allows for a much faster learning. Consequently, the retrieval of k-band shortest paths from the dependency graphs should be executed when creating the learning format. It requires that for all possible combinations of parameters effective to the k-band shortest paths to be retrieved, a separate learning corpus have to be created. Those parameters are k, $q_{\min}$ and $q_{\max}$. The provided zip file contains the learning corpus for kBSPS for the following parameter settings:

- $k = \{0, 1\}$, $q_{\min} = \{1, 2\}$ and $q_{\max} = \{2, 3\}$

When experiments with different parameter settings are required to be performed new learning corpora have to be generated. The description of the script that creates the learning format can be found in Section 3.3, details in Appendix D.

The source code of the classifier can be found in `Kernels/svm_light/trunk` (SVN). The kernel function is embedded in the same way as in the Moschitti implementation, using the user-defined `kernel.h` file.

The kernel-based classifier can be compiled with

```
cd Kernels/svm_light/trunk
make all
```

that creates two executables, `svm_learn` and `svm_classify`.

The `experiment` and `experiment-test` targets call the `Experiments/kBSPS/execute-kBSPS.sh` script with the appropriate setting. The script is analogous to the `execute-syntree.sh` script detailed in Appendix B.

4.5 Cosine similarity (cosine) based kernel

4.6 Edit distance (edit) based kernel

4.7 All-path graph (APG) kernel

This kernel does not require any compilation step before running. Experiments can be run using the `experiment` and `experiment-test` targets as specified above (page 6). Before running the script, make sure that python[2] and NumPy[3] are installed on your computer.

4.8 Kim's kernels

We obtained the implementation of Kim's kernels from Timur Fayruzov [4] (December 2009), because the original kernels published in [7] were not available at Autumn 2009. The package contains four kernels called lex, syn, pos, and comp, which are, respectively, based on lexical dependency trees, deep syntactic features, POS dependency trees, and the combination of all. The kernels are integrated into the libSVM package [2]. Because we received the Fayruzov-implementation after the submission of our paper [15], the results related to these kernels are not included therein, but are considered on our subsequent PPI-related studies.

The source code of the classifier can be found in `Kernels/Fayruzov` (SVN). The kernels are embedded into the libSVM package as the precomputed Gram-matrix of all instances (pairwise distance of the all instances). Consequently, when working on large corpora, the size of those matrices can be pretty large, for CL evaluation it attain even several GBs, therefore we refrain from providing these files even in the zip archive.

The kernel-based classifier can be compiled with

```
cd Kernels/Fayruzov
make all
```

that compiles the Java project. The two executables, `svm-train` and `svm-predict`, can be found in `Kernels/Fayruzov/resource/svm`; these are the binaries of the libSVM version 2.9 (November 2009)[4].

The `experiment` and `experiment-test` targets call the `Experiments/KimKernel/execute-KimKernel.sh` script with the appropriate setting. The script is analogous to the `execute-syntree.sh` script detailed in Appendix B.

[2] http://www.python.org/
[3] http://www.numpy.org
[4] http://www.csie.ntu.edu.tw/~cjlin/cgi-bin/libsvm.cgi?+http://www.csie.ntu.edu.tw/~cjlin/libsvm+zip

5 The PPI experiments database

We use PostgreSQL DBMS to store the experiments. We provide scripts to initialize the database tables, create views and functions and finally to insert the results of the experiments described above into the database. The database schema is further discussed in Appendix E

5.1 Creating the tables

The tables are created via the following three SQL scripts located in folder `Database` (SVN):

1. `init-ppiCV.sql` – creates the tables for CV experiments: `ppiCV`, `ppiCVoutput` and `ppiCVfolds`;
2. `init-ppiCC.sql` – creates the tables for CC experiments: `ppiCC` and `ppiCCoutput`;
3. `init-ppiCL.sql` – creates the tables for CL experiments: `ppiCL` and `ppiCLoutput`.

The scripts can be invoked e.g. via the PostgreSQL command line interface, assuming the name `ppi` for the database

```
cd Database
psql ppi
> \i init-ppiCV.sql
```

or from the shell (if the DBMS is located on the same machine):

```
cd Database
psql ppi -f init-ppiCV.sql
```

5.2 Uploading database scripts into the database

Uploading is done by `Experiment/Makefile`. The main target in the makefile is `output2db` that should be invoked as:

```
make output2db Corpora="AIMed BioInfer HPRD50 IEPA LLL" Kernel="ST | SST | PT | SpT | kBSPS | APG |
    cosine | edit | SL | Kim" expType="CV CC CL"
```

To be able to run this target the connection to the Postgres database should be assured (see Appendix A.3). The target can be called with any subset of the available `Corpora` and `expType`, but with only a single kernel given in `Kernel`. The target accepts only those corpora, kernels and experimentation types that are given in `CORPORA`, `KERNELS`, and `EXPTYPES`, resp., defined in `Makefile.config`. In case of kernels SL, cosine and edit, the value of `Corpora` is need not to be specified, since all results are stored in one larger file (consequently the value of `Corpora` is ignored).

The final step assigns a separate identifier to the 10 fold-wise separately stored experiments of each cross-validation run, which enables to handle the 10 runs together. This is performed by the script `manage_folds.sql` as

```
cd Database
psql ppi -f manage_folds.sql
```

We remark that in addition to the averaged results of the experiments in terms of precision, recall, F-measure and AUC, we also keep track of the prediction given for each test case. This allow of an instance level error analysis, that is the focus of of current study.

References

[1] A. Airola, S. Pyysalo, J. Björne, T. Pahikkala, F. Ginter, and T. Salakoski. All-paths graph kernel for protein-protein interaction extraction with evaluation of cross-corpus learning. *BMC bioinformatics*, 9(Suppl 11):S2, 2008.

[2] C.-C. Chang and C.-J. Lin. LIBSVM: a library for support vector machines, 2001. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[3] M. C. de Marneffe, B. MacCartney, and C. D. Manning. Generating typed dependency parses from phrase structure parses. In *Proceedings of LREC'06*, 2006.

[4] T. Fayruzov, M. De Cock, C. Cornelis, and V. Hoste. Linguistic feature analysis for protein interaction extraction. *BMC Bioinformatics*, 10(1):374, 2009.

[5] C. Giuliano, A. Lavelli, and L. Romano. Exploiting shallow linguistic information for relation extraction from biomedical literature. In *Proc. of the 11st Conf. of the European Chapter of the Association for Computational Linguistics (EACL'06)*, pages 401–408, Trento, Italy, 2006. The Association for Computer Linguistics.

[6] T. Joachims. *Making large-scale support vector machine learning practical, Advances in kernel methods: support vector learning*. MIT Press, Cambridge, MA, 1999.

[7] S. Kim, J. Yoon, and J. Yang. Kernel approaches for genic interaction extraction. *Bioinformatics*, 24(1):118–126, Jan 2008.

[8] T. Kuboyama, K. Hirata, H. Kashima, K. Aoki-Kinoshita, and H. Yasuda. A spectrum tree kernel. *Information and Media Technologies*, 2(1):292–299, 2007.

[9] M. Lease and E. Charniak. Parsing biomedical literature. In *Natural Language Processing – IJCNLP 2005*, number 3651 in LNCS, pages 58–69. Springer, Berlin/Heidelberg, 2005.

[10] D. McClosky. *Any Domain Parsing: Automatic Domain Adaptation for Natural Language Parsing*. PhD thesis, Department of Computer Science, Brown University, 2009.

[11] A. Moschitti. Efficient convolution kernels for dependency and constituent syntactic trees. In *Proc. of The 17th European Conf. on Machine Learning*, pages 318–329, Berlin, Germany, 2006.

[12] A. Moschitti. Kernel methods, syntax and semantics for relational text categorization. In *Proceeding of ACM 17th Conf. on Information and Knowledge Management (CIKM'08)*, Napa Valley, CA, USA, 2008.

[13] P. Palaga. Extracting relations from biomedical texts using syntactic information. Master's thesis, Humboldt-Universität zu Berlin, May 2009.

[14] S. Pyysalo, A. Airola, J. Heimonen, J. Björne, F. Ginter, et al. Comparative analysis of five protein-protein interaction corpora. *BMC Bioinformatics*, 9 Suppl 3:S6, 2008.

[15] D. Tikk, P. Thomas, P. Palaga, J. Hakenberg, and U. Leser. A comprehensive benchmark of kernel methods to extract protein-protein interaction from literature. *PLoS Computational Biology*, 6(7):e1000837, 07 2010.

Appendix

A Getting dependencies to work

A.1 Setting up Subversion repository access

A.1.1 Installing Subversion client

Install Subversion client on Debian/Ubuntu:

```
sudo apt-get install subversion
```

Most other operating systems will have some way of installing Subversion as a pre-configured package. In a special case, you may still refer to the official Subversion download site[5].

A.1.2 Obtaining the 10-PPI-Kernels package

To check out the package enter:

```
svn co 'http://categorizer.tmit.bme.hu/svn/ppi-benchmark' ${HOME}/ppi-benchmark
```

This will download the package files into the local `ppi-benchmark/` folder into your home directory.

A.2 Decompressing the zip files

The zip files should be decompressed as

```
unzip ppi-benchmark1.zip ppi-benchmark1.zip -d TARGET_DIRECTORY
```

uncompress both zip files to the given target directory. The content of a zip files can be listed with

```
unzip -l ppi-benchmark1.zip
unzip -l ppi-benchmark2.zip
```

The first one contains 30 files, the second one 913 files.

A.3 Setting up a PostgreSQL database

A.3.1 Installing PostgreSQL client and server

Install PostgreSQL server and client on Debian/Ubuntu:

```
sudo apt-get install postgresql
```

If you already have a PostgreSQL server running on a remote machine:

```
sudo apt-get install postgresql-client
```

Most other operating systems will have some way of installing PostgreSQL as a pre-configured package. In a special case, you may still refer to the official PostgreSQL download site[6].

A.3.2 Creating a new user and database

On the machine running the PostgreSQL server create a new user called 'ppi' with password 'ppi':[7]

```
sudo useradd ppi -s /bin/false
sudo passwd ppi
```

Log in to the database as the database administrator:

[5] http://subversion.apache.org/packages.html
[6] http://www.postgresql.org/download/
[7] See http://www.cyberciti.biz/faq/howto-add-postgresql-user-account/ for a more detailed description.

```
sudo su -c "psql" postgres
```

You should get a database command prompt similar to this:

```
psql (8.4.4)
Type "help" for help.

postgres=#
```

On the database command prompt, enter the following commands to create a new database:

```
CREATE USER ppi WITH PASSWORD 'ppi';
CREATE DATABASE ppi;
GRANT ALL PRIVILEGES ON DATABASE ppi TO ppi;
CREATE LANGUAGE plpgsql;
```

Type '`\q`' to quit.

A.3.3 Allow network access

Allow network-based access (both local and remote) Edit the server machine's `pg_hba.conf`, e.g.:

```
sudo vim /etc/postgresql/8.*/main/pg_hba.conf
```

and append the following line:

```
host    ppi     ppi     0.0.0.0/0       md5
```

Optionally allow remote access If you will e accessing the PostgreSQL server from a remote machine, you will have to allow remote access to the server.

Edit the server's configuration file `postgresql.conf`, e.g.:

```
sudo vim /etc/postgresql/8.*/main/postgresql.conf
```

Locate the line

```
listen_addresses='localhost'
```

and add the server's outside address.

Restart PostgreSQL Restart the PostgreSQL server process to make the changes take effect:

```
sudo /etc/init.d/postgresql-8.* restart
```

Test your connection:

```
psql -h localhost --user ppi -d ppi --password
```

If using a remote server, you should change `localhost` to the remote server's address. Notice that you will have to enter the password for user 'ppi'.

You should get a database command prompt:

```
psql (8.4.4)
SSL connection (cipher: DHE-RSA-AES256-SHA, bits: 256)
Type "help" for help.

ppi=>
```

A.3.4 Set up password-less access

To avoid having to type the password for each connection, set up the `~/.pgpass` password file on the local machine:

```
touch ~/.pgpass
chmod 600 ~/.pgpass
cat >> ~/.pgpass <<EOF
#hostname:port:database:username:password
localhost:*:ppi:ppi:ppi
EOF
```

If using a remote server, you should change `localhost` to the remote server's address.

Test your connection:

```
psql -h localhost -U ppi -d ppi
```

B Detailed description of cross-validation scripts

B.1 The `execute-syntree.sh` script

The script is normally called via the `experiment` target of the `Experiment/Makefile`, but can also be invoked directly as

```
cd Experiments/ST-SST-PT
bash -c "corpusDir='$(corpusDir)' learner=$(learner) tester=$(tester) test='$(test)'
    allCorpora='$(Corpora)' kernels='$(Kernel)' expType='$(expType)' ./execute-syntree.sh"
```

where the parameters are the followings:

- `$(corpusDir)` is the full path of directory `Corpora` which contains the learning format in `learning-format`;
- `$(learner)` is the full path of the `svm_learn` file;
- `$(tester)` is the full path of the `svm_classify` file;
- `$(test)` is optional; its should be set to 1 to perform a test;
- `$(Corpora)` is the list of corpora on which the experiment should be run;
- `$(Kernel)` is any of ST, SST or PT, i.e. specifies which kernel function to be used;
- `$(expType)` is any of CV, CC, CL; i.e. specifies the evaluation type.

Now we describe the content of the script in detail.

```
. ../config.sh
```

Imports the kernel-independent configuration of the experiments. This includes the calculation of available processing units, and variable validation functions.

```
. config.sh
```

Imports the kernel-dependent configuration of the experiments. This includes the definition of test and normal settings, relating the kernel and SVM parameters.

```
# when CC then "location" is empty, otherwise, see below
if [ ${expType} != "CC" ];
then
  location=$corpus-folds
fi
inDir=${corpusDir}/learning-format/${expType}/MOSCHITTI/${location}
```

Specifies the location of the learning format. The `$(location)` is set according to the directory structure of the learning format files. `inDir` defines the actual directory where learning format files reside.

```
commands+="./${expType}_syntree_core.sh '${expType}' '${corpus}' '${j}' '${c}' '${lambda}' '${mu}'
    '${inDir}' '${learner}' '${kernel}' '${tester}'"
```

Append the actual to-be-executed experiments to the command list. Depending on the type of experiment `${expType}` one of the three scripts are called:

1. `CV_syntree_core.sh` – CV evaluation;
2. `CL_syntree_core.sh` – CL evaluation;
3. `CC_syntree_core.sh` – CC evaluation.

Various parametrization of the experiment is collected in `commands`, which are then passed to the computing cores using `xargs`:

```
echo ${commands} | nice nice xargs --no-run-if-empty -d '#' -P ${numberofcores} --max-args 1 -t bash -c
```

B.2 The `CV_syntree_core.sh` script

```
expType=$1; shift 1
corpus=$1; shift 1
j=$1; shift 1
c=$1; shift 1
lambda=$1; shift 1
mu=$1; shift 1
inDir=$1; shift 1
learner=$1; shift 1
kernel=$1; shift 1
tester=$1; shift 1
```

Parameter passing from the embedding script.

```
folds='0 1 2 3 4 5 6 7 8 9'
useFolds='0 1 2 3 4 5 6 7 8 9'
```

The definition of the ten folds. The variable `folds` is used for training and `useFolds` is used for test.

```
. kernelparams.sh
```

Imports the local `kernelparams.sh` file containing fix kernel-specific values to be inserted into the database input file.

```
outDir=./${kerneltext}/${expType}/${corpus}/out-${expType}-${corpus}-j${j}-c${c}-L${lambda}-M${mu}
tmpDir=./${kerneltext}/${expType}/${corpus}/tmp-${expType}-${corpus}-j${j}-c${c}-L${lambda}-M${mu}
```

Location of output and temporal directory is defined.

```
logFile=${outDir}/${expType}-${corpus}-j${j}-c${c}-L${lambda}-M${mu}.log
```

```
evalFile=${outDir}/${expType}-${corpus}-j${j}-c${c}-L${lambda}-M${mu}.sql
```

Location of `logFile` and the `evalFile` is defined. The results of the experiments are printed into the latter.

```
for fold in ${useFolds};
```

Each fold is used once as test.

```
learnFolds=${folds/$fold/}
for learnFold in ${learnFolds};
do
  cat ${inDir}/${learnFold}.txt.id >> ${learnCorpus}
done
cp ${inDir}/${fold}.txt.id ${testCorpus}
model=${outDir}/${fold}-model.txt
```

learnCorpus and testCorpus are constructed. It is important here that the files with .txt.id extension are used, which contains also the identifier of each instance (entity pair).

```
echo -n "insert into ppi${expType} (corpus, parsertype, parser, kernel, c, j, fold, kernel_script) values
    (" >> ${evalFile}
echo "'${corpus}', '${parsertype}', '${combo}', '${kerneltext}', ${c}, ${j}, ${fold}, 'lambda:${lambda}
    mu:${mu}');" >> ${evalFile}
```

Specifies the unique parameters of the current experiment.

```
${learner} -v -1 -t 5 -F ${kernel} -D ppi${expType} -C T -c ${c} -j ${j} -L ${lambda} -M ${mu}
    ${learnCorpus} ${model} >> ${evalFile}
```

```
${tester} -v -1 -D ppi${expType} ${testCorpus} ${model} >> ${evalFile}
```

Calls the training process and then evaluates the created model on testCorpus

```
sed -i 's/nan/0.0/g' ${evalFile}
```

Substitutes numerical errors caused by zero-devision.

C Details of learning format creation for syntax tree kernels

C.1 Step 1: preparing parsing input

The software is a part of Palaga's Learning format API Java library. The program extracts each sentence from the derived XML file and embeds them within <s>...</s> tags.

- Program: PtbRawSentenceTransformer.java
- Input: derived XML (specified with the -f parameter)
- Parameters:

  -f|--file input file name

  -o|--out output file name (optional, default: <input>-ptb-s.txt)

- Run from command line:

  ```
  java -classpath "${LFJARGSCLASSPATH}:${LFCLASSPATH}"
      org.learningformat.transform.PtbRawSentenceTransformer -f PPIcorpus.xml
  ```

  where ${LFJARGSCLASSPATH} is the learning-format-api/lib directory, and ${LFCLASSPATH} is the directory of java files.

- Output: PPIcorpus.xml-ptb-s.txt

C.2 Step 2: parsing

The second step is the parsing the corpora with Charniak-Lease-Johnson-McClosky parser. The input format is done in Step 1 (PPIcorpus.xml-ptb-s.txt file) the output is the parsed trees of the each sentence, e.g., the result of "*ykuD was transcribed by SigK RNA polymerase from T4 of sporulation.*" is

```
(S1 (S (NP (NNP ykuD)) (VP (AUX was) (VP (VBN transcribed)
(PP (IN by) (NP (NP (NNP SigK) (NNP RNA) (NN polymerase))
(PP (IN from) (NP (NNP T4))) (PP (IN of) (NP (NN sporulation))))))) (. .)))
```

- Parser: Charniak–Lease reranking parser executables: parseIt and bestparses, downloaded automatically by the Makefile, if missing. Located under RERANKINGPARSER=${baseDir}/Parsing/Charniak-Lease-2006Aug-reranking-parser.

- Model file: Self-training biomedical parsing model file from David McClosky (June 2009) [10] (several files), downloaded by the `Makefile`, if missing. Located under `BIOPARSINGMODEL=${baseDir}/Parsing/Models/McClosky-2009`
- Input: `PPIcorpus.xml-ptb-s.txt` files in format `<s>sentence</s>`.
- Parameter: to-be-parsed file (`ptb-s.txt`), output and error file should be redirected
- Run from command line:

```
${parseIt} -l399 -N50 ${parser} PPIcorpus.xml-ptb-s.txt | ${bestparses} -l ${featuresgz}
    ${weightsgz} > PPIcorpus.xml-ptb-s-parsed.txt 2>PPIcorpus.xml-ptb-s-parsed.txt.err
```

  where

  - `parseIt=${RERANKINGPARSER}/first-stage/PARSE/parseIt`
  - `parser=${BIOPARSINGMODEL}/parser/`
  - `bestparses=${RERANKINGPARSER}/second-stage/programs/features/best-parses`
  - `featuresgz=${BIOPARSINGMODEL}/reranker/features.gz`
  - `weightsgz=${BIOPARSINGMODEL}/reranker/weights.gz`
  - Output: `PPicorpus.xml-ptb-s.txt-parsed.txt` and `PPIcorpus.xml-ptb-s-parsed.txt.err`

C.3 Step 3: Injection of parsing results into XML files

This program injects into the original XML files the parsing results in `<bracketings>...</bracketings>` tags.

- Program: `PtbTreeInjector.java`
- Input: `PPIcorpus.xml` and the corresponding parse file: `PPIcorpus.xml-ptb-s.txt-parsed.txt`.
- Parameters:

  `-f|--file` input file name

  `-i|--inject` this is a switch which should be given for this step

  `-o|--out` output file name

  `-p|--parse` parse file name

  `-t|--token` token file name, this must not be given for Step 3.

- Running from command line:

```
java -classpath "${LFJARGSCLASSPATH}:${LFCLASSPATH}" org.learningformat.transform.PtbTreeInjector
    -f PPIcorpus.xml -p PPIcorpus.xml-ptb-s.txt-parsed.txt -o PPIcorpus.xml.inj1 -i \;
```

- Output: specified by the `-o` option, an XML file that contains now the parsing results in `<bracketings>...</bracketings>` tags, by convention: `PPIcorpus.xml.inj1`

C.4 Step 4: alignments of the original sentence with the parsing results

This program aligns the original sentence with the parsing results, that is it specifies the character offsets of the (from-to) of the tokens of the original text in the parsed texts. For the example sentence *“ykuD was transcribed by SigK RNA polymerase from T4 of sporulation.”* it is:

```
LLL.d2.s1,0-3:16-19,5-7:32-34,9-19:46-56,21-22:67-68,24-27:84-87,29-31:95-97,
33-42:104-113,44-47:125-128,49-50:140-141,52-53:154-155,55-65:166-176,66-66:188-188
```

- Program: `BracketingTokenMapper.java`

- Input: PPIcorpus.xml.inj1 that is the output file of Step 3.
- Parameter:
- Running from command line:

```
java -classpath "${LFJARGSCLASSPATH}:${LFCLASSPATH}"
    org.learningformat.transform.BracketingTokenMapper PPIcorpus.xml.inj1
```

- Output: a text file containing the alignments. The output text files gets the -bracketing-tokens.txt suffix, i.e. for PPIcorpus.xml.inj1 it is PPIcorpus.xml.inj1-bracketing-tokens.txt.

C.5 Step 5: Injection of sentence–parsing alignments into XML files

This program injects into the original XML files the alignments created in Step 4, the alignments are injected into the XML file within **<charOffsetMapEntry>** tags.

- Program: PtbTreeInjector.java
- Input: PPIcorpus.xml.inj1 and the corresponding PPIcorpus.xml.inj1-bracketing-tokens.txt
- Parameters:

  -f|--file input file name

  -i|--inject this switch must not given for Step 5

  -o|--out output file name

  -p|--parse parse file name, this must not be given for Step 5

  -t|--token token file name.

- Running from command line:

```
java -classpath "${LFJARGSCLASSPATH}:${LFCLASSPATH}" org.learningformat.transform.PtbTreeInjector
    -f PPIcorpus.xml.inj1 -t PPIcorpus.xml-bracketing-tokens.txt -o PPIcorpus.xml.inj1.inj2
```

- Output: specified by the -o option, an XML file that contains now the parsing results in **<charOffsetMapEntry>** tags. By convention this is PPIcorpus.xml.inj1.inj2.

C.6 Step 6: Creating folds for CV-evaluation

These programs create the training format for SVM based classifier, therefore are the last step in the preprocessing pipeline. The programs assume the availability of test-train splits (see Section 2).

The first program creates the training data for SVM learners using the syntax tree based kernels.

- Program: SvmLightTreeKernelTransformer.java
- Input: the PPIcorpus.xml.inj.inj2 files.
- Parameters:

  -f|--file input file name

  -o|--out output directory name

  -s|--split location of the split files (folder)

  -m|--moschitti flag for Moschitti style learning format (for subtree (ST), subset tree (SST), and partial tree (PT) kernels)

  -c|--custom flag for custom learning format (for spectrum tree (SpT) kernel)

  Exactly one of the -m and -c flags has to be given.

- Running from command line:

```
java -classpath "${LFJARGSCLASSPATH}:${LFCLASSPATH}"
    org.learningformat.transform.SvmLightTreeKernelTransformer -f -m PPIcorpus.xml.inj.inj2 -s
    ${splitDir} -o ${outDir}
```

  where under the subdirectory `splitDir` resides the `PPIcorpus` directory, which includes the corresponding splits for the `PPIcorpus.xml`, files names `PPIcorpus1.txt` .. `PPIcorpus10.txt`

- Output: under the directory specified by `-o` another directory named `MOSCHITTI` (for `-m` flag) and `CUSTOM_KERNEL` (for `-c` flag), under which a third directory `PPIcorpus.xml-folds` is created, which contains 10 text files (named: `0.txt` ... `9.txt`).

Remark: There is only a slight difference between the two format (Moschitti and custom), namely that the Moschitti based SVM learners require an embedding `|BT|..|ET|` tags around the training parsed tree instances, while the custom format follows the requirement of T. Joachims' SVM^{light}.

D Details of learning format creation for kBSPS kernel

This program creates the 10-fold cross-validation data for the k-band shortest path spectrum kernel based SVM learner.

- Program: SvmLightDependencyTreeKernelTransformer.java
- Input: the original `PPIcorpus.xml`.
- Parameters:

  `-f|--file` input file name
  `-o|--out` output directory name
  `-s|--split` location of the split files (folder)
  `-qmin` Minimal length of q-grams (allowed range: $[1, 5]$, default: 2)
  `-qminmax` Maximum of minimal length of q-grams (allowed range: $[1, 5]$, default: 2)
  `-qmaxmin` Minimum of maximal length of q-grams (allowed range: $[1, 5]$, default: 2)
  `-qmax` Maximal length of q-grams (allowed range: $[1, 5]$, default: 2)
  `-k|--kvalue` Value of k (allowed range: [0,2], default: 0)

- Running from command line:

```
java -classpath '${LFJARGSCLASSPATH}:${LFCLASSPATH}'
    org.learningformat.transform.SvmLightDependencyTreeKernelTransformer -f PPIcorpus.xml --qmin 1
    --qminmax 2 --qmaxmin 2 --qmax 3 -k 1 -s ${splitDir} -o ${CVDir}
```

  which includes the corresponding splits for the `PPIcorpus.xml`, files names `PPIcorpus1.txt` .. `PPIcorpus10.txt`

- Output: under the directory specified by `-o` several other directories are created, under each of which another directory `PPIcorpus-folds` created, which contains 10 text files (named: `0.txt` ... `9.txt`). The name of second level directories follow the following pattern:
  `CUSTOM_KERNEL-b-`$q_{\min}$`to`$q_{\max}$`-k`k`-UP_TO-OUTSIDE-NO_SELF_REF-STEM-none` where $q_{\min} \leq q_{\max}$.

Remark: unlike for the syntax tree kernels, here strings are converted into numbers in order to speed up the learning process of the SVM learner.

E Database schema

The database schema separates annotation data from experimental settings and their results. Cross-validation and cross-learning and cross-corpus experiments are stored in separate sets of tables. A fragment of the schema is shown in Figure 1.

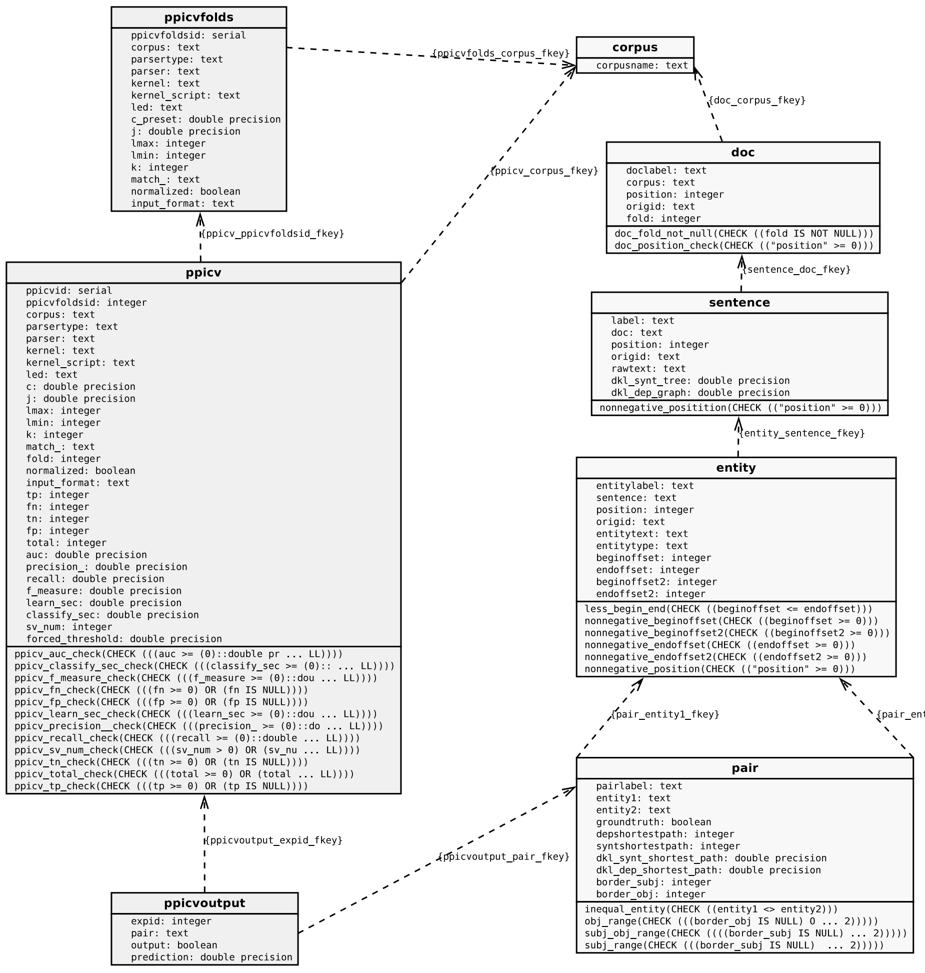
ppicvfolds
ppicvfoldsid: serial
corpus: text
parsertype: text
parser: text
kernel: text
kernel_script: text
led: text
c_preset: double precision
j: double precision
lmax: integer
lmin: integer
k: integer
match_: text
normalized: boolean
input_format: text
{ppicvfolds_corpus_fkey}
corpus
corpusname: text
{doc_corpus_fkey}
{ppicv_corpus_fkey}
doc
doclabel: text
corpus: text
position: integer
origid: text
fold: integer
doc_fold_not_null(CHECK ((fold IS NOT NULL)))
doc_position_check(CHECK (("position" >= 0)))
{ppicv_ppicvfoldsid_fkey}
{sentence_doc_fkey}
ppicv
ppicvid: serial
ppicvfoldsid: integer
corpus: text
parsertype: text
parser: text
kernel: text
kernel_script: text
led: text
c: double precision
j: double precision
lmax: integer
lmin: integer
k: integer
match_: text
fold: integer
normalized: boolean
input_format: text
tp: integer
fn: integer
tn: integer
fp: integer
total: integer
auc: double precision
precision_: double precision
recall: double precision
f_measure: double precision
learn_sec: double precision
classify_sec: double precision
sv_num: integer
forced_threshold: double precision
ppicv_auc_check(CHECK (((auc >= (0)::double pr ... LL))))
ppicv_classify_sec_check(CHECK (((classify_sec >= (0):: ... LL))))
ppicv_f_measure_check(CHECK (((f_measure >= (0)::dou ... LL))))
ppicv_fn_check(CHECK (((fn >= 0) OR (fn IS NULL))))
ppicv_fp_check(CHECK (((fp >= 0) OR (fp IS NULL))))
ppicv_learn_sec_check(CHECK (((learn_sec >= (0)::dou ... LL))))
ppicv_precision__check(CHECK (((precision_ >= (0)::do ... LL))))
ppicv_recall_check(CHECK (((recall >= (0)::double ... LL))))
ppicv_sv_num_check(CHECK (((sv_num > 0) OR (sv_nu ... LL))))
ppicv_tn_check(CHECK (((tn >= 0) OR (tn IS NULL))))
ppicv_total_check(CHECK (((total >= 0) OR (total ... LL))))
ppicv_tp_check(CHECK (((tp >= 0) OR (tp IS NULL))))
sentence
label: text
doc: text
position: integer
origid: text
rawtext: text
dkl_synt_tree: double precision
dkl_dep_graph: double precision
nonnegative_posititition(CHECK (("position" >= 0)))
{entity_sentence_fkey}
entity
entitylabel: text
sentence: text
position: integer
origid: text
entitytext: text
entitytype: text
beginoffset: integer
endoffset: integer
beginoffset2: integer
endoffset2: integer
less_begin_end(CHECK ((beginoffset <= endoffset)))
nonnegative_beginoffset(CHECK ((beginoffset >= 0)))
nonnegative_beginoffset2(CHECK ((beginoffset2 >= 0)))
nonnegative_endoffset(CHECK ((endoffset >= 0)))
nonnegative_endoffset2(CHECK ((endoffset2 >= 0)))
nonnegative_position(CHECK (("position" >= 0)))
{pair_entity1_fkey}
{pair_entity2_fkey}
{ppicvoutput_expid_fkey}
{ppicvoutput_pair_fkey}
pair
pairlabel: text
entity1: text
entity2: text
groundtruth: boolean
depshortestpath: integer
syntshortestpath: integer
dkl_synt_shortest_path: double precision
dkl_dep_shortest_path: double precision
border_subj: integer
border_obj: integer
inequal_entity(CHECK ((entity1 <> entity2)))
obj_range(CHECK (((border_obj IS NULL) O ... 2)))))
subj_obj_range(CHECK ((((border_subj IS NULL) ... 2)))))
subj_range(CHECK (((border_subj IS NULL) ... 2)))))
ppicvoutput
expid: integer
pair: text
output: boolean
prediction: double precision

Figure 1: Fragment of the databse schema. Corpora and CV experiments related tables are drawn with light and dark colors respectively.