chem-bla-ics

pKa prediction, or, how to convert a JCIM paper into Java

Egon Willighagen

Published October 6, 2008

Citation
Willighagen, E. (2008). pKa prediction, or, how to convert a JCIM paper into Java. In *chem-bla-ics*. chem-bla-ics. https://doi.org/10.59350/mnn5d-bwg40

Keywords
Cdk, Chemistry

Abstract
Lee et al. published last week a paper on pKa prediction (doi:10.1021/ci8001815). As the paper says, the pKa, and in particular the ionic state of a molecule at physiological pH, affects pharmacokinetics and pharmacodynamics. The paper describes a (binary) decision tree using presence or absence of SMARTS substructures to traverse the tree, allowing prediction of monoprotic molecules.

Lee et al. published last week a paper on pK_a prediction (doi:10.1021/ci8001815). As the paper says, *the pKa, and in particular the ionic state of a molecule at physiological pH, affects pharmacokinetics and pharmacodynamics*. The paper describes a (binary) decision tree using presence or absence of SMARTS substructures to traverse the tree, allowing prediction of monoprotic molecules.

Now, the paper's Supplementary Information contains the full model. I'd rather rebuild the model, but the full training set does not seem available. Still, the paper's model shows comparible predictive power as commercial models, so I'd say it would be a welcome addition to the CDK.

And as the CDK already has a SMARTS parser, adding this model should be easy enough. So, here goes :) First, let us outline the API:

```
/* $Revision$ $Author$ $Date$
 *
 * Copyright (C) 2008  Egon Willighagen <egonw@users.sf.net>
 *
 * Contact: cdk-devel@list.sourceforge.net
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public License
 * as published by the Free Software Foundation; either version 2.1
 * of the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA.
 */
package org.openscience.cdk.charges.pka;

/**
 * Tool to predict a molecule's pKa. The class implements
 * the algorithm published by Lee et al. {@cdk.cite Lee2008}
 * which is based on a SMARTS-based decision tree, trained
 * with 1693 monoprotic compounds.
 *
 * @cdk.module extra
 */
```

```
public class PkaPredictor {

  /**
   * Predicts the pKa value of a molecule.
   *
   * @param  container IMolecule to predict the pKa for
   * @return           The predicted pKa
   *
   * @throws CDKException upon failure of the prediction algorithm
   */
  public static float predict(IMolecule container) throws CDKException {}

}
```

The first line is picked up by SVN, which will add the revision number, the last commiter and when the last commit happened. The third line is important: it indicates who has the right or need to be asked permission to modify the license, if ever needed. If people provide patches to the code, they are added to this list. The rest of the source file header includes a general contact email address, and the *LGPL v2* license the CDK uses. The *package* declaration puts it in the *cdk.charges.pka* package, which seemed appropriate.

The class JavaDoc contains two CDK specific tags. The tag `${cdk.cite Lee2008}` is used to point to the literature reference database in `doc/refs/cheminf.bibx`. When the HTML JavaDoc is compiled, the full reference gets included in the HTML. The other tag, `@cdk.module` is used by the CDK build system to determine in which CDK module the Class should end up; `extra` in this case. The method's JavaDoc is pretty default.

Next, we need some logic to traverse the look up the predicted pK_a from the decision tree, and I implemented this as:

```
public static float predict(IMolecule container) throws CDKException {
  if (node1 == null) initalize();

  DecisionTreeNode node = node1;
  // traverse down tree until we end up in a leave
  while (!node.isTerminal()) {
    node = node.decide(container);
  }
  return node.getValue();
}
```

The root node of the tree is called `node1`, and I explain its initialization later. Then, the code traverses the tree by asking each node to decide whether the SMARTS substructure is present or not. It returns a new `DecisionTreeNode` matching the presence or absence. At some point, the terminal node is reached, and we can ask this node the associated prediction value.

The Java version of the Decision Tree

The paper's supplementary information contains the tree encoded like this:

```
1,0,,,5.9131093
2,1,[#G6H]C(=O),1,3.6849957
3,1,[#G6H]C(=O),0,7.206913
```

That is, each line lists the node identifier, the parent identifier, the SMARTS query, presence (1) or absence (0), and the node value. Actually, a bit more, but these are the important bits for now. The first line is the root node `node1`, and the second and third line the two children of the root node. If the `[#G6H]C(=O)` substructure is present, then `node2` applies, and the predicted value would be 3.6849957; if the substructure is absent, then `node3` applies, and pK_a 7.206913.

Now, these nodes and there interdepencies are encoded in the initialize() method as:

```
private static void initalize() throws CDKException {
  node1 = new DecisionTreeNode(5.9131093f, 17.32f);
  DecisionTreeNode node2 = new DecisionTreeNode(3.6849957f, 5.9569998f);
  DecisionTreeNode node3 = new DecisionTreeNode(7.206913f, 17.32f);
  node1.setChildNodes("[#G6H]C(=O)", node2, node3);
}
```

The second argument in the `DecisionTreeNode` constructor is the value range for the node, and is an indication of the variance of the prediction value.

A simple Perl script can convert the file from the supplementary information into Java source code. With more than 1500 nodes in the tree, this beats manual hacking up of the tree.

The JUnit4 test

The unit tests now looks like:

```
package org.openscience.cdk.charges.pka;

/**
 * Unit test to test the functionality of the {@link PkaPredictor}.
 *
 * @author     egonw
 * @cdk.module test-extra
 */
public class PkaPredictorTest extends NewCDKTestCase {

  @Test public void testThrowsNoException() throws Exception {
    IMolecule methane = NoNotificationChemObjectBuilder.getInstance().newMolecule();
    IAtom carbon = methane.getBuilder().newAtom(Elements.CARBON);
```

```
    methane.addAtom(carbon);

    float result = PkaPredictor.predict(methane);
    // the actual value depends on the number of nodes I actually added,
    // but I *do* know the min and max without having to have all nodes
    // implemented
    Assert.assertTrue(result < 15.526);
    Assert.assertTrue(result > -0.6659999);
  }
}
```

Note that I cannot assert the real prediction value until the full decision tree has been implemented in the class, but I do note the full range and thus test for that. You may have noted that several methods throw `CDKException`'s, which would have been caused by SMARTS expressions the CDK cannot handle...

SMARTS problems...

Now, the SMARTS used in the supplementary information indeed do not work with the CDK SMARTS engine; the paper indicates that they used MOE which extends the original Daylight SMARTS. So, if you ever wondered about the forking risk of Open Standards...

So far, I have identified these three patterns used in the paper's model, but not parsable by the CDK engine:

1. **[i]** a SP2 hybridized carbon (aromatic or delocalized)
2. **[#G6]** matches carbon and sulfur, so seems to indicate a group in the periodic table
3. **[#X]** no idea... (no internet at home yet, so cannot Google either)

The #G syntax can be rewritten in a OR form, and possible the others too. However, I'd rather see the CDK SMARTS engine support these industry adopted extensions.

Conclusion

The CDK shows its power as *development kit*, and allowed me to hack up the code of the paper on a casual Saturday evening (sitting on the couch next to a fire in our kacheloven with a glass of beer). Writing up this blog was done the next day.

Once the missing SMARTS patterns have been added to the CDK (or proper replacements have been defined), I'll compare the test set results of the paper with the CDK implementation. I probably also convert the test set results from the supplementary information into unit tests (the SI contains SMILES, experimental and predicted values).