Introduction to BIOCAT algorithms

There are 4 types of algorithms that are used in BioCAT.

1. **2D Feature extractor algorithms** find features in a 2D image and transform those features into useable data.
2. **3D Feature extractor algorithms** find features in a 3D image set and transform those features into useable data.
3. **Feature selector algorithms** take the data produced by the feature extractor and choose what data is relevant to the options that the user has chosen.
4. **Classifier algorithms** take data from either a feature extractor or selector and calculate the class that the data belongs to and optionally the probability of the classification to be correct. Some classifier can be savable, that is the data model can be saved for later use.

The algorithm plugins implement the following Java interfaces:

- For the 2D and 3D feature extractors: annotool.extract.FeatureExtractor.
- For the feature selectors: annotool.select.FeatureSelector.
- For classifiers: annotool.classify.Classifier interface, or in the case of a savable classifier algorithm, annotool.classify.SavableClassifier.

Writing an algorithm plugin

Feature Extractor Algorithm Example

```
/**
 * FeatureExtractor finds features and saves them to a two dimensional array
 */
public interface FeatureExtractor
{

   /**
    * Sets algorithm parameters from para
    *
    * @param  para  Each element of para holds a parameter's name for its key
    *               and a parameter's value for its value. The parameters
    *               should be the same as those in the algorithms.xml file.
    */
   public void  setParameters(java.util.HashMap<String, String> para);

   /**
    * Get features based on raw image stored in problem.
    *
    * @param   problem    Image to be processed
    * @return             Array of features
    * @throws  Exception  Optional, generic exception to be thrown
    */
   public float[][] calcFeatures(DataInput problem) throws Exception;
```

```
// The DataInput class encapsulates the data of an image. It allows for a
// simple interface to the image's properties via member functions. Member
// functions allow for reading image properties, data, and images from a
// stack. For further reference, see the DataInput API.

   /**
    * Get features based on data, imageType, and dim.
    * <p>
    * ArrayList of array can accommodate FloatProcessor and IntProcessor
    * 09/02/2011
    *
    * @param   data       Data taken from the image
    * @param   imageType  Type of the image
    * @param   dim        Dimenstions of the image
    * @return             Array of features
    * @throws  Exception  Optional, generic exception to be thrown
    */
   public float[][] calcFeatures(ArrayList data, int imageType, ImgDimension
dim) throws Exception;

   /**
    * Returns whether or not the algorithm is able to extract from a 3D image
    * stack.
    *
    * @return  <code>True</code> if the algorithm is a 3D extractor,
    *          <code>False</code> if not
    */
   boolean is3DExtractor();
}
```

Feature Selector Algorithm Example

```
/**
 * FeatureSelector selects features that are accurate and of interest
 */
public interface FeatureSelector
{
   /**
    * Sets algorithm parameters from para
    *
    * @param   para  Each element of the hashmap holds a parameter name
    *                for its key and a its value is that of the parameter.
    *                The parameters should be the same as those in the
    *                algorithms.xml file.
    */
   public void setParameters(java.util.HashMap<String, String> para);

   /**
    * Stores features from data and store any relevant values such as
    * the dimensions of data and targets.
    *
    * @param   data     Array of extracted image data
    * @param   targets  Array of the targets for the image
    * @return           Array of features that are selected
    */
```

```
   public float[][] selectFeatures(float[][] data, int[] targets);


   /**
    * Returns the indices of the selected features
    * (an index is between 0 and one less than the number of features).
    *
    * @return  The indices of the selected features.
    */
   public int[] getIndices();

   /**
    * Selects features using “indices” and returns the selected features.
    *
    * @param   data     Array of extracted image data
    * @param   indices  Array of indices to the data columns
    * @return           Array of features that are selected
    */
   public float[][] selectFeaturesGivenIndices(float[][] data, int []
indices);

}
```

Classifier Algorithm Example

```
/**
 * Classifier returns a prediction based on training and testing image sets.
 *
 * @see  SavableClassifier
 */
public interface Classifier
{
   /**
    * Sets algorithm parameters from para
    *
    * @param   para  Each element of para holds a parameter name
    *                for its key and a its value is that of the parameter.
    *                The parameters should be the same as those in the
    *                algorithms.xml file.
    */
   public void setParameters(java.util.HashMap<String, String> para);

   /**
    * Classifies the patterns using the input parameters.
    *
    * @param   trainingpatterns  Pattern data to train the algorithm
    * @param   trainingtargets   Targets for the training pattern
    * @param   testingpatterns   Pattern data to be classified
    * @param   predictions       Storage for the resulting prediction
    * @param   prob              Storage for probability result
    * @throws  Exception         Optional, generic exception to be thrown
    */
   public void classify(float[][] trainingpatterns, int[] trainingtargets,
float[][] testingpatterns, int[] predictions, double[] prob) throws
Exception;
```

```
   /**
    * Returns whether or not the algorithm uses probability estimations.
    *
    * @return  <code>True</code> if the algorithm uses probability
    *          estimations, <code>False</code> if not
    */
   boolean doesSupportProbability();
}
```

Savable Classifier Algorithm Example

```
/**
 * Savable classifier returns a model (an object of a Serializable class),
 * so that the classifier may be persisted and loaded back to memory for use.
 *
 * @see  Classifier
 */
public interface SavableClassifier extends Classifier
{
   /**
    * Trains and returns an internal model using a training set.
    *
    * @param   trainingpatterns  Pattern data to train the algorithm
    * @param   trainingtargets   Targets for the training pattern
    * @return                    Model created by the classifier
    * @throws  Exception         Optional, generic exception to be thrown
    */
     Object trainingOnly(float[][] trainingpatterns, int[] trainingtargets)
throws Exception;

   /**
    * Gets the internal model from the classifier
    *
    * @return  Model created by the classifier.
    */
     Object getModel();

   /**
    * Sets an internal model to be used by the classifier
    *
    * @param   model      Model to be used by the classifier
    * @throws  Exception  Exception thrown if model is incompatible
    */

     void setModel(java.lang.Object model) throws Exception;
```

```
    /**
     * Classifies the internal model using one testing pattern
     *
     * @param   model            Model to be used by the classifier
     * @param   testingPattern   Pattern data to be classified
     * @param   prob             Storage for probability result
     * @return                   The prediction result
     * @throws  Exception        Exception thrown if model is incompatible
     */
      int classifyUsingModel(Object model, float[] testingPattern, double[]
prob) throws Exception;

    /**
     * Classifies the internal model using multiple testing patterns
     *
     * @param   model             Model to be used by the classifier
     * @param   testingPatterns   Pattern data to be classified
     * @param   prob              Storage for probability result
     * @return                    Array of prediction results
     * @throws  Exception         Exception thrown if model is incompatible
     */
      int[] classifyUsingModel(Object model, float[][] testingPatterns,
double[] prob) throws Exception;

    /**
     * Saves a specified model to a specified file
     *
     * @param   trainedModel     Trained model that is to be saved
     * @param   model_file_name  Name of the file to be saved to
     * @throws  Exception        Exception thrown if model cannot be saved
     */
    void saveModel(Object trainedModel, String model_file_name) throws
java.io.IOException;

    /**
     * Loads a previously saved model back into the classifier.
     *
     * @param   model_file_name  Name of the file to be loaded
     * @return                   Model that was loaded
     * @throws  Exception        Exception thrown if file cannot be found
     */
    Object loadModel(String model_file_name) throws java.io.IOException;
}
```

How to add an algorithm to BioCAT as a plugin:

1. Navigate to the plugins folder of **BioCAT**. If it is not created yet, create a folder named plugins in the root folder of **BioCAT**.
2. Create a new folder, within the plugins folder, for the new algorithm.

3. Move the algorithm's class files and any jar files into the new folder. If the class has a package structure, the corresponding subdirectories need to be generated.
4. Create an **algorithm.xml** file within the algorithm's folder.
5. Refer to the section below on what to write into the **algorithm.xml** file.

How the algorithm.xml file works:

The **algorithm.xml** file defines the user interface options that are shown when the algorithm is chosen for use. In turn, these options are turned into a special parameter list for the algorithm.

The **algorithm.xml** file's format is the same as a system-wide **Algorithms.xml** file (the latter contains the default built-in algorithms and typically resides at the home folder). Such xml file(s) can have varying numbers of algorithms and parameters. Each parameter can have a default, domain, min, or max tags with the respective value in between the tags.

Here is an example of the **algorithm.xml** file:

```
<Algorithms>
   <Algorithm type="Selector"> <- This is a "Selector" algorithm (Can be
                                  Selector, Classifier, 2DExtractor, or
                                  3dExtractor)

      <Name>Example</Name> <- The algorithm name is "Example"(This is
                               (This is what will show up in GUI)

      <Desc>Description of Example</Desc> <- The description of the algorithm

      <ClassName>Example</ClassName> <- The fully qualified class name of
                                        the algorithm or the jar file that
                                        contains the algorithm

      <Path>Example/</Path> <- The file path of the algorithm

      <Parameter type="String"> <- The parameter is a "String"
         <Name>Method</Name> <- The parameter name is "Method"
         <Domain>A,B,C,D</Domain> <- 'A'-'D' are the possible values
                                     (Only for Strings)
         <Default>A</Default> <- 'A' is set as the default value
      </Parameter>

      <Parameter type="Integer"> <- The parameter is an "Integer"
         <Name>Number</Name> <- The parameter name is "Number"
         <Default>8</Default> <- '8' is set as the default value
         <Min>2</Min> <- '2' is the minimum value (Only for Integers)
         <Max>32</Max> <- '32' is the maximum value (Only for Integers)
      </Parameter>
   </Algorithm>
</Algorithms>
```