Extending Classifiers for Incomplete Data

A dissertation submitted in partial fulfillment of
the requirements for the degree of
BACHELOR OF ENGINEERING in Computer Science
in
The Queen's University of Belfast
by

Christopher McKee

February 27, 2017

SCHOOL OF ELECTRONICS, ELECTRICAL ENGINEERING and COMPUTER SCIENCE

CSC3002 – COMPUTER SCIENCE PROJECT

Dissertation Cover Sheet

A signed and completed cover sheet must accompany the submission of the Software Engineering dissertation submitted for assessment.

Work submitted without a cover sheet will **NOT** be marked.

Student Name: Student Number:

Project Title:

Supervisor:

Declaration of Academic Integrity

Before signing the declaration below please check that the submission:

1. Has a full bibliography attached laid out according to the guidelines specified in the Student Project Handbook
2. Contains full acknowledgement of all secondary sources used (paper-based and electronic)
3. Does not exceed the specified page limit
4. Is clearly presented and proof-read
5. Is submitted on, or before, the specified or agreed due date. Late submissions will only be accepted in exceptional circumstances or where a deferment has been granted in advance.

I declare that I have read both the University and the School of Electronics, Electrical Engineering and Computer Science guidelines on plagiarism - http://www.qub.ac.uk/schools/eeecs/Education/StudentStudyInformation/Plagiarism/ - and that the attached submission is my own original work. No part of it has been submitted for any other assignment and I have acknowledged in my notes and bibliography all written and electronic sources used.

Student's signature *Date of submission*

Abstract

Abstract goes here

Contents

Chapter 1

Introduction

The reliability of a machine learning model is heavily dependent on the data which was used to train it. Training data sets with missing values are therefore an interesting problem since there is no way of knowing what any given missing value could have been, and so the only two useful approaches involve either making statistical approximations of missing or ignoring any record which contains missing data. Both mentioned approaches have positives as well as negatives. Ignoring a record prevents you from using unreliable data, but also may lead to valuable data in the other, non-missing attributes of the same record being discounted. Attempts to predict a missing value have no guarantee of being correct, but may be worth the risk to gain value from the other complete attributes. Using either approach may produce false outcomes. It is therefore extremely important that any imputation is carried out with care, and it must be very likely to be correct.

This project aims to investigate the usage of machine learning techniques on incomplete sets of discrete data. It will attempt to impute any missing values in the training data, by iteratively building and applying models until no further change is observed. It is expected that predicted missing values will trend towards the mean, leading to more reliable classification when this trained model is then applied to target data. Given sufficient time, it is also intended that this approach can be used to infer 'hidden variables', similarly to how neural networks produce hidden layers to create relationships between data. This will be achieved by adding a new attribute to a data set, and initially filling it full of random valid data. We will then repeatedly build, apply and rebuild models to classify this attribute until it stops changing, in an identical manner to how missing data is imputed in the training set. New data will be added to the training data set by this method, and this project aims to determine whether or not these hidden variables lead to a greater classification accuracy.

The goal of this project is to study the effects of iteratively building common classifiers and comparing the performance of classifiers built this way against the performance of the same classifier, but without attempting to impute missing training data. The main aims of the software to be developed will thereore be:

- A Weka plugin will be developed. This should integrate smoothly into any version of Weka >3.6, and will be made freely available.
- The classifier developed within the plugin should be usable from both the CLI and GUI.
- Running experiments using the plugin should be easy, and have well explained documen-

tation.

- Allow any currently implemented Weka classifier which is valid to use for a given dataset to be used upon it, and for this to be easily configurable.
- Run a number of experiments to compare the performance of the iteratively trained classifier against some control cases on the same data sets.

Chapter 2

Design and Implementation

2.1 Design

This body of work is mainly interested in common classifiers, and in trying to use them in a novel way. It therefore makes sense to extend a package in which these are already implemented, and in which they are easily accessible. The Weka project therefore suits this purpose since it has well maintained, efficient implementations of most machine learning algorithms, as well as handling file I/O, and having both command line and graphical user interfaces. Since these are written in Java, it follows that the plugin should also be written in Java.

It also follows that any files which are used for testing should be in either .csv, or .arff format. The former is commonly used in data processing while the latter is a proprietary Weka format which is very similar, but differs in that it contains additional data at the top of the file pertaining to the dataset. Both are supported by Weka, and therefore either is usable.

Since the project relies upon the idea that predictions for missing training data values should converge, it is assumed that the developed classifier will initially only work on nominal data sets. This is because nominal data sets should eventually settle on concrete values, whilst numeric values are not guaranteed to every settle on a particular value at all. It is more likely that numeric predicitions would just change less and less between iterations, until the changes were miniscule. While it would be possible to implement a solution which stops iterating once a numeric data set only changes to a certain degree, as a proof of concept it is sensible to stick to nominal data.

It is intended that the implemented algorithm will work as follows:

1. Take a given training set, and make a copy of it
2. Fill the copy set with random, viable values
3. Use the produced data set to train a new model
4. Use the newly trained model to impute values into a new copy of the training set
5. Repeat 3 and 4 until identical data sets are produced by two consecutive iterations
6. Use the model built by the final iteration for classification of a test set

2.2 Implementation

As an existing piece of software is being extended, much of the work to implement the various classifiers to be tested is already done. The main aim of this work is to integrate successfully into Weka as a plugin, and to interface with it as necessary.

2.2.1 Weka Architecture

Weka is a Java package, with a source code layout which looks somewhat like this if unused packages are omitted:

```
weka
├── associations
├── attributeSelection
├── classifiers
│   ├── AbstractClassifier
│   ├── IterativeClassifier
│   └── SingleClassifierEnhancer
├── clusterers
├── core
│   ├── Attribute
│   ├── Capabilities
│   ├── Instance
│   ├── Instances
│   ├── Option
│   └── Utils
├── datagenerators
├── estimators
├── experiment
├── filters
├── gui
└── knowledgeflow
```

Each of these subpackages is reasonably self explanatory. This project is mostly concerned with classifications, and thus code implemented for it lives in the *weka.classifiers* package, although there are some dependcies on *weka.core* since it contains much of the shared code for Weka. Provided that the rest of the project works as intended and the documentation is followed, there is no need to modify or discuss the other packages.

2.2.2 weka.core

From *weka.core*, the following classes are used:

- **Instances** - An Instances object in Weka is used to represent a complete data set. It is roughly equivalent to the object representation of an ARFF file. It is comprised of a list

of Instance objects, a list of Attribute objects and some other metadata such as the class attribute for the data set and its name.

- **Attribute** - An Attribute in Weka refers to a single column in a data set. This object is mostly responsible for tracking the column's data type (e.g. nominal, numeric, etc), and other related information common to the entire column.
- **Instance** - An Instance is the representation of a single row of data, roughly analogous to a row in a CSV file. Belongs to a particular Instances object. Each attribute value is stored as a Java double, which is then used in connection with an Attribute object to determine the actual value of an attribute.
- **Option** - An Option object represents a particular command line parameter, and the way in which it should be handled. This also contains a description to be printed in the help dialog if incorrect parameters are passed to the CLI.

2.2.3 AbstractClassifier

AbstractClassifier is the class from which all classifiers which make nominal or numeric predictions in Weka (citation? this is nearly word for word from the Weka javadoc) are derived from. It provides a number of helpful implementations to prevent code duplication in classifiers, and to ease implementation. A pseudocode description of some notable methods can be seen below:

```
abstract class AbstractClassifier {
    forName(classifierName, options) {
        return an instance of classifier <classifierName> with options <option>
    }

    runClassifier(classifier, options) {
        run any classifier by passing options to it
    }

    classifyInstance(instance) {
        returns the class which the instance is most
            likely to belong to as a double in Weka's internal representation
    }

    distributionForInstance(instance) {
        return a list of probabilities for each
            possibile class value for a given instance
    }

    listOptions() {
        return all of the command line options which
            a classifier accepts to configure how it will run
    }

    getOptions() {
        return the list of options which a classifier
            is currently set up to run with
    }

    setOptions(options) {
        sets the command line options for a
            classifier to options
```

```
    }

    getCapabilities() {
        returns a list of the attribute and class attribute
            types which are supported by this classifier
    }
}
```

There are a number of other utility and stub methods within this abstract class which may prove useful for other people in other cases, but have been omitted from the description above.

2.2.4 SingleClassifierEnhancer

SingleClassifierEnhancer is a concrete class which extends from *AbstractClassifier*, and from which the implemented ProjectClassifier extends. Extending from this class gives us a couple of benefits:

- *SingleClassifierEnhancer* is designed for 'meta' classifiers, or those which wrap another classifier. It already has an implementation classifier in which it wraps an internal classifier and handles options to it. This can easily be extended to copy this classifier multiple times with the same options
- The internal classifier is already displayed nicely in the Weka GUI. This allows it to be configured via a drop down menu, as shown in figure (put screenshot here?)
- It already handles some edge cases, such as calling the *preExecution()* and *postExecution()* methods of the internal classifier, which handles any specific setup and cleanup needed. These are small edge cases that would be easy to forget about, so extending in this way improves reliability by removing some of the implementation overhead.

2.2.5 IterativeClassifier

IterativeClassifier is an Interface which is provided by Weka for 'classifiers which build models of increasing complexity' (Javadoc citation?). It provides three method signatures:

```
public initializeClassifier() {
    do setup for the classifier
}

public next() {
    increase in complexity by one step by retraining the model
    return false if finished, or true if another step can be taken
}

public done() {
    stop increasing complexity and do any necessary cleanup
}
```

These methods generally come together in the following pattern in a classifier which implements the interface:

```
public buildClassifier(instances) {
    // code
    initializeClassifier(instances)
    while(next()) {}
    done()
    // code
}
```

This means that building the classifier continues until it no longer makes sense to keep doing so, or no gain is achieved. This suits the project's use case, and therefore the interface makes sense to use.

2.2.6 Weka Interfaces

Calling a Weka classifier from the command line (assuming that the Weka jar file is already on the classpath) is reasonably straightforward. The target classifier is firstly called by its full path e.g. *java weka.classifiers.trees.J48*. Anything which comes after this point is passed to the classifier as an Option object. These are used as parameters to configure the behaviour of a particular classifier, and each class which extends from *AbstractClassifier* generally implements its own specific set of Options through a combination of private variables and specifically named methods. The GUI appears works in much the same way, except that it provides windows with dropdown menus and text fields to allow classifiers and their Options to be configured.

For example, imagine a given classifier had a private variable named *m_Example*. In order to expose this to the GUI and CLI it would require getter and setter methods, as well as an *exampleTipText()* method to display help text. Code specifying how to handle this Option would also need to be added to the *getOptions()*, *setOptions()* and *listOptions()* methods. (paragraph too vague / not enough detail)

2.2.7 StateAnalyser

The *StateAnalyser* class has been designed to track the progress of the *ProjectClassifier* as it is being trained. It contains a list of the training sets which have been produced by the *next()* method, and records the number of differences between them. All of the methods described below are just utility methods for working with or comparing the training sets which have been produced.

```
private convertToMatrix(instances) {
    create a matrix
    for each row in the data set {
        turn it into an array of floats
        add that array to the matrix
    }
    return matrix
}

public getNumberDifferences() {
    if at least two training sets recorded {
        convert the two most recent training sets to matrixes
        for each row in both matrices {
```

```
            if the two rows differ, increment the difference counter
        }

        return the number of different rows;
    }

    else
        return a negative number to indicate an invalid test;
}


public getNumberIterations() {
    return number of recorded sets
}
```

2.2.8 ProjectClassifier

The *ProjetClassifier* class is the implementation of the algorithm described in section 2.1. This is extends from *SingleClassifierEnhancer* and implements the *IterativeClassifier* interface. It maintains an internal array of classifiers for every attribute in a data set, and uses these to impute values into the training set as well as to classify test data. It also maintains three copies of training set while building:

- *original* - a copy of the training set
- *last* - the last data set produced by imputing missing values, which can be used for retraining during the next iteration
- *current* - the data set which is currently having its missing values filled in

```
public buildClassifier(instances) {
    record the index of the original class attribute
    take a reference copy of the training set (original)
    find the location of missing data in the training set
    set up the classifier for the first run
    repeat until no change is observed between current and last
    perform any cleanup needed
}

public findMissingAttributes(instances) {
    for each instance in a training set {
        record any values which are missing for later use
    }
}

public initializeClassifier(instances) {
    create new tracker object
    create one classifier object per attribute in the data set
    take two modifiable copies of the training data set (last/current)
    replace any missing data in last using random values
}

public replaceMissingValues(instances) {
    for any each piece of missing data {
        input a random possible value
```

```
    }
}

public next() {
    replace last with current
    replace current with a new copy of the training data
    retrain classifiers against last
    for each recorded missing value {
        use trained classifiers to impute missing values in current
    }
    add current to the tracker object
    if (current == last)
        stop iterating
    else
        iterate again
}

public done() {
    do option specific cleanup
}

public retrainClassifiers(instances) {
    for each attribute in the training set {
        retrain a classifier against that attribute
    }
}
```

ProjetClassifier also accepts some options, which can be used to configure how it runs:

- **maxIterations** (-M number) - Some classifiers may not converge in the way in which we expect, or they make take a very long time to reach this point. This flag allows for a maximum number of training iterations to be set in order to ensure experiments finish in a timely fashion.
- **supervised** (-S) - It may be interesting to observe the difference between classifiers which are trained against a training set where the class attribute is trained iteratively along with the rest of the data set against a training set where missing class attribute values are not imputed until the classifier rest of the classifier has converged. This flag allows for the latter case to be tested.
- **randomData** (-R) - As a control set during experiments, it make sense to fill a training set with random data and determine whether or not the implemented algorithm outperforms classifiers which are trained against this. This allows this experiment to be performed.
- **hiddenVariables** (-N) - See appendix B (or something, update later)

Stuff about : - array of classifiers - missing value array - variables used for tracking - find missing attributes - initializeClassifier - replaceMissingValues - next - done - retrainClassifiers - classify / distributionForInstance

Chapter 3

Experiments

3.1 Results

3.1.1 Naive Bayes

3.1.2 Bayesian Network

3.1.3 J48 Decision Tree

Chapter 4

Conclusion

Do result discussion stuff

Appendix A

JavaDoc

Contents

Chapter 1

Package cdc04

Package Contents *Page*

Classes

1.1 Class StateAnalyser

1.1.1 Declaration

```
public class StateAnalyser
 extends java.lang.Object implements java.io.Serializable
```

1.1.2 Constructor summary

StateAnalyser()

1.1.3 Method summary

addInstances(Instances)
getNumberDifferences()
getNumberIterations()

1.1.4 Constructors

- **StateAnalyser**

```
public StateAnalyser()
```

1.1.5 Methods

- **addInstances**

```
public void addInstances(weka.core.Instances toAdd)
```

- **getNumberDifferences**

```
public int getNumberDifferences()
```

- **getNumberIterations**

```
public int getNumberIterations()
```

Chapter 2

Package weka.classifiers.meta

Package Contents *Page*

Classes

2.1 Class ProjectClassifier

A classifier which is iteratively trained, imputing missing values into copies of the training data until no further change is observed. Builds one learner per attribute, and therefore can take quite a while to run. Valid options are:

```
-W classifier
Full path to the target classifier to use, e.g. weka.classifiers.trees.J48

-S
Defines whether or not the classifier will impute a value for the class
attribute as it trains.

-R
If set, the classifiers will be trained with any missing arguments filled in
by random data. The classifier will then only iterate once.

-M integer
Sets the maximum number of times that a particular classifier will iterate
before determining that it is trained.
```

Options after – are passed to the currently selected classifier.

2.1.1 Declaration

```
public class ProjectClassifier
```

extends weka.classifiers.SingleClassifierEnhancer **implements** weka.classifiers.IterativeClassifier

2.1.2 Constructor summary

ProjectClassifier() Constructor

2.1.3 Method summary

buildClassifier(Instances) Builds a set of classifiers based on the training data.
classifierOptionsTipText() Tip text to be displayed in the GUI for this property
classifyInstance(Instance) Classifies an instance.
defaultClassifierString() String describing default classifier.
distributionForInstance(Instance) Returns class probabilities for an instance.
done() Method called when iteration has terminated.
getCapabilities() Returns default capabilities of the classifier.
getClassifierOptions() Gets classifier options
getMaxIterations() Get the value of m_MaxIterations
getOptions() Gets the current settings of the Classifier.
getRandomData() Get the value of m_RandomData
getSupervised() Get the value of m_Supervised
globalInfo() Global information about the class
initializeClassifier(Instances) Makes copies of the training data which can be mutated, and initialise the array of Classifier objects
listOptions() Returns an enumeration describing the available options.
main(String[]) Main method for testing this class.
maxIterationsTipText() Tip text to be displayed in the GUI for this property
next() Retrains each of the classifiers, then attempts to impute missing data in a copy of the training data.
randomDataTipText() Tip text to be displayed in the GUI for this property
setClassifierOptions(String[]) Sets classifier options
setMaxIterations(int) Set the value of m_MaxIterations.
setOptions(String[]) Parses a given list of options.
setRandomData(boolean) Set the value of m_RandomData
setSupervised(boolean) Set the value of m_Supervised
supervisedTipText() Tip text to be displayed in the GUI for this property

2.1.4 Constructors

- **ProjectClassifier**

  public ProjectClassifier()

  - **Description**
    Constructor

2.1.5 Methods

- **buildClassifier**

  ```
  public void buildClassifier(weka.core.Instances data) throws
      java.lang.Exception
  ```

  - **Description**
    Builds a set of classifiers based on the training data. These are iteratively trained on copies of the data.
  - **Parameters**
    * `data` – the Instances object which comprises the training data
  - **Throws**
    * `java.lang.Exception` – exception thrown is raised to a Weka error handler

- **classifierOptionsTipText**

  ```
  public java.lang.String classifierOptionsTipText()
  ```

  - **Description**
    Tip text to be displayed in the GUI for this property
  - **Returns** – tip text to be displayed in the GUI

- **classifyInstance**

  ```
  public double classifyInstance(weka.core.Instance instance)
      throws java.lang.Exception
  ```

  - **Description**
    Classifies an instance.
  - **Parameters**
    * `instance` – the instance to classify
  - **Returns** – the classification for the instance
  - **Throws**
    * `java.lang.Exception` – if instance can't be classified successfully

- **defaultClassifierString**

  ```
  protected java.lang.String defaultClassifierString()
  ```

  - **Description**
    String describing default classifier.

- **distributionForInstance**

```
public double[] distributionForInstance(weka.core.Instance
    instance) throws java.lang.Exception
```

  - **Description**
    Returns class probabilities for an instance.
  - **Parameters**
    - `instance` – the instance to calculate the class probabilities for
  - **Returns** – the class probabilities
  - **Throws**
    - `java.lang.Exception` – if distribution can't be computed successfully

- **done**

```
public void done() throws java.lang.Exception
```

  - **Description**
    Method called when iteration has terminated. Imputes class values if m_Supervised is set.

- **getCapabilities**

```
public weka.core.Capabilities getCapabilities()
```

  - **Description**
    Returns default capabilities of the classifier.
  - **Returns** – the capabilities of this classifier

- **getClassifierOptions**

```
public java.lang.String[] getClassifierOptions()
```

  - **Description**
    Gets classifier options
  - **Returns** – array of String objects to be passed to each classifier

- **getMaxIterations**

```
public int getMaxIterations()
```

  - **Description**
    Get the value of m_MaxIterations
  - **Returns** – value of m_MaxIterations

- **getOptions**

```
public java.lang.String[] getOptions()
```

  - **Description**
    Gets the current settings of the Classifier.
  - **Returns** – an array of strings suitable for passing to setOptions

- **getRandomData**

```
public boolean getRandomData()
```

  - **Description**
    Get the value of m_RandomData
  - **Returns** – value of m_RandomData

- **getSupervised**

```
public boolean getSupervised()
```

  - **Description**
    Get the value of m_Supervised
  - **Returns** – value of m_Supervised

- **globalInfo**

```
public java.lang.String globalInfo()
```

  - **Description**
    Global information about the class
  - **Returns** – information about the classifier which is displayed in the CLI/GUI

- **initializeClassifier**

```
public void initializeClassifier(weka.core.Instances instances)
  throws java.lang.Exception
```

  - **Description**
    Makes copies of the training data which can be mutated, and initialise the array of Classifier objects
  - **Parameters**
    * `instances` – the training data

- **listOptions**

```
public java.util.Enumeration listOptions()
```

  - **Description**
    Returns an enumeration describing the available options.
  - **Returns** – an enumeration of all the available options.

- **main**

```
public static void main(java.lang.String[] args)
```

  - **Description**
    Main method for testing this class.
  - **Parameters**
    * `args` – the options

- **maxIterationsTipText**

```
public java.lang.String maxIterationsTipText()
```

  - **Description**
    Tip text to be displayed in the GUI for this property
  - **Returns** – tip text to be displayed in the GUI

- **next**

```
public boolean next() throws java.lang.Exception
```

  - **Description**
    Retrains each of the classifiers, then attempts to impute missing data in a copy of the training data. Does not iterate again if the results of current iteration match the results of the previous iteration, or the max number of iterations has been reached.
  - **Returns** – true if another iteration should be performed, otherwise false.

- **randomDataTipText**

  ```
  public java.lang.String randomDataTipText()
  ```

  - **Description**
    Tip text to be displayed in the GUI for this property
  - **Returns** – tip text to be displayed in the GUI

- **setClassifierOptions**

  ```
  public void setClassifierOptions(java.lang.String[]
      classifierOptions)
  ```

  - **Description**
    Sets classifier options
  - **Parameters**
    * `classifierOptions` – array of String objects to be passed to each classifier

- **setMaxIterations**

  ```
  public void setMaxIterations(int maxIterations)
  ```

  - **Description**
    Set the value of m_MaxIterations. Defaults to Integer.MAX_VALUE if value less than 0 is supplied.
  - **Parameters**
    * `maxIterations` – new value of m_MaxIterations

- **setOptions**

  ```
  public void setOptions(java.lang.String[] options) throws java.
      lang.Exception
  ```

  - **Description**
    Parses a given list of options. Valid options are:
    -W classifier
    Full path to the target classifier to use, e.g. weka.classifiers.trees.J48
    -S
    Defines whether or not the classifier will impute a value for the class attribute as it trains.

-R
If set, the classifiers will be trained with any missing arguments filled in by random data. The classifier will then only iterate once.
-M integer
Sets the maximum number of times that a particular classifier will iterate before determining that it is trained.
Options after – are passed to the currently selected classifier.

- **Parameters**
  - `options` – The list of options as an array of Strings
- **Throws**
  - `java.lang.Exception` – if an option is not supported

- **setRandomData**

```
public void setRandomData(boolean randomData)
```

  - **Description**
    Set the value of m_RandomData
  - **Parameters**
    - `randomData` – new value of m_RandomData

- **setSupervised**

```
public void setSupervised(boolean supervised)
```

  - **Description**
    Set the value of m_Supervised
  - **Parameters**
    - `supervised` – the new value of m_Supervised

- **supervisedTipText**

```
public java.lang.String supervisedTipText()
```

  - **Description**
    Tip text to be displayed in the GUI for this property
  - **Returns** – tip text to be displayed in the GUI

2.1.6 Members inherited from class SingleClassifierEnhancer

weka.classifiers.SingleClassifierEnhancer

- public String **classifierTipText**()
- protected String **defaultClassifierOptions**()
- protected String **defaultClassifierString**()
- public Capabilities **getCapabilities**()
- public Classifier **getClassifier**()
- protected String **getClassifierSpec**()
- public String **getOptions**()
- public Enumeration **listOptions**()
- protected **m_Classifier**
- public void **postExecution**() throws java.lang.Exception
- public void **preExecution**() throws java.lang.Exception
- public void **setClassifier**(Classifier **arg0**)
- public void **setOptions**(java.lang.String[] **arg0**) throws java.lang.Exception

2.1.7 Members inherited from class AbstractClassifier

weka.classifiers.AbstractClassifier

- public static **BATCH_SIZE_DEFAULT**
- public String **batchSizeTipText**()
- public double **classifyInstance**(weka.core.Instance **arg0**) throws java.lang.Exception
- public String **debugTipText**()
- public double **distributionForInstance**(weka.core.Instance **arg0**) throws java.lang.Exception
- public double **distributionsForInstances**(weka.core.Instances **arg0**) throws java.lang.Exception
- public String **doNotCheckCapabilitiesTipText**()
- public static Classifier **forName**(java.lang.String **arg0**, java.lang.String[] **arg1**) throws java.lang.Exception
- public String **getBatchSize**()
- public Capabilities **getCapabilities**()
- public boolean **getDebug**()
- public boolean **getDoNotCheckCapabilities**()
- public int **getNumDecimalPlaces**()
- public String **getOptions**()
- public String **getRevision**()
- public boolean **implementsMoreEfficientBatchPrediction**()
- public Enumeration **listOptions**()
- protected **m_BatchSize**
- protected **m_Debug**
- protected **m_DoNotCheckCapabilities**
- protected **m_numDecimalPlaces**
- public static Classifier **makeCopies**(Classifier **arg0**, int **arg1**) throws java.lang.Exception
- public static Classifier **makeCopy**(Classifier **arg0**) throws java.lang.Exception
- public static **NUM_DECIMAL_PLACES_DEFAULT**
- public String **numDecimalPlacesTipText**()
- public void **postExecution**() throws java.lang.Exception
- public void **preExecution**() throws java.lang.Exception
- public void **run**(java.lang.Object **arg0**, java.lang.String[] **arg1**) throws java.lang.Exception
- public static void **runClassifier**(Classifier **arg0**, java.lang.String[] **arg1**)
- public void **setBatchSize**(java.lang.String **arg0**)
- public void **setDebug**(boolean **arg0**)
- public void **setDoNotCheckCapabilities**(boolean **arg0**)
- public void **setNumDecimalPlaces**(int **arg0**)
- public void **setOptions**(java.lang.String[] **arg0**) throws java.lang.Exception