Manual of SoMMinT: Developer's Guide

Jinjing Ma
jinjingm@cs.unc.edu

Contents

1 Introduction

SoMMinT is a Java-based package for end-to-end social media mining process from processing raw social media data to generate prediction model or correlation analysis. It integrates a set of source data and general machine-learning toolkits such as Weka[1] and Mallet[2], and researchers can switch data source and retarget the toolkits easily. Moreover, researchers can extract features in layers. Figure1 shows the whole process of using SoMMinT. Download link is https://github.com/pdewan/SocialMediaMining.

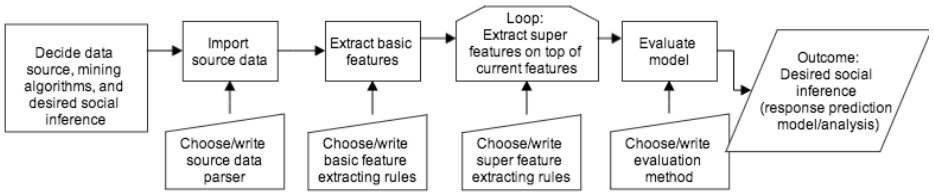

Figure 1: Process of using this high-level toolkit for response prediction/analysis

SoMMinT is designed and programmed following principles below.

1. Minimize repeated code.
2. Encapsulate what varies.
3. Use composition over inheritance if possible.
4. Programming for interface not implementation.
5. Delegation principle. This toolkit integrate mining algorithms from existing toolkit and external data parsers.
6. Open closed principle. Researchers can easily extend this toolkit to broader social media application only by implementing interfaces, inheriting classes, or replacing a module.

Guided by these principles, we applied following object-oriented design patterns.

- Composite pattern is used to construct intermediate data set.
- Adapter pattern and template-method pattern are used to uniform interfaces of multiple mining algorithms from various low-level packages.
- Strategy pattern is used to extract features from data and perform algorithms.

2 Import source data

SoMMinT transform different social media source data to the inter-stage data `ThreadDataSet` as shown in Figure 2.

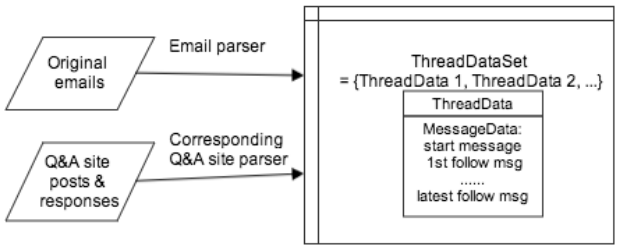

Figure 2: Import source data in uniform format

- `ThreadDataSet` is a collection of `ThreadData` representing a conversation.
- `ThreadData` contains a chronological sequence of `MessageData`.
- `MessageData` stands for a message or a post.

For email, a `ThreadData` is an email conversation consisting of email messages. While for YahooAnswers data, a `ThreadData` is a discussion of a certain question, when the question and each answer becomes a `MessageData` . Different source data for response research are parsed into the uniform data of `ThreadDataSet`.

2.1 The use of source data import module

SoMMinT has two build-in parsers: one for emails preprocessed by an external package and another for JSON-format data. Each data source needs a corresponding data config to set the data fields and types. The following code shows the process to parse crawled YahooAnswers data into `ThreadDataSet`. `JsonThreadParser` is a general parser which can parse all JSON-format data into a `ThreadDataSet` with information offered by certain `JsonDataConfig`. All classes and interfaces for importing source data are in package `dataimport`.

```
/** Define the data config for YahooAnswers question and answer */
JsonDataConfig qconfig = new YahooAnswersQuestionConfig();
JsonDataConfig aconfig = new YahooAnswersAnswerConfig();

/** Create a parser to parse YahooAnswers question and answer data
  *  from given file directory */
JsonThreadParser parser =
               new JsonThreadParser(qconfig, aconfig,YahooAnswersDataConfig.DATE_DEFAULT);
File dirfile = new File("directoryOfData");
ThreadDataSet threads = parser.parseDirectory(dirfile);
```

2.2 Extension

The future work of SoMMinT would include integrating more social media data source, which would following the rules below. If developers wish to apply the framework of SoMMinT on data source whose parser has not been included in the toolkit yet, they should follow the rules as well. The

new code for other data sources, including data configure and parser, will be added to appropriate package.

- Data configure. Each kind of message in the source data should have corresponding data configure class to define the attribute names and types. The data configure class must extend `MsgDataConfig` or its appropriate inheritance.
- Date-type attribute. The messages must have an attribute named by `MsgDataConfig.DATE_DEFAULT` and assigned value with format of `MsgDataConfig.DATEFORMAT_DEFAULT`. This is crucial to make `ThreadData` keep `MessageData` in chronological order.
- Data parser. The developers should implement a data parser for a new data source. If the messages are not organized in threads, they should assign related thread id to each message then let `ThreadRetriever` to sort the message in to `ThreadDataSet`.

3 Towards intermediate data format for mining tasks: extract basic features

3.1 The use of basic feature extraction module

A `ThreadDataSet` will be converted to a `IntermediateDataSet`. `IntermediateDataSet` is an interface of wrapped data for certain machine-learning toolkit format. For example, `IntermediateDataSet` for Weka contains data in Weka format. An `IntermediateDataSet` and be save to or load from a file. Figure **??**.(b) shows the transformed translation process between source data and low-level toolkits.

Figure 3 shows the detailed `ThreadDataSet`-to-`IntermediateDataSet` conversion process. The process is based on a `BasicFeatureExtractor` and a set of `BasicFeatureRule`. A `BasicFeatureRule` is a function taking a `ThreadData` as input, then produces the desired "basic feature" (a feature that can be extracted directly from a conversation, such as the number of responses). The `BasicFeatureExtractor` organizes a set of `BasicFeatureRules`, using them to convert a `ThreadDataSet` to a specified `IntermediateDataSet` saving all features produced by an array of `BasicFeatureRules`.

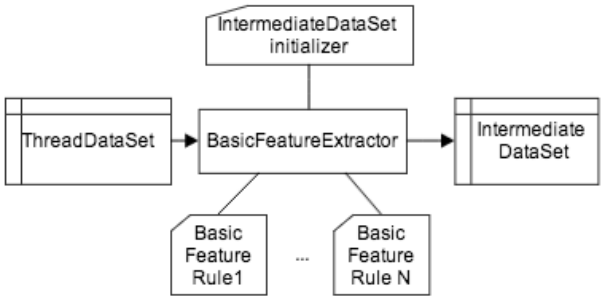

Figure 3: Framework of Basic feature extraction

User can use basic feature extracting module to convert `ThreadDataSet` to `IntermediateDataSet` as follows.

```
/** Create intermediate data initializer to define particular
 * intermediate data initializing in basic feature extracting
  * Here we use Weka intermediate dataset
  */
WekaDataSetInitializer initializer = new WekaDataSetInitializer();
BasicFeatureExtractor basicExtractor = new BasicFeatureExtractor(initializer);

/** Create rules to extract basic features of questions.
 *  These features are human judger related, so rules are defined to read files
 */
IBasicFeatureRule[] basicRules = new IBasicFeatureRule[4];
basicRules[0] = new EmailSenderIdRule("senderId", addressNum);
basicRules[1] = new EmailRecipientIdsRule("recipient", addressNum);
basicRules[2] = new EmailSubjectLengthRule("subjectLength");
basicRules[3] = new EmailRecipientNumRule("recipientNum");

/** Extract basic features, store in an IntermediateDataSet */
IntermediateDataSet featureSet = basicExtractor.extract(anEmailThreadDataSet, "feature", basicRules);
```

Table 2 list the package, interfaces, and classes involved in converting `ThreadDataSet` to `IntermediateDataSet`.

3.2 Extension

If developers would use basic features not included in the toolkit yet, they should follow the rules to implement corresponding basic feature extracting rules.

- All basic feature rules must inherit appropriate rules in `rule` according to the data-type of the feature they are extracting.
- Do not write features that simply copy attributes from message data, call corresponding rules in `rule.basicfeature.copyraw` directly.

4 Mining tasks: extract super features and produce final model

4.1 The use of super feature extraction module

The super feature extraction module aims at integrating multiple low-level machine-learning toolkits for mining tasks. Super feature in SoMMinT is defined as features derived from `IntermediateDataSet`, including those produced by mining algorithms. Since mining algorithms can serve as both extracting derived features and the model for prediction or correlation analysis, this module use layered feature extracting rules: the top-layer rule works as prediction/correlation model, while others simply extract super features. For example, in StackOverflow, each question is assigned with several tags indicating the topic. However, the number of tags is enormous, and

the semantics of tags often overlap. Thus, semantic tag groups extracted from original tags would be a better feature to use. Figure 4 shows the functioning structure of the super feature extracting module.

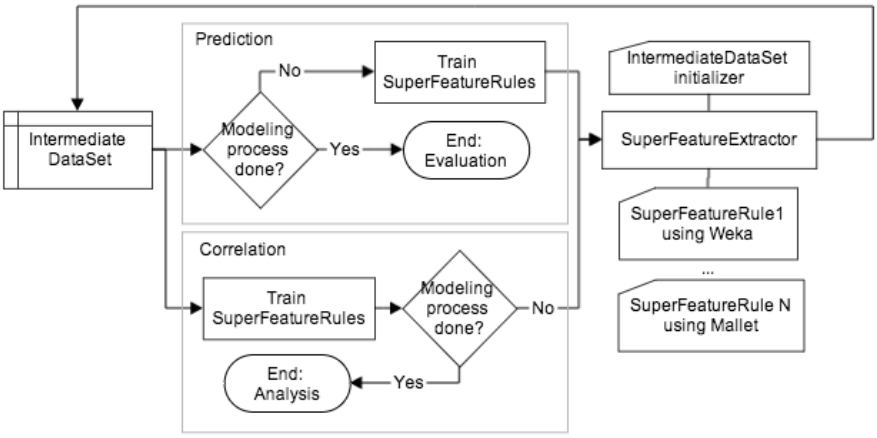

Figure 4: Email response time prediction process

User can use super feature extracting module to extract super feature from `IntermediateDataSet` as follows.

```
/** Extract basic measurements, stored in an IntermediateDataSet */
IntermediateDataSet measurementSet = basicExtractor.extract(threads, "answerMeasure", basicRulesA);

/** Create a super feature extractor to extract features from intermediate data.
 *  Use the same Weka intermediate data initializer in super feature extracting
  */
SuperFeatureExtractor superExtractor = new SuperFeatureExtractor(initializer);

/** Create rules to extract a combined measurement of answer quality.
 */
ISuperFeatureRule[] superRulesAnswer = new ISuperFeatureRule[1];
double[] weight = {0.2, 0.2, 0.3, 0.3};
superRulesAnswer[0] = new WeightedSumRule("answerQuality", weight);
IntermediateDataSet combinedMeasurement =
                  superExtractor.extract(measurementSet, "answerMeasure", superRulesAnswer);
```

User can use super feature extracting module to produce prediction/correlation model as follows, by using only the rule wrapping target model.

```
/** split data set into [trainset, testset] with 80% instances as trainset */
IntermediateDataSet[] traintest = predictionSet.splitToTrainAndTest(0.8);

/** Create rule of logistic regression and train.
 */
```

```
ISuperFeatureRule[] logi = new ISuperFeatureRule[1];
ArrayList<String> domain = new ArrayList<String>(2);
domain.add("y"); domain.add("n");
logi[0] = new WekaLogisticRegressionModelRule("hasResponse", domain);
((IWekaModelRule)logi[0]).train(traintest[0], null);

/** Get prediction result for test sets.
 */
SuperFeatureExtractor superExtractor = new SuperFeatureExtractor(initializer);
IntermediateDataSet finDataSet = superExtractor.extract(traintest[1], "test", logi);
```

Table 3 list the package, interfaces, and classes involved in converting `ThreadDataSet` to `IntermediateDataSet`.

4.2 Extension

The future work of SoMMinT would include integrating more general data mining toolkits, which would following the rules below. If developers wish to apply the framework of SoMMinT on data source whose parser has not been included in the toolkit yet, they should follow the rules as well. The new code for other data mining toolkits should be added to appropriate package.

- All super feature rules must inherit appropriate rules in `rule.superfeature` according to the data-type of the feature they are extracting.
- All model rules must implement interface `IModelRule`.
- Do not write features that simply copy features from intermediate data, call corresponding rules in `rule.superfeature.copy` directly.

5 Model evaluation

Developers can evaluate prediction model or analysis correlation by accessing the `evaluation(...)` method of model rules as follows, to use evaluation offered by machine-learning toolkits.

```
/** Evaluate the models with Weka built-in evaluation
 */
((IWekaModelRule)logi[0]).evaluate((WekaDataSet)trainset, (WekaDataSet)testset);
```

Otherwise, developers can use the prediction or correlation produced by final model on test set as follows, and write their own evaluation method.

```
/** Get prediction for test sets.
 */
SuperFeatureExtractor superExtractor = new SuperFeatureExtractor(initializer);
IntermediateDataSet predictionSet = superExtractor.extract(testset, "testtest", logi);
```

A List of packages, interfaces, and classes

Package	Class/Interface	Role
dataimport		- Contains code for common needs to import social media data to an integrated inter-stage data
	MessageData	- Stands for a message or a post.
	MsgDataConfig	- Define fields name/type of a message data.
	ThreadData	- Stands for a conversation. It contains a chronological sequence of message data.
	ThreadDataSet	- An thread data stands for a conversation. It contains a chronological sequence of message data.
	ThreadRetriever	- Sort messages into conversations. It's usually called in a source data parser.
-email		- Contains code to import email data
	...	
-json		- Contains code to import JSON-format data
	...	
-json.yahooanswers		- Contains code to import JSON-format YahooAnswers data
	...	

Table 1: Packages, interfaces, and classes for importing data.

Package	Class/Interface	Role
dataconvert		- Contains code for intermediate data set
	IntermediateData	- Intermediate data interface. All schemes of intermediate data in SoMMinT should implement this interface.
	IntermediateDataSet	- Intermediate data set interface. All schemes of intermediate dataset in * SoMMinT implement this interface.
	IntermediateDataSet-Initializer	- Intermediate data set initializer interface. Each implementation of intermediate dataset in SoMMinT should implement one initializer for feature extraction. *
	WekaData	- An intermediate data implemented in Weka format
	WekaDataSet	- An intermediate data set implemented in Weka format
	WekaDataInitializer	- Implementation of intermediate data set initializer for Weka intermediate dataset
	IFeatureExtractor	- Feature extractor interface. All schemes for feature extraction in SoMMinT implement this interface. Note that a feature extractor for particular type of dataset should define its own extracting method.
	BasicFeatureExtractor	- Feature extractor which extract basic features from ThreadDataSet to IntermediateDataSet
	...	

Package	Class/Interface	Role
rule		- Defines types of feature extracting rules
	IFeatureRule	- Basic interface of all feature extracting rules
	FeatureRule	- Abstract, superclass of all feature extracting rules
	DateFeatureRule	- Abstract, Superclass of all rules extracting date-type features
	NominalFeatureRule	- Abstract, superclass of all rules extracting nominal features
	NumericFeatureRule	- Abstract, superclass of all rules extracting numeric features
	NumericVector-FeatureRule	- Abstract, superclass of all rules extracting an array of numeric features
	StringFeatureRule	- Abstract, superclass of all rules extracting string features
	BinaryFeatureRule	- Subclass of NominalFeatureRule to extract binary features
rule.basicfeature		- Contains implementation of basic feature extracting rules
	...	
rule.basicfeature .copyraw		- Contains basic feature extracting rules which copies attribute of certain message of the thread directly
	...	

Table 2: Packages, interfaces, and classes for basic feature extraction

Package	Class/Interface	Role
dataconvert		- Contains code for intermediate data set
	IFeatureExtractor	- Feature extractor interface. All schemes for feature extraction in SoMMinT implement this interface. Note that a feature extractor for particular type of dataset should define its own extracting method.
	SuperFeatureExtractor	- Feature extractor which extract basic features from IntermediateDataSet
	...	
rule.superfeature		- Defines types of super feature extracting rules, and contains implementation of non-mining super feature rules
	ISuperFeatureRule	- Basic interface of all super feature extracting rules
	DateSuperFeatureRule	- Abstract, Superclass of all super feature rules extracting date-type features
	NominalSuperFeatureRule	- Abstract, superclass of all super feature rules extracting nominal features
	NumericSuperFeatureRule	- Abstract, superclass of all super feature rules extracting numeric features

Package	Class/Interface	Role
	NumericVectorSuperFeatureRule	- Abstract, superclass of all super feature rules extracting an array of numeric features
	StringFeatureRule	- Abstract, superclass of all super feature rules extracting string features
	...	
rule.superfeature.copyraw		- Contains super feature extracting rules which copies feature directly from IntermediateData
	...	
rule.superfeature.model		- Contains superclasses of rules wrapping mining algorithms
	IModelRule	- Basic interface of all super feature extracting rules wrapping mining algorithms
	NominalModelRule	- Abstract, superclass of all model rules extracting nominal features
	NumericModelRule	- Abstract, superclass of all model rules extracting numeric features
	NumericVectorModelRule	- Abstract, superclass of all model rules extracting an array of numeric features
	...	
rule.superfeature.model.mallet		- Contains superclasses of rules wrapping mining algorithms from Mallet
	IMalletModelRule	- Basic interface of all model rules wrapping mining algorithms from Mallet
	MalletTopicModelRule	- Abstract, superclass of all Mallet model rules for topic modeling
	MalletParallelLDAModelRule	- Wrap parallel LDA model from Mallet
	...	
rule.superfeature.model.Weka		- Contains superclasses of rules wrapping mining algorithms from Weka
	IWekatModelRule	- Basic interface of all model rules wrapping mining algorithms from Weka
	WekaRegressionModelRule	- Abstract, superclass of all Weka regression model rules
	WekaClussifyModelRule	- Abstract, superclass of all Weka classification model rules
	WekaClusterModelRule	- Abstract, superclass of all Weka cluster model rules
	WekaDecisionTreeModelRule	- An implementation of Weka classification model rule of decision tree
	WekaKmeansModelRule	- An implementation of Weka cluster model rule of K-means

Package	Class/Interface	Role
	WekaLinearRegressio-nModelRule WekaLogisticRegressi-onModelRule ...	- An implementation of Weka regression model rule of linear regression - An implementation of Weka regression model rule of logistic regression

Table 3: Packages, interfaces, and classes for super feature extraction

B User Cases

B.1 Prediction

We use an email response time prediction case derived from the ongoing work of the UNC-CH group; some researchers in our group are exploring prediction model for email response time. Temporally we are focusing on predicting response existence and the first response time based on the initiating message. The work can be found at https://bitbucket.org/jbartel/recipientprediction/. Figure 5 shows the current research process.

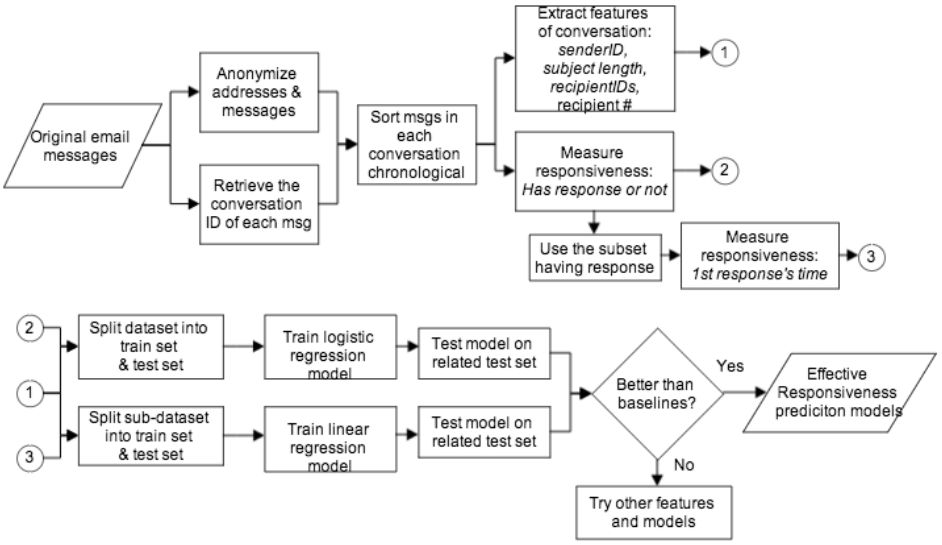

Figure 5: Email response time prediction process

The source data is generated by users' communication through email. After anonymizing messages/users and reconstructing conversations, conversation features and response time measurements are extracted as input of mining algorithms. Then, Logistic regression model for predicting response existence and linear regression for first response time are trained separately. If the trained

models have satisfactory performance on test set, we then have the expected outcome of effective email response prediction models. Otherwise, we would iteratively try alternative features, measurements, and models.

The code of this case is as follows, which can be found in `usercase` package.

```
package usercase;

import java.util.ArrayList;

import dataconvert.IntermediateDataSet;
import dataimport.email.EmailThreadParser;
import rule.basicfeature.ContainsFollowMessageRule;
import rule.basicfeature.EmailRecipientIdsRule;
import rule.basicfeature.EmailRecipientNumRule;
import rule.basicfeature.EmailSenderIdRule;
import rule.basicfeature.EmailSubjectLengthRule;
import rule.basicfeature.FirstResponseTimeRule;
import rule.basicfeature.IBasicFeatureRule;
import rule.filterrule.HasResponseFilterRule;
import rule.superfeature.ISuperFeatureRule;
import rule.superfeature.model.weka.IWekaModelRule;
import rule.superfeature.model.weka.WekaLinearRegressionModelRule;
import rule.superfeature.model.weka.WekaLogisticRegressionModelRule;
import dataconvert.BasicFeatureExtractor;
import dataconvert.SuperFeatureExtractor;
import dataconvert.ThreadDataFilter;
import dataconvert.WekaDataSet;
import dataconvert.WekaDataSetInitializer;
import dataimport.ThreadDataSet;

public class EmailResponseTimePrediction {

public static void main(String[] args) throws Exception {

/** Create an email parser to parse files into thread data set */
EmailThreadParser emailParser = new EmailThreadParser();
ThreadDataSet dataset1 = emailParser.parse("subjects.txt", "attachments.txt", "messages.txt");

/** Select & copy those with response into another thread data set */
HasResponseFilterRule filterRule = new HasResponseFilterRule(null);
ThreadDataFilter filter = new ThreadDataFilter();
ThreadDataSet dataset2 = filter.filt(dataset1, filterRule);

/** predefine the email address (anonymized) number */
int addressNum = 170;

/** Create rules to extract basic features of questions.
 *  These features are human judger related, so rules are defined to read files
 */
IBasicFeatureRule[] basicRules = new IBasicFeatureRule[4];
```

```
basicRules[0] = new EmailSenderIdRule("senderId", addressNum);
basicRules[1] = new EmailRecipientIdsRule("recipient", addressNum);
basicRules[2] = new EmailSubjectLengthRule("subjectLength");
basicRules[3] = new EmailRecipientNumRule("recipientNum");

/** Create rules to extract basic measurements of answer quality.
 */
IBasicFeatureRule[] basicRules1 = new IBasicFeatureRule[1];
basicRules1[0] = new ContainsFollowMessageRule("hasResponse");

IBasicFeatureRule[] basicRules2 = new IBasicFeatureRule[1];
basicRules2[0] = new FirstResponseTimeRule("responseTime");

/** Create intermediate data initializer to define particular
  * intermediate data initializing in basic feature extracting
  * Here we use Weka intermediate dataset
  */
WekaDataSetInitializer initializer = new WekaDataSetInitializer();
BasicFeatureExtractor basicExtractor = new BasicFeatureExtractor(initializer);

/** Extract basic features & measurements , stored in an IntermediateDataSet */
IntermediateDataSet featureSet1 = basicExtractor.extract(dataset1, "feature", basicRules);
IntermediateDataSet measure1 = basicExtractor.extract(dataset1, "measure1", basicRules1);
IntermediateDataSet featureSet2 = basicExtractor.extract(dataset2, "feature", basicRules);
IntermediateDataSet measure2 = basicExtractor.extract(dataset2, "measure2", basicRules2);

/** Merge features and measurement "hasResponse" in the same intermediate data set
 * to fit logistic regression model
 */
IntermediateDataSet predictionSet1 = featureSet1.mergeByAttributes(measure1);
/** Set attribute index of dependent variable */
predictionSet1.setTargetIndex();

/** Merge features and measurement "responseTime" in the same intermediate data set
 * to fit linear regression model
 */
IntermediateDataSet predictionSet2 = featureSet2.mergeByAttributes(measure2);
/** Set attribute index of dependent variable */
predictionSet2.setTargetIndex();

/** save a prediction data set */
predictionSet2.save("temp.arff");

/** split data set into [trainset, testset] with 80% instances as trainset */
IntermediateDataSet[] traintest1 = predictionSet1.splitToTrainAndTest(0.8);
IntermediateDataSet[] traintest2 = predictionSet2.splitToTrainAndTest(0.8);

/** Create rule of logistic regression and train.
 */
ISuperFeatureRule[] logi = new ISuperFeatureRule[1];
```

```
ArrayList<String> domain = new ArrayList<String>(2);
domain.add("y"); domain.add("n");
logi[0] = new WekaLogisticRegressionModelRule("hasResponse", domain);
((IWekaModelRule)logi[0]).train(traintest1[0], null);

/** Create rule of linear regression and train.
 */
ISuperFeatureRule[] linear = new ISuperFeatureRule[1];
linear[0] = new WekaLinearRegressionModelRule("responseTime");
((IWekaModelRule)linear[0]).train(traintest2[0], null);

/** Get prediction for test sets.
 */
SuperFeatureExtractor superExtractor = new SuperFeatureExtractor(initializer);
IntermediateDataSet finDataSet1 = superExtractor.extract(traintest1[1], "test", logi);
IntermediateDataSet finDataSet2 = superExtractor.extract(traintest2[1], "test", linear);

/** Evaluate the models with Weka built-in evaluation
 */
((IWekaModelRule)logi[0]).evaluate((WekaDataSet)traintest1[0], (WekaDataSet)traintest1[1]);
((IWekaModelRule)linear[0]).evaluate((WekaDataSet)traintest2[0], (WekaDataSet)traintest2[1]);

}

}
```

B.2 Correlation

We use a YahooAnswers answer quality correlation case derived from F.M. Harper and his colleagues' work of analyzing factors affecting answer quality across common Q&A sites, including Google Answers, Library Reference, AllExperts, YahooAnswers, and Live QnA [3]. Here we only use the part of YahooAnswers for case study with process, which is illustrated in Figure 6.

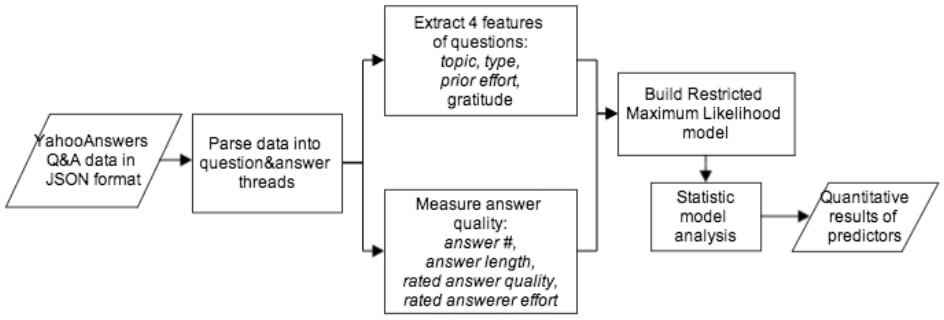

Figure 6: Process of YahooAnswers answer quality factoring

In this case, user online Q&A interactions generate the source data. Then it is parsed into questioning and answering threads (conversations), extracted from which are question features and answer quality measurements. Then, a restricted maximum likelihood regression model is built to analyze the correlation between features and measurements to find outstanding factors of answer quality.

The code of this case is as follows, which can be found int the `usercase` package

```
package usercase;

import java.io.File;

import rule.basicfeature.IBasicFeatureRule;
import rule.basicfeature.ReadFromFileNumericRule;
import rule.basicfeature.ReadFromFileNumericVectorRule;
import rule.basicfeature.ResponseAvgCharLength;
import rule.basicfeature.ResponseNumRule;
import rule.superfeature.ISuperFeatureRule;
import rule.superfeature.WeightedSumRule;
import rule.superfeature.model.weka.IWekaModelRule;
import rule.superfeature.model.weka.WekaLinearRegressionModelRule;
import dataconvert.BasicFeatureExtractor;
import dataconvert.IntermediateDataSet;
import dataconvert.SuperFeatureExtractor;
import dataconvert.WekaDataSet;
import dataconvert.WekaDataSetInitializer;
import dataimport.ThreadDataSet;
import dataimport.json.JsonDataConfig;
import dataimport.json.JsonThreadParser;
import dataimport.json.yahooanswers.YahooAnswersAnswerConfig;
import dataimport.json.yahooanswers.YahooAnswersDataConfig;
import dataimport.json.yahooanswers.YahooAnswersQuestionConfig;

public class YahooAnswersQualityCorrelation {

      public static void main(String[] args) throws Exception {

            /** Define the data config for YahooAnswers question and answer */
            JsonDataConfig qconfig = new YahooAnswersQuestionConfig();
            JsonDataConfig aconfig = new YahooAnswersAnswerConfig();

            /** Create a parser to parse YahooAnswers question and answer data
              * from given file directory
              */
            JsonThreadParser parser = new JsonThreadParser(qconfig, aconfig,
                                                YahooAnswersDataConfig.DATE_DEFAULT);
            File dirfile = new File("data/YahooAnswers/rawdata");
            ThreadDataSet threads = parser.parseDirectory(dirfile);

            /** Create rules to extract basic features of questions.
             *  These features are human judger related, so rules are defined to read files
```

```
 */
IBasicFeatureRule[] basicRulesQ = new IBasicFeatureRule[4];
basicRulesQ[0] = new ReadFromFileNumericVectorRule("questionType", "typeFile");
basicRulesQ[1] = new ReadFromFileNumericVectorRule("gratitude", "gratitudeFile");
basicRulesQ[2] = new ReadFromFileNumericVectorRule("priorEffort", "priorEffortFile");
basicRulesQ[3] = new ReadFromFileNumericVectorRule("topic", "topicFile");

/** Create intermediate data initializer to define particular
  * intermediate data initializing in basic feature extracting
  * Here we use Weka intermediate dataset
  */
WekaDataSetInitializer initializer = new WekaDataSetInitializer();
BasicFeatureExtractor basicExtractor = new BasicFeatureExtractor(initializer);

/** Extract basic features, stored in an IntermediateDataSet */
IntermediateDataSet featureSet =
               basicExtractor.extract(threads, "questionFeature", basicRulesQ);

/** Create rules to extract basic measurements of answer quality.
 */
IBasicFeatureRule[] basicRulesA = new IBasicFeatureRule[4];
basicRulesA[0] = new ResponseNumRule("answerNum");
basicRulesA[1] = new ResponseAvgCharLength("answerAvgL", YahooAnswersDataConfig.CONTENT);
basicRulesA[2] = new ReadFromFileNumericRule("ratedAnswerQuality", "ratedAnswerQualityFile");
basicRulesA[3] = new ReadFromFileNumericRule("ratedAnswerEffort", "ratedAnswerEffortFile");

/** Extract basic measurements, stored in an IntermediateDataSet */
IntermediateDataSet measurementSet =
              basicExtractor.extract(threads, "answerMeasure", basicRulesA);

/** Create a super feature extractor to extract features from intermediate data.
 *  Use the same Weka intermediate data initializer in super feature extracting
  */
SuperFeatureExtractor superExtractor = new SuperFeatureExtractor(initializer);

/** Create rules to extract a combined measurement of answer quality.
 */
ISuperFeatureRule[] superRulesAnswer = new ISuperFeatureRule[1];
double[] weight = {0.2, 0.2, 0.3, 0.3};
superRulesAnswer[0] = new WeightedSumRule("answerQuality", weight);
IntermediateDataSet combinedMeasurement =
              superExtractor.extract(measurementSet, "answerMeasure", superRulesAnswer);

/** Merge features and measurement in the same intermediate data set
 */
IntermediateDataSet dataset = featureSet.mergeByAttributes(combinedMeasurement);
dataset.setTargetIndex();

/** Create the rule containing regression model to fit the correlation
 */
```

```
            ISuperFeatureRule[] finalRule = new ISuperFeatureRule[1];
            finalRule[0] = new WekaLinearRegressionModelRule(null);
            ((IWekaModelRule)finalRule[0]).train(dataset, null);

            /** Use the model's build-in evaluation to do the analysis
             */
            ((IWekaModelRule)finalRule[0]).evaluate((WekaDataSet)dataset, (WekaDataSet)dataset);

}

}
```

References

[1] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, November 2009.

[2] Andrew Kachites McCallum. Mallet: A machine learning for language toolkit. http://mallet.cs.umass.edu, 2002.

[3] F. Maxwell Harper, Daphne Raban, Sheizaf Rafaeli, and Joseph A. Konstan. Predictors of answer quality in online q&a sites. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, pages 865–874, New York, NY, USA, 2008. ACM.