Introduction

Purpose of Document

Predictive analytics software helps to find non-obvious, hidden patterns in large data sets. With the rapid growth of text information, text classification has become one of the key techniques for organizing text data. Compared to the state-of-art method, Support Vector Machines (SVM) classifiers with Bag-of-Words (BoW) representation of text data show substantial classification performance with respect to accuracy and generalization. The main challenge is extracting features and selecting the SVM parameters to get the best possible performance. We propose a generic framework for building text classifiers using IBM SPSS Modeler and Java, with no need to domain specific dictionaries.

This article explores building SVM-based classification framework for text classification. We show the complete design and implementation details for this framework in IBM SPSS Modeler and Java. In addition, we show a case study of applying this framework for the classification of sample software defect data for smart software engineering. We illustrate experimentation steps along with obtained results.

Applicability

The project described in this document was developed with IBM SPSS Modeler 16.

Exclusions and Exceptions

This document will not go into details for each item discussed or provide step-by-step instructions. It is intended to provide a high level overview of the project, and highlight some of the analysis techniques that were used.
Assumptions

This document assumes readers have experience with IBM SPSS Modeler along with Java programming experience.

Overview

Predictive analytics software is used in many applications to discover unknown, hidden patterns in data. A typical example from the area of business analytics is the task of text classification. The availability of large amount of text data showed the need to categorize them for better decision making.

An important application of text classification in software engineering is Orthogonal Defect Classification (ODC) [1], which is a proven method for performing software defect analysis and fixing. However, ODC requires text description of defects to be able to classify them correctly. One approach is to classify defects manually, but of course this approach can't scale with large amount of defect data, hence the need for a classification engine to automatically classify next defect descriptions into predefined category to take proper actions.

There are many state-of-art methods for text classification - decision trees, rule based and Support Vector Machines (SVM) [2]. SVM has been proven to give better performance than other methods. In this article we describe framework for building a classifier to classify text defect data and how it can be generalized for different types of text data [3].

We divide the description of our work into two main articles - the main article (this one) and a supplementary article that is contained in the attachment for this article. The main article is organized as follows:

• The next section, titled Problem definition and solution approach, will describe the problem at hand.
• The section titled Framework architecture and development will present the framework architecture, the design objectives, an overview for the methods and formulae used as well as the the process and details for developing this framework using SPSS Modeler and Java.
• The section titled Experimentation setup and parameters selection presents the framework parameters, the experimentation setup and the results.

The supplementary article contains the details for the algorithms and formulae used in the framework as well as descriptions for the additional supplementary material.

Problem definition and solution approach

In this section we present a formal definition of the problem at hand and the general approach for solving it. Given a set of training records \( x_i \), \( i \) ranges from 1 to \( l \) (the number of training records). Each record \( x_i \) is a vector of features of size \( k \). A label \( y_i \) is associated with each record \( x_i \), where \( y_i \) ranges from 1 to \( m \) (the number of predefined classes). The classification can be thought of as a mapping function \( y_i = f(x_i) \) that maps the input records to one of the set of predefined classes. The
task of building the classifier is to estimate a function (classifier) \( f^* \) that maximizes the average classification accuracy. The classification accuracy for each class is the ratio between the number of correctly classified records \( (f^*(x_i)= f(x_i)) \) to the total number of records in that class. The average accuracy is calculated over all predefined classes.

For text classification, we have to convert each input text record to a features vector to be able to build the classifier. This will be illustrated in detail in the next section under the topic Data pre-processing.

**Framework architecture and development**

In data mining, a standard process called CRISP-DM (Cross Industry Standard Process for Data Mining) [13] is used to tackle data mining and predictive analytics problems. In this section we present the architecture of the SVM text classifier and how it is developed based on CRISP-DM process using Java and IBM SPSS Modeler.

**Architecture and Design**

In Figure 1, we show the architecture of the framework based on three main components:

1. **Data Pre-processing**: This component is responsible for pre-processing the input text data to generate features vectors based on the Bag-of-Words model [3].
2. **Model Building and Evaluation**: This component is responsible for building the classification model based on the generated features vectors. The input features vectors are divided into training, test and validation partitions. The training partition is used for initial estimation for the SVM classification model. The testing partition is used to fine tune the initial parameter estimations. The validation partition is used to measure the model performance.
3. **Data Repository**: This component is responsible for storing and managing the data use within the framework. There are two main types of data - the input data and the generated features vectors. There is also auxiliary data that is used in the data pre-processing component.

**Figure 1. Architecture for SVM-based text classification framework**
The design objectives of the proposed framework are twofold:

1. **Generality**: In feature generation and selection, we don't depend on any domain specific dictionaries for extracting important terms from the text data for feature generation. Instead we consider each term in a document as feature that is used on classification, which will be shown later. Also, in the classification model we have chosen SVM as it is proven to have better generalization performance over other classification methods and alleviates the problem of over-fitting. This is important in text classification as generated features vectors tend to be sparse which increases the possibility of over-fitting.

2. **Pluggability**: We design our solution as a framework whose components can be modified and/or replaced easily. For example, in feature selection a developer can add or remove filters as needed. For building the classification model, the developer can replace SVM with any other classification technique that uses Bag-of-Words [3] model such as k-NN (K-Nearest Neighbours) and **decision trees**.

**Development of the framework based on CRISP-DM process**

As with any data mining project, CRISP-DM is the standard process for development. In this section we show the tasks for developing the framework architecture shown earlier in Figure 1 based on the CRISP-DM process. We will show the development details along with best practice recommendations to get the best results.

First we briefly discuss the major phases of the CRISP-DM process

- Business understanding
- Data understanding
- Data pre-processing
- Modeling
- Evaluation
- Deployment

In the work presented here, the data pre-processing phase is developed in Java, the modeling and evaluation phases are developed in SPSS Modeler, and the data repository is developed using simple Excel spreadsheets.

**Business understanding**

The target is to build a framework for classifying new text data given a predefined set of classes. This framework should work without the need to any domain specific dictionaries. The input is a set of labelled text records.

**Data understanding**

Below is a formal description of the input data model, descriptions of the field's roles in building the classifier, and sample data that we are expecting.

The input is a table of labelled records with the following fields:
• **ID** - A unique identifier for each record.
• **Text data** - The text data from which we extract features to build the classifier.
• **Record class** - Used as a label for each record to build the classifier.

In Table 1 we show the skeleton of the input data for any text classifier.

### Table 1. General skeleton for input data model for any text classifier

<table>
<thead>
<tr>
<th>ID</th>
<th>Text data</th>
<th>Record class</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T&lt;sub&gt;1&lt;/sub&gt;</td>
<td>y&lt;sub&gt;1&lt;/sub&gt;</td>
</tr>
<tr>
<td>2</td>
<td>T&lt;sub&gt;2&lt;/sub&gt;</td>
<td>y&lt;sub&gt;2&lt;/sub&gt;</td>
</tr>
<tr>
<td>3</td>
<td>T&lt;sub&gt;3&lt;/sub&gt;</td>
<td>y&lt;sub&gt;3&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

A snapshot of the input data for the software defect data case study is shown in Table 2.

### Table 2. Sample from software defect input data

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
<th>R_Trigger</th>
</tr>
</thead>
<tbody>
<tr>
<td>42797</td>
<td>See this in regression run can’t reproduce it. 03.06.25 JOB37915 +DFHKE0030 - Abend DC4/AKEX in Program DFHS2PP Entry Point 00000000_28F34DB8. 03.06.25 JOB37915 +DFHKE0040 - Instruction Address: 00000000_01855F9E Offset in Program FFFFFFFF ...</td>
<td>l_coverage</td>
</tr>
<tr>
<td>42803</td>
<td>It’s not possible to build the BSF stream because two modules DFHADWB0 and DFHADDRM build with RC=4. This is ...</td>
<td>l_build</td>
</tr>
<tr>
<td>42323</td>
<td>I built an ICM to parse a schema that contains optional dateTime attributes. on mapping level 2.2 I believe it provides a string representation but at level 3.0 CICS tries ...</td>
<td>l_developer_test</td>
</tr>
</tbody>
</table>

In Table 2, the **Description** field corresponds to the **Text data** field in Table 1 and the **R_Trigger** field represents the class of the software defect described in that record and corresponds to the **Record class** field in Table 1.

### Data pre-processing

As we plan to use Support Vector Machines for building the text classifier, we need to pre-process the text data to convert it into a set of features vectors based on Bag-of-Words [3] representation to build the SVM classifier. In other words, we convert each text record \( T_i \) to a vector \( x_i=(x_{i1},x_{i2},...,x_{ij},...,x_{ik}) \), where \( x_{ij} \) is an extracted term. As we will see later, each extracted term in a document corresponds to a feature that will be used to build and run the document classifier. The pre-processing phase can be divided into two main sub-phases:

1. Feature selection/reduction
2. Feature generation
Feature selection/reduction

We design the feature selection component as a term tokenizer followed by a set of filters to filter the extracted terms to the most important one in the context of classification. The design of the feature selection module is shown in Figure 2 and is followed by a description of each sub-component.

Figure 2. The feature selection module designed as a term tokenizer followed by set of filters to select the set of the most important features for classification

Term tokenizer

We use a simple whitespace tokenizer to extract all possible terms from the text data. For simplicity, our implementation will focus on single-word terms. However, after running the tokenizer on real word documents we face the problem of having large (in order of thousands) sets of features for each document. This problem is called the curse-of-dimensionality[4] in the context of classification methods. It’s called a curse because the running time of model building increases exponentially as the number of features increases (remember that each extracted term from a document corresponds to a feature). Hence, efficient methods are needed to select the most important terms without significantly losing information from the input data and subsequently affecting classification accuracy. This is why feature selection is a must for building the classifier.

Minimum document frequency filter

To make sure that a term is frequently used in the training data and to avoid large, unnecessary features vectors, we remove terms that have a document frequency below some given threshold.

Common terms filter

Any text contains common terms that are not domain specific and have low discriminatory power for classifying records into categories. We use a list of the top 5000 frequent words and/or lemmas from the 450 million word Corpus of Contemporary American English, which is the only large and balanced corpus of American English [5]. In Table 3 we show a sample of the terms list we use in our implemented filters.

Table 3. Sample list of common keywords

<table>
<thead>
<tr>
<th>Common Words</th>
</tr>
</thead>
<tbody>
<tr>
<td>The</td>
</tr>
<tr>
<td>Be</td>
</tr>
<tr>
<td>And</td>
</tr>
</tbody>
</table>
One approach to implement this filter is to directly search for these common terms in the list of extracted terms and remove them. However, this approach has two major problems.

- The input data may not be clean. For example, the word "enough" is considered as a common keyword but due to mistyping or any other noise source in the data it can appear as *enoug*, *enogh*, *enough* and so on. Hence these mistyped words will pass the filter, which is not desirable.
- The common words are stored in the source repository as lemmas and may appear into the input text data as morphological variants. For example, *produce* is a common word but can appear as *produced*, *product*, *producing* and so on. The direct removal approach will fail to remove these variants.

The intuitive solution to these two problems is to remove any word in the text that is similar to any word in the common terms list, not only exact matches. To implement this solution, first we need to define a similarity measure between two words. We use a normalized *Levenshtein similarity measure*[^6][^7] to measure the similarity between each extracted term and the terms in the common words list. The common terms filters simply works as follows - filter out any term that has a similarity score to any common terms above a given threshold. After running this filter, most of the common terms either appearing as lemmas, morphological variation or even with typos should be removed. For more details on this filter, see Section 1.1 in the supplementary document contained in the attachment for this article.

**Information gain filter**

After removing common terms, we need to extract the most useful terms for text classification. **Usefulness** means the terms with the higher discriminatory power to classify the containing documents into the right class. We use **information gain (IG)** or **entropy** of each term as the measure for that discriminatory power. The IG filter works as follows - we calculate IG for each term passed from previous filters, then we select a ratio from the terms having the highest IG values. For more details on this filter, see Section 1.2 in the supplementary document contained in the attachment for this article.

The reasons why we put the common keywords filter before information gain filter are:

- Common terms are defined as the most frequent words within most English documents, hence we know a priori that their information gain is low anyway.
- The computation complexity is *O*(l*d*k) where l is the number of records in the training data, d is the number of terms (features vectors dimension), k is the number of classes. By applying...
the common terms filter before the IG filter, we significantly reduce d which reduces the running time of the IG filter.

Other filters

Other filters are added based on the application. For example, in our experimentation on defect data, we add a filter to remove numeric tokens, the token which consists completely of digits.

Feature generation

After selecting a subset of the most useful terms in context of classification, we now have a set of features for each record (recall that each term is considered as a feature). The next step is to calculate the feature value for each feature in each document. In our case, the feature value will be the well-known Term Frequency-Inverse Document Frequency (tf-idf). As for each record we have a features vector, this aggregated data structure is sometimes called the features matrix (recall that a matrix is a group of vectors). For more details, see Section 1.2 in the supplementary document contained in the attachment for this article.

After calculating tf-idf for all terms/features in the training data, we generate a features matrix which will have the following structure:

- Each row represents a record - in our case each record is a document with a unique ID.
- Each column represents a unique feature/term.
- Each cell in the matrix has the tf-idf value for a specific term (the corresponding column) in a specific document (the corresponding column).

In Figure 3 we show a snapshot from the features matrix generated from the data of the software defect case study. This matrix is used to build the SVM model for classifying the documents. For example, in Figure 3 in the cell in the red circle, the record ID=8219 and the feature = "dfhke003" (an extracted term), and the value of that feature for that document (tf-idf) = 15.8488.

Figure 3. A snapshot from the features matrix generated from the data of the software defect case study
Modeling

After converting the data into a Bag-of-Words model, we can reformulate the problem without loss of generality to a binary classification problem as follows - given a set of records with features vectors $x^i$ we want to classify them into class $y^i = \{-1,1\}$. In SVM we want to find the hyperplane that best separates the data points in the two classes. For details on this see Section 1.4 in the supplementary document contained in the attachment for this article.

The SPSS Modeler stream for building and evaluating the SVM model for text classification is shown in Figure 4. The stream contains nodes for reading the features vectors, partitioning the input records, building and evaluating the model.

**Figure 4. SPSS Modeler stream for building SVM classifier**

The node labelled **Input Data** is for reading a CSV file that contains the features vectors table generated using the data pre-processing module. The **Filter** node is used to remove any unnecessary fields such as record ID, which is usually removed as it is not used in modeling. Furthermore, in the defect data example, after applying filtering, some tokens appear but they do not have any meaning in the domain so we remove them. Notice that in the **Data Audit Nodes** filtering reduced the number of fields from 2896 to 2881.

The **Partition** node splits data into three groups - **training**, **test** and **validation**. It is a good practice when building any classification model to partition the input data into these three groups. The **training** partition is used to build the model with given parameters and (initially) estimate its internal coefficients. The **test** partition is used to test model classification accuracy to fine-tune model training parameters, rebuild and update its internal coefficients to improve its accuracy. The
Validation partition is used to estimate the classifier accuracy for new data. The validation partition doesn't affect the model building process either directly like training data or indirectly like test data, so the accuracy measure taken based on the validation group is a good estimator for the built classifier performance for new unlabeled data.

The **Type** node specifies the role of each field. All extracted features fields are marked as **input** for training the classifier and the **Category** field is marked as **Target** to use it as the class field for each record.

The **Category_SVM** node is a SVM model building node using extracted features as input and the Category field as the target (as configured in the Type node). We use the **Expert** mode to get more control in fine-tuning classifier parameters. The main challenge will be to select the best set of parameters to get the best possible accuracy. We illustrate parameter selection and describe how to configure the values for SVM model in the section titled **Experimentation setup and parameters selection**.

**Evaluation**

The **Analysis** node in the SPSS Modeler stream is used to measure the model performance by comparing the predicated class value with original one for each partition group (training, test, and validation). It shows the average accuracy of the model when applying it each partition. The classification accuracy for each category is calculated as the ratio between the number of correctly classified records to the total number of records in that category. The average accuracy is calculated over all categories.

In the section titled **Experimentation setup and parameters selection** we show the detailed experimental setup along with accuracy results shown in the Analysis node. Figure 5 shows how the Analysis node is configured to display the average accuracy for the classifier over all categories, grouped for each partition. Note that the **Separate by partition** checkbox has been checked in order to separate the accuracy result by training, testing, and validation partitions. This will make sure we selected the best possible parameter values and will avoid the problem of overfitting. Overfitting happens when we have very high accuracy in the training and testing partitions and very low accuracy in the validation partition. This means the model has become tailored to the training and test data and is not general enough to classify any new incoming data.
Figure 5. Configuring the Analysis node to display average overall accuracy grouped by partitions

Deployment

We simply deploy our solution as process integration between the Java-based data pre-processing program and model building stream from SPSS Modeler. For more details, see Section 2 in the supplementary document for a detailed description of the supplementary files we refer to in this section. We deploy the data pre-processing components as a standalone Java program to generate the features vectors file as a CSV (Comma Separated Values) file, then this CSV file is used as input in the SVM classifier stream.

The Java data pre-processing program has the following dependencies commons-lang3-3.3.2.jar, javacsv.jar, jxl.jar. These libraries can be downloaded using the links shown in the corresponding citation in the Resources section. To run the pre-processing module, follow these steps.

1. Edit the configuration file named config.properties for the input and output file paths and the parameters for feature selection filters. The configuration file contains documentation for each variable. We will show the details of parameters selection in the next section.
2. Download the required libraries mentioned above and put them in the folder named SVM_FV_Gen_lib which is in the same directory where the features vectors generation program SVM_FV_Gen.jar resides.


4. In the output directory specified in the configuration file you will find the generated features vectors CSV files that use the following naming convention (Figure 6):

   `fv_textField_<text data column header name>_nfrac_<top IG fraction value>_commSim_<common words similarity threshold>_minDF_<min doc frequency threshold>_removeAllDigits_<flag value to remove all-digits tokens>.csv`

**Figure 6. Sample output for the features selection and features vectors generation module**

After generating a features vectors Experimentation setup and parameters selection file, we feed it into the SPSS Modeler stream for building the SVM model. The model is built using the stream we created as follows:

1. In SPSS Modeler open the stream SVM_Stream.str.
2. Select the desired features vectors file for the source Input Data node as shown in Figure 7.
Figure 7. Selecting features vectors file as input to SVM model building in SPSS Modeler

3. Run the stream using the Run button. As shown in Figure 8, the model will be generated as a nugget named Category_SVM.
Experimentation setup and parameters selection

To summarize, the framework proposed in this paper has the following parameter set. They can be divided into two main groups,

- Features generation and selection parameters
- SVM model building parameters

Setup features generation and selection parameters

These parameters are configured in the file named `config.properties`. Here is a description for each parameter:

- **inputDataFileName**: The file path for the input Excel .XLS file containing text records for building the classifier.
- **commonTermsListFilePath**: The file path for the Excel .XLS file containing the list of common keywords for the filtering phase in the feature selection module.
- **minTermDocFreqThr**: The threshold of the minimum document frequency for each term to pass the filter.
- **logFilePath**: Path to file containing logs.
- **idFieldColHeader**: The header of the column containing record IDs.
• **textFieldColHeader**: The header of the column containing text data used to build the classifier.

• **categoryFieldColHeader**: The column header for the column containing classes for classification - in our case the category of each record that represents a software defect.

• **commonKeywordSimilarityThresholdVals**: This is a comma separated list of values that controls the filter for removing common terms. As the threshold decreases, the filter becomes more conservative and removes more terms that appears to be common.

• **splitter**: The character used to separate multiple values in the configuration file. We set it to be comma.

• **fractionTopNumTermsIGvals**: A list of values for the fraction to be selected terms for the highest information gains.

• **commonKeywordSimilarityThresholdVals**: A list of values for the threshold parameter for the common terms filter.

• **minTermDocFreqThr**: Parameter for the document frequency filter.

• **removeAllDigitsTerms**: Flag to detect whether or not to remove terms that consists of only digits.

• **outFeaturesVectorsDir**: Output directory for generated features vectors.

**Setup SVM model building parameters**

For building the SVM model using the generated features vectors, we configure a set of parameters in **Expert** mode as shown in Figure 9 and described below.

**Figure 9. Parameters for building SVM model using SPSS node**

![Figure 9](image)

The important parameters for building SVM model are:

• **Stopping criteria**: This is the difference between the estimated SVM model internal coefficients after each iteration of optimization in SVM using the Sequential Minimal
Optimization (SMO) algorithm. Increasing this parameter will increase accuracy but will also increase model building time.

- **Regularization parameter (C):** This parameter gives weight to the sum of errors in SVM in the objective function for estimating the optimal hyperplane in SVM (see the supplementary document for more information). It provides a tradeoff between training set (in-sample) error and margin. For example, as C increases the SVM tends to make the hyperplane take care of training points close to classification boundaries and the separation margin become smaller which can lead to misclassification of new points near boundaries (this is overfitting). On the other hand, as C decreases the SVM becomes more conservative as the margin increases and less weight is given to training points classification error. In other words, increasing C increases accuracy for training data but may lead to overfitting.

- **Kernel type:** The kernel function is used to non-linearly map the given feature vector to higher dimension space where data becomes "more" linearly separable with increasing computational cost. Typical types of kernel functions are polynomial of degree d Radial-Basis-Function of radius gamma. For more information on usage of kernels with SVM see [14] in the Related topics section.

### Parameters selection

For parameters selection, we use a simple semi-grid search approach to find the best set of parameters. As mentioned before, the evaluation criteria to guide the grid search is the overall average accuracy over the testing data partition - that is the model building method estimates the initial model internal coefficients using training partition, evaluate and fine-tune model parameters based on the test partition. This approach guides us to the best possible accuracy and avoids the problem of overfitting.

We assume that the set of parameters (C, kernel type and stopping criteria) are not correlated hence the parameters search method we use is as follows:

1. Initial setting for Parameters (the default in SPSS Modeler):
   - C=10
   - Kernel function = Radial
   - Gamma = 1
   - Stopping criteria = $10^{-3}$
2. Try C = 0.1 to 1 with a step of 0.1 and from 10 to 100 with a step of 10 until you find C with best possible accuracy in the training data partition.
3. For the value of C found in Step 2, try kernel types Radial, Linear and Polynomial until you find the one that gets best accuracy for the training data set.
4. For the kernel found in Step 3, try the following parameters ranges until you get the best possible accuracy:
   - If kernel = Radial, try Gamma = 0.1 to 5 with an additive step of 0.1.
   - If kernel = Polynomial, try Degree = 1 to 10, with an additive step of 1.
5. For the stopping criteria try from $10^{-3}$ to $10^{-6}$, with multiplicative step of $10^{-1}$.

After finishing Steps 2 through 5 we get the set of parameters to use for building the SVM. We found the set of parameters shown in Table 4 to give the best possible accuracy results.
Table 4. The optimal parameter set configuration for getting the best overall average accuracy

<table>
<thead>
<tr>
<th>Phase</th>
<th>Parameter Name</th>
<th>Parameter Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature Selection</td>
<td>minTermDocFreqThr</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>commonKeywordSimilarityThresholdVals</td>
<td>0.7</td>
</tr>
<tr>
<td></td>
<td>fractionTopNumTermsIGvals</td>
<td>0.8</td>
</tr>
<tr>
<td></td>
<td>removeAllDigits</td>
<td>True</td>
</tr>
<tr>
<td>Model Building</td>
<td>C</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>Kernel Type</td>
<td>Polynomial</td>
</tr>
<tr>
<td></td>
<td>Kernel Parameter</td>
<td>Polynomial Degree=6</td>
</tr>
<tr>
<td></td>
<td>Stopping criteria</td>
<td>$10^{-3}$</td>
</tr>
</tbody>
</table>

The original input training data has 4783 records and one text data field (Description). The generated feature vector contains the same number of records and 2896 fields. The running time for building the SVM model was 2 minutes and 10 seconds on an Intel 2.7 GHz dual core with 1 GB RAM reserved for the IBM SPSS Modeler Version 16 server process. The accuracy results obtained from the Analysis node are shown in Table 5 by comparing the original with estimated category for each record.

Table 5. Overall accuracy for the proposed framework on the software defect data case study obtained from the Analysis node

<table>
<thead>
<tr>
<th>Partition</th>
<th>1_Training</th>
<th>2_Testing</th>
<th>3_Validation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct</td>
<td>2247</td>
<td>516</td>
<td>306</td>
</tr>
<tr>
<td>Wrong</td>
<td>1083</td>
<td>462</td>
<td>169</td>
</tr>
<tr>
<td>Total</td>
<td>3330</td>
<td>978</td>
<td>475</td>
</tr>
</tbody>
</table>
### Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Supplementary Material</td>
<td>965KB</td>
</tr>
</tbody>
</table>
Related topics

- [1] Orthogonal Defect Classification
- [8] (2014, April) Download Apache Commons Lang
- [10] (2013, Mar.) Java Excel API - A Java API to read, write and modify Excel spreadsheets
- Dustin Boswell [14] Introduction to Support Vector Machines August 6, 2002

© Copyright IBM Corporation 2015
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)