Weka[35] StringToWordVector 源代码分析

作者：Koala++/屈伟

最近使用 wvtool 去算 tf-idf，但它要求输入是文件，而我的数据都是很短的几句话，然而个数很多，我试着产生 300 万个文件，产生个字典十几个小时都完成不了，并且给我的硬盘还很小，才 100G，一下就用完了，删除也要花无数个小时才能把这些小文件删除，所以我想如果可以以行为单位，而不是以文件为单位，可以自己定义行的解析函数，这样速度会提高很多，因为没有那么多的 I/O 操作了。本想写出来一个的，但是我的计算机很慢（让我用，我很难会感觉哪个计算机快），数据一加载就不动了，耐性有限，我也没心情做了。

下面的代码是从黄少力他们那里要来的，我当时只知道有这么回事，到底怎么用的，也懒得去 google 了，就直接拿来用了。如果只用 weka，这也可以。我自己用王义以前的代码写了一个 wvtool 产生 VSM 模型，最后产生 libsvm 数据集，还可以进一步生成.arff 的代码（网上的转换不能将产生真实的属性名），这代码我好像写过 3 次，一次用 c++，两次用 java，每次都以为是最后一次用。

```
/**
 * 预处理数据集，并生成Arff文件格式
 * @param dataDir  原始文档目录
 * @param desTi      存储的目标文件
 * @throws Exception
 */
public void priProcessData(String dataDir,String desTi) throws
Exception{

    //将dataDir目录下的所有文档转换成字符串属性的形式存储
    TextDirectoryLoader tdl = new TextDirectoryLoader();
    tdl.setDirectory(new File(dataDir));
    Instances ins = tdl.getDataSet();
    ins.setClassIndex(0);

    //将字符串属性转换为表示词频的词属性向量空间
    StringToWordVector filter = new StringToWordVector();
    filter.setUseStoplist(true);
    filter.setTFTransform(true);
    filter.setIDFTransform(true);
    LovinsStemmer  stemmer = new LovinsStemmer ();
    filter.setStemmer(stemmer);
    filter.setMinTermFreq(5);
    filter.setWordsToKeep(500);
    filter.setInputFormat(ins);
    Instances newtrain = Filter.useFilter(ins, filter);
    BufferedWriter bw = new BufferedWriter(new FileWriter(new
        File(desTi)));
```

```
    bw.write(newtrain.toString());
    bw.flush();
    bw.close();

}
```

weka.filters.unsupervised.attribute.StringToWordVector 注释写到：Converts String attributes into a set of attributes representing word occurrence (depending on the tokenizer) information from the text contained in the strings. The set of words (attributes) is determined by the first batch filtered (typically training data). 将 String 属性转换成一个表示词出现信息的属性集合，出现信息是从字符串中得到的文本中得到，词的集合由第一次批过滤来确定。

```
public boolean input(Instance instance) throws Exception {

    if (getInputFormat() == null) {
        throw new IllegalStateException("No input instance format
            defined");
    }
    if (m_NewBatch) {
        resetQueue();
        m_NewBatch = false;
    }
    if (isFirstBatchDone()) {
        FastVector fv = new FastVector();
        int firstCopy = convertInstancewoDocNorm(instance, fv);
        Instance inst = (Instance) fv.elementAt(0);
        if (m_filterType != FILTER_NONE) {
            normalizeInstance(inst, firstCopy);
        }
        push(inst);
        return true;
    } else {
        bufferInput(instance);
        return false;
    }
}
```

m_NewBatch 是表示是不是一个新的 Batch，现在当然是了，所以 resetQueue：

```
/** The output instance queue */
private Queue m_OutputQueue = null;
protected void resetQueue() {

    m_OutputQueue = new Queue();
}
```

重置 m_OutputQueue。

isFirstBatchDone 的代码如下：

```
/** True if the first batch has been done */
```

```
protected boolean m_FirstBatchDone = false;
/**
 * Returns true if the first batch of instances got processed. Necessary
 * for supervised filters, which "learn" from the first batch and then
 * shouldn't get updated with subsequent calls of batchFinished().
*/
public boolean isFirstBatchDone() {
    return m_FirstBatchDone;
}
```

如果 instances 第一批已经处理过了，返回真，对于监督学习是必需的，它从第一批中学习，然后不应该再调用 batchFinished 更新。现在 m_FirstBatchDone 为 false 执行 bufferInput：

```
protected void bufferInput(Instance instance) {

    if (instance != null) {
        copyValues(instance, true);
        m_InputFormat.add(instance);
    }
}
```

再看 copyValues 和 add 函数实在没什么意义，向下看 batchFinished 函数中调用了 determineDictionary 函数：

```
/**
 * a file containing stopwords for using others than the default Rainbow
 * ones.
 */
private File m_Stopwords = new File(System.getProperty("user.dir"));
*****************************************************************
// initialize stopwords
Stopwords stopwords = new Stopwords();
if (getUseStoplist()) {
    try {
        if (getStopwords().exists() && !getStopwords().isDirectory())
            stopwords.read(getStopwords());
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

初始化停词，Stopwords 中自己有几百个的停词。

```
// Operate on a per-class basis if class attribute is set
int classInd = getInputFormat().classIndex();
int values = 1;
if (!m_doNotOperateOnPerClassBasis && (classInd != -1)) {
    values = getInputFormat().attribute(classInd).numValues();
}
```

```
// TreeMap dictionaryArr [] = new TreeMap[values];
TreeMap[] dictionaryArr = new TreeMap[values];
for (int i = 0; i < values; i++) {
    dictionaryArr[i] = new TreeMap();
}
```

Values 是类别的个数，dictionaryArr 是每个类别的带记数的字典。

```
private void determineSelectedRange() {

    Instances inputFormat = getInputFormat();

    // Calculate the default set of fields to convert
    if (m_SelectedRange == null) {
        StringBuffer fields = new StringBuffer();
        for (int j = 0; j < inputFormat.numAttributes(); j++) {
            if (inputFormat.attribute(j).type() == Attribute.STRING)
                fields.append((j + 1) + ",");
        }
        m_SelectedRange = new Range(fields.toString());
    }
    m_SelectedRange.setUpper(inputFormat.numAttributes() - 1);

    // Prevent the user from converting non-string fields
    StringBuffer fields = new StringBuffer();
    for (int j = 0; j < inputFormat.numAttributes(); j++) {
        if (m_SelectedRange.isInRange(j)
                && inputFormat.attribute(j).type() == Attribute.STRING)
            fields.append((j + 1) + ",");
    }
    m_SelectedRange.setRanges(fields.toString());
    m_SelectedRange.setUpper(inputFormat.numAttributes() - 1);
}
```

如果没有选择范围，就把所有为 String 类型的属性都认为要转换成 word vector。如果选择了范围，如果选择的属性的确是 String 类型，就加入这些属性。

```
// Tokenize all training text into an orderedMap of "words".
long pruneRate = Math.round((m_PeriodicPruningRate / 100.0)
        * getInputFormat().numInstances());
```

计算有用剪除的比例。

```
// Iterate through all relevant string attributes of the current
// instance
Hashtable h = new Hashtable();
for (int j = 0; j < instance.numAttributes(); j++) {
    if (m_SelectedRange.isInRange(j)
            && (instance.isMissing(j) == false)) {
```

```
        // Get tokenizer
        m_Tokenizer.tokenize(instance.stringValue(j));

        // Iterate through tokens, perform stemming, and remove
        // stopwords
        // (if required)
        while (m_Tokenizer.hasMoreElements()) {
            String word = ((String) m_Tokenizer.nextElement())
                    .intern();

            if (this.m_lowerCaseTokens == true)
                word = word.toLowerCase();

            word = m_Stemmer.stem(word);

            if (this.m_useStoplist == true)
                if (stopwords.is(word))
                    continue;

            if (!(h.contains(word)))
                h.put(word, new Integer(0));

            Count count = (Count) dictionaryArr[vInd].get(word);
            if (count == null) {
                dictionaryArr[vInd].put(word, new Count(1));
            } else {
                count.count++;
            }
        }
    }
}
```

对循环的每个样本，再对它的属性进行循环，将它要 to word vector 的属性先用 Tokenizer 分词。将分词所得的词词干化，还要将停词去掉，最后在相应的属性 dictionaryArr 下，或是这个词没出现过，加入这个词，或是出现过，将词出现次数加 1。

```
// updating the docCount for the words that have occurred in this
// instance(document).
Enumeration e = h.keys();
while (e.hasMoreElements()) {
    String word = (String) e.nextElement();
    Count c = (Count) dictionaryArr[vInd].get(word);
    if (c != null) {
        c.docCount++;
    } else
        System.err
```

```
                .println("Warning: A word should definitely be in the "
                        + "dictionary.Please check the code");
}
```

将这些词的 docCount 加 1 ，刚才不加是因为一个词可能多次出现。

```
if (pruneRate > 0) {
    if (i % pruneRate == 0 && i > 0) {
        for (int z = 0; z < values; z++) {
            Vector d = new Vector(1000);
            Iterator it = dictionaryArr[z].keySet().iterator();
            while (it.hasNext()) {
                String word = (String) it.next();
                Count count = (Count) dictionaryArr[z].get(word);
                if (count.count <= 1) {
                    d.add(word);
                }
            }
            Iterator iter = d.iterator();
            while (iter.hasNext()) {
                String word = (String) iter.next();
                dictionaryArr[z].remove(word);
            }
        }
    }
}
```

这里可以看出 pruneRate 这个是一定多少个样，就要执行的一次的值，这当然可以节约内存，但也有缺点，一个词我前面出现很少，总是被 pruned，后面才多起来，前面的就没算到。这也就是为什么要把类别分开的原因吧，不然一个类别样本少的特别点的词就被 prune 了。可以看一下过程，到 pruneRate 的位数后，就把词典里所以只出现两次以下的全去掉，作法是先把要删的词加入到 d 这个集合中，再把这个集合中的词全部删掉，这样写我想应该是逻辑更清楚些。

```
// Figure out the minimum required word frequency
int totalsize = 0;
int prune[] = new int[values];
for (int z = 0; z < values; z++) {
    totalsize += dictionaryArr[z].size();

    int array[] = new int[dictionaryArr[z].size()];
    int pos = 0;
    Iterator it = dictionaryArr[z].keySet().iterator();
    while (it.hasNext()) {
        String word = (String) it.next();
        Count count = (Count) dictionaryArr[z].get(word);
        array[pos] = count.count;
        pos++;
```

```
    }

    // sort the array
    sortArray(array);
    if (array.length < m_WordsToKeep) {
        // if there aren't enough words, set the threshold to
        // minFreq
        prune[z] = m_minTermFreq;
    } else {
        // otherwise set it to be at least minFreq
        prune[z] = Math.max(m_minTermFreq, array[array.length
                - m_WordsToKeep]);
    }
}
```

将每个类别的字记数放到 array 里去，再对它进行排序，词的个数少于 m_WordsToKeep，那就全留下，也就是出现 1 次就可以了。否则就 m_minTermFreq 和刚才排对序的数组中第 m_WordsToKeep 元素之间的最大值，把它赋给 prune[z]。而 totalSize 是全部词的数量。

```
// Convert the dictionary into an attribute index
// and create one attribute per word
FastVector attributes = new FastVector(totalsize
        + getInputFormat().numAttributes());

// Add the non-converted attributes
int classIndex = -1;
for (int i = 0; i < getInputFormat().numAttributes(); i++) {
    if (!m_SelectedRange.isInRange(i)) {
        if (getInputFormat().classIndex() == i) {
            classIndex = attributes.size();
        }
        attributes.addElement(getInputFormat().attribute(i).copy());
    }
}
```

Attributes 大小为词的总大小(重复算的)和全部原属性的大小，接下来，把没有 to word vector 的原属性加到 attributes 中。

```
// Add the word vector attributes (eliminating duplicates
// that occur in multiple classes)
TreeMap newDictionary = new TreeMap();
int index = attributes.size();
for (int z = 0; z < values; z++) {
    Iterator it = dictionaryArr[z].keySet().iterator();
    while (it.hasNext()) {
        String word = (String) it.next();
        Count count = (Count) dictionaryArr[z].get(word);
        if (count.count >= prune[z]) {
```

```
            if (newDictionary.get(word) == null) {
                newDictionary.put(word, new Integer(index++));
                attributes.addElement(new Attribute(m_Prefix + word));
            }
        }
    }
}
```

这里是将词合起来成为 newDictionary，如果一个词不会被去除（即它出现次次大于 prune[z]）就将这个词做为一个新的属性加进去，属性的名字是 m_Prefix+word。

```
// Compute document frequencies
m_DocsCounts = new int[attributes.size()];
Iterator it = newDictionary.keySet().iterator();
while (it.hasNext()) {
    String word = (String) it.next();
    int idx = ((Integer) newDictionary.get(word)).intValue();
    int docsCount = 0;
    for (int j = 0; j < values; j++) {
        Count c = (Count) dictionaryArr[j].get(word);
        if (c != null)
            docsCount += c.docCount;
    }
    m_DocsCounts[idx] = docsCount;
}
```

将一个词的几个类别的 docCount 合并，合并后的计数放到 m_DocCounts 里。

```
// Trim vector and set instance variables
attributes.trimToSize();
m_Dictionary = newDictionary;
m_NumInstances = getInputFormat().numInstances();

// Set the filter's output format
Instances outputFormat = new Instances(getInputFormat().relationName(),
        attributes, 0);
outputFormat.setClassIndex(classIndex);
setOutputFormat(outputFormat);
```

trimToSize 是把刚才申请空间时，那些没用到的去掉。下面就产生一个新的输出格式，relationName 就是 arff 的名字，一般显示在.arff 文件的第一行。

```
// Convert all instances w/o normalization
FastVector fv = new FastVector();
int firstCopy = 0;
for (int i = 0; i < m_NumInstances; i++) {
    firstCopy = convertInstancewoDocNorm(getInputFormat().instance(
            i), fv);
}
```

convertInstancewoDocNorm 的代码拆开来：

```
// Convert the instance into a sorted set of indexes
TreeMap contained = new TreeMap();

// Copy all non-converted attributes from input to output
int firstCopy = 0;
for (int i = 0; i < getInputFormat().numAttributes(); i++) {
    if (!m_SelectedRange.isInRange(i)) {
        if (getInputFormat().attribute(i).type() != Attribute.STRING) {
            // Add simple nominal and numeric attributes directly
            if (instance.value(i) != 0.0) {
                contained.put(new Integer(firstCopy), new Double(
                        instance.value(i)));
            }
        } else {
            if (instance.isMissing(i)) {
                contained.put(new Integer(firstCopy), new Double(
                        Instance.missingValue()));
            } else {
                // If this is a string attribute, we have to first add
                // this value to the range of possible values, then add
                // its new internal index.
                if (outputFormatPeek().attribute(firstCopy).numValues()
                    == 0) {
                    // Note that the first string value in a
                    // SparseInstance doesn't get printed.
                    outputFormatPeek()
                            .attribute(firstCopy)
                            .addStringValue(
                                    "Hack to defeat SparseInstance bug");
                }
                int newIndex = outputFormatPeek().attribute(firstCopy)
                        .addStringValue(instance.stringValue(i));
                contained.put(new Integer(firstCopy), new Double(
                        newIndex));
            }
        }
        firstCopy++;
    }
}
```

对这个样本的所有属性循环，如果不是 STRING 类型，自然没有 to word vector 的可能，相应属性值不为 0，就加入到 contained 中去。若为缺失值，就赋给 Instance.missingValue() 值。如果它是一个 string 属性，我们需要先将这个值加入到可能值的范围中，再加入它的新内部索引。

```
for (int j = 0; j < instance.numAttributes(); j++) {
```

```
    if (m_SelectedRange.isInRange(j)
            && (instance.isMissing(j) == false)) {

        m_Tokenizer.tokenize(instance.stringValue(j));

        while (m_Tokenizer.hasMoreElements()) {
            String word = (String) m_Tokenizer.nextElement();
            if (this.m_lowerCaseTokens == true)
                word = word.toLowerCase();
            word = m_Stemmer.stem(word);
            Integer index = (Integer) m_Dictionary.get(word);
            if (index != null) {
                if (m_OutputCounts) { // Separate if here rather than
                                      // two lines down to avoid
                                      // hashtable lookup
                    Double count = (Double) contained.get(index);
                    if (count != null) {
                        contained.put(index, new Double(count
                                .doubleValue() + 1.0));
                    } else {
                        contained.put(index, new Double(1));
                    }
                } else {
                    contained.put(index, new Double(1));
                }
            }
        }
    }
}
```

这里和刚才求字典时差不多，只是没有停词这一步，这是因为如果在词典中找不到就可以把停词过掉了（当然，也去掉了一些低频词），即 index == null。m_OutputCounts 为 true 为要记录出现了多少次，而 false 只管出现与否。

```
// Doing TFTransform
if (m_TFTransform == true) {
    Iterator it = contained.keySet().iterator();
    for (int i = 0; it.hasNext(); i++) {
        Integer index = (Integer) it.next();
        if (index.intValue() >= firstCopy) {
            double val = ((Double) contained.get(index)).doubleValue();
            val = Math.log(val + 1);
            contained.put(index, new Double(val));
        }
    }
}
```

如果在进行 TFTransform，就把求得的词向量中的值+1 后求 log，+1 应该是处理 1 这个情况的，因为 log1 等于 0。

```
// Doing IDFTransform
if (m_IDFTransform == true) {
    Iterator it = contained.keySet().iterator();
    for (int i = 0; it.hasNext(); i++) {
        Integer index = (Integer) it.next();
        if (index.intValue() >= firstCopy) {
            double val = ((Double) contained.get(index)).doubleValue();
            val = val * Math.log(m_NumInstances
                    / (double) m_DocsCounts[index.intValue()]);
            contained.put(index, new Double(val));
        }
    }
}
```

这里相当于求 tf-idf 了，右边的 val 为 tf，而 idf 为 log(N/n)，N 为总出现次数，n 为包含有这个词的文档数。

```
// Convert the set to structures needed to create a sparse instance.
double[] values = new double[contained.size()];
int[] indices = new int[contained.size()];
Iterator it = contained.keySet().iterator();
for (int i = 0; it.hasNext(); i++) {
    Integer index = (Integer) it.next();
    Double value = (Double) contained.get(index);
    values[i] = value.doubleValue();
    indices[i] = index.intValue();
}

Instance inst = new SparseInstance(instance.weight(), values, indices,
        outputFormatPeek().numAttributes());
inst.setDataset(outputFormatPeek());

v.addElement(inst);
```

把 contained 中的值再转到 values 和 indices 中。存到稀疏样本类 SparseInstance 中。再将这个样本保存到 v 这个 FastVector 中。

```
// Need to compute average document length if necessary
if (m_filterType != FILTER_NONE) {
    m_AvgDocLength = 0;
    for (int i = 0; i < fv.size(); i++) {
        Instance inst = (Instance) fv.elementAt(i);
        double docLength = 0;
        for (int j = 0; j < inst.numValues(); j++) {
            if (inst.index(j) >= firstCopy) {
                docLength += inst.valueSparse(j)
```

```
                    * inst.valueSparse(j);
            }
        }
        m_AvgDocLength += Math.sqrt(docLength);
    }
    m_AvgDocLength /= m_NumInstances;
}

// Perform normalization if necessary.
if (m_filterType == FILTER_NORMALIZE_ALL) {
    for (int i = 0; i < fv.size(); i++) {
        normalizeInstance((Instance) fv.elementAt(i), firstCopy);
    }
}

// Push all instances into the output queue
for (int i = 0; i < fv.size(); i++) {
    push((Instance) fv.elementAt(i));
}
```

回到 batchFinished 中，如果要过滤，过滤的几个定义如下：

```
/** normalization: No normalization. */
public static final int FILTER_NONE = 0;
/** normalization: Normalize all data. */
public static final int FILTER_NORMALIZE_ALL = 1;
/** normalization: Normalize test data only. */
public static final int FILTER_NORMALIZE_TEST_ONLY = 2;
```

m_AvgDocLength 并不是词的个数，而是 doc 的 tfidf（如果是 tf-idf）的绝对值之和平均值。如果要对所有的样本进行 normalization：

```
private void normalizeInstance(Instance inst, int firstCopy)
        throws Exception {

    double docLength = 0;

    // Compute length of document vector
    for (int j = 0; j < inst.numValues(); j++) {
        if (inst.index(j) >= firstCopy) {
            docLength += inst.valueSparse(j) * inst.valueSparse(j);
        }
    }
    docLength = Math.sqrt(docLength);

    // Normalize document vector
    for (int j = 0; j < inst.numValues(); j++) {
        if (inst.index(j) >= firstCopy) {
```

```
            double val = inst.valueSparse(j) * m_AvgDocLength / docLength;
            inst.setValueSparse(j, val);
        }
    }
}
```

很简单，就是如刚才一样求得 docLength，再乘以 m_AvgDocLength，再除以 docLength。

```
protected void push(Instance instance) {

    if (instance != null) {
        if (instance.dataset() != null)
            copyValues(instance, false);
        instance.setDataset(m_OutputFormat);
        m_OutputQueue.push(instance);
    }
}
```

设置.arff 的文件头，再将样本加入到 m_OutputQueue。