IFN645 Major Assignment

Name: Timothy Jeoung

Student ID: n10887601

Task 1: Association mining in Java (13 Marks)

Q1. *<u>Comparison of two frequent pattern mining algorithms with 3 different minimum supports on ‘bank.arff ‘dataset</u>*

Minimum support with **0.1**

```
=============  APRIORI - STATS =============
 Candidates count : 1906
 The algorithm stopped at size 6
 Frequent itemsets count : 1354
 Maximum memory usage : 64.75152587890625 mb
 Total time ~ 2296 ms
===================================================
=============  FP-GROWTH 2.42 - STATS =============
 Transactions count from database : 45211
 Max memory usage: 46.094627380371094 mb
 Frequent itemsets count : 1354
 Total time ~ 187 ms
===================================================
```

Minimum support with **0.3**

```
=============  APRIORI - STATS =============
 Candidates count : 198
 The algorithm stopped at size 6
 Frequent itemsets count : 154
 Maximum memory usage : 64.12033081054688 mb
 Total time ~ 314 ms
===================================================
=============  FP-GROWTH 2.42 - STATS =============
 Transactions count from database : 45211
 Max memory usage: 36.80833435058594 mb
 Frequent itemsets count : 154
 Total time ~ 143 ms
===================================================
```

Minimum support with **0.5**

```
============= APRIORI - STATS =============
 Candidates count : 68
 The algorithm stopped at size 5
 Frequent itemsets count : 37
 Maximum memory usage : 64.79058837890625 mb
 Total time ~ 155 ms
===================================================
============= FP-GROWTH 2.42 - STATS =============
 Transactions count from database : 45211
 Max memory usage: 36.669410705566406 mb
 Frequent itemsets count : 37
 Total time ~ 133 ms
===================================================
```

Minsup	Apriori Time(ms)	FP-Growth Time(ms)	Apriori Memory (mb)	FP-Growth Memory(mb)	#Patterns
0.1	2296	187	46.09	46.09	1354
0.3	314	143	64.12	36.80	154
0.5	155	133	64.79	36.66	37

- Referring to 3 results of Apriori and FP-Growth algorithms the **FP-Growth algorithm** has better performance with minimum support of **0.5.**

Q2. *Using chosen algorithm (FP-Growth) generate top 5 most frequent size-3 patterns from the yes class and no class*

Yes-class top 5 most frequent size 3 patterns output:

```
marital=married default_credit=no loan=no #SUP: 2471
age=21-30s default_credit=no loan=no #SUP: 2519
default_credit=no loan=no call_duration=100-500s #SUP: 2712
default_credit=no housing=no loan=no #SUP: 3120
default_credit=no loan=no past_marketing=unknown #SUP: 2988
```

No-class top 5 most frequent size 3 patterns output:

```
default_credit=no loan=no marital=married #SUP: 19799
default_credit=no past_marketing=unknown marital=married #SUP: 20357
default_credit=no loan=no call_duration=100-500s #SUP: 21217
default_credit=no call_duration=100-500s past_marketing=unknown #SUP: 21352
default_credit=no loan=no past_marketing=unknown #SUP: 27371
```

- Referring to the results of *yes* class and *no* class, there is a similarity of **'default_credit=no'**, **'call_duration=100-500s'**, **'past_marketing=unknown'** and **'loan=no'** being included in the patterns. However, there is difference for *yes* class which includes **'housing=no'** and **'age=21-30s'** and *no* class doesn't include these two at all. This can tell us that customer who has subscribed the term-deposit product will have a chance of not having a house and their age will be around 21-30s.

Q3. *Top 5 most frequent maximum patterns from yes-class and no-class*

Yes-class top 5 most frequent maximum patterns output:

```
default_credit=no balance=below-1k #SUP: 2521
default_credit=no marital=married #SUP: 2735
default_credit=no loan=no age=21-30s #SUP: 2519
default_credit=no loan=no call_duration=100-500s #SUP: 2712
default_credit=no loan=no housing=no #SUP: 3120
```

No-class top 5 most frequent maximum patterns output:

```
default_credit=no housing=yes #SUP: 22789
default_credit=no marital=married #SUP: 24031
loan=no call_duration=100-500s #SUP: 21540
default_credit=no past_marketing=unknown call_duration=100-500s #SUP: 21352
default_credit=no past_marketing=unknown loan=no #SUP: 27371
```

- In terms of maximum patterns, above results shows that *yes*-class and *no*-class have a lot of similar attributes except the **'housing'** attribute has different description. The yes-class has **'housing=no'** and no-class has **'housing=yes'**. Moreover, yes-class has an attribute of **'balance=below-1k'**.

Q4. *Use three algorithms to generate frequent closed patterns for entire dataset*

```
=============  FP-GROWTH 2.42 - STATS =============
 Transactions count from database : 45211
 Max memory usage: 40.487091064453125 mb
 Frequent itemsets count : 154
 Total time ~ 162 ms
===================================================
=============  APRIORI - STATS =============
 Candidates count : 198
 The algorithm stopped at size 6, because there is no candidate
 Frequent closed itemsets count : 154
 Maximum memory usage : 33.30458068847656 mb
 Total time ~ 317 ms
===================================================
=============  FP-Close v0.96r14  - STATS =============
 Transactions count from database : 45211
 Max memory usage: 60.06244659423828 mb
 Closed frequent itemset count : 154
 Total time ~ 137 ms
===================================================
=============  CHARM v96r6 Bitset - STATS =============
 Transactions count from database : 45211
 Frequent closed itemsets count : 154
 Total time ~ 50 ms
 Maximum memory usage : 48.47520446777344 mb
===================================================
```

Following table displays the record of 3 algorithms time efficiency in 5 different minimum supports

Minsup	**AprioriClose Time (ms)**	**FPClose Time(ms)**	**Charm Time (ms)**	No. of closed patterns
0.5	159	121	34	37
0.4	166	110	40	81
0.3	317	137	50	154
0.2	706	148	64	401
0.1	2418	180	104	1482

According to the table, it shows that when the minimal support is reduced from 0.5 to 0.1, the time taken by the Apriori-like method AprioriClose increased about 1000 times, while the FP-Growth based methods FPClose and Charm has a moderate increase in terms of the time, about 3-4 times. Therefore, FPClose and Charm are much more efficient than AprioriClose for larger datasets.

Q5. *Generate top 10 most frequent association rules with subscribed=yes & subscribed=no as the consequent.*

a) *List the rules generated for each class*

No-class 10 rules:

Rules	Antecedents	Consequent	#Sup	#Conf
1	default_credit=no	subscribed=no	39159	0.88
2	past_marketing=unknown	subscribed=no	33573	0.90
3	loan=no	subscribed=no	33162	0.87
4	default_credit=no, past_marketing=unknown	subscribed=no	32862	0.90
5	default_credit=no, loan=no	subscribed=no	32685	0.87
6	loan=no, past_marketing=unknown	subscribed=no	27814	0.90
7	default_credit=no, loan=no, past_marketing=unknown	subscribed=no	27371	0.90
8	call_duration=100-500s	subscribed=no	26044	0.90
9	default_credit=no, call_duration=100-500s	subscribed=no	25538	0.89
10	martial= married	subscribed=no	24459	0.89

Yes-class 10 rules:

Rules	Antecedents	Consequent	#Sup	#Conf
1	call_duration=500-1k	subscribed=yes	1646	0.38
2	default_credit=no, call_duration=500-1k	subscribed=yes	1614	0.38
3	loan=no, call_duration=500-1k	subscribed=yes	1448	0.39
4	default_credit=no, loan=no, call_duration=500-1k	subscribed=yes	1428	0.39
5	past_marketing=unknown, call_duration=500-1k	subscribed=yes	1242	0.35
6	default_credit=no, past_marketing=unknown, call_duration=500-1k	subscribed=yes	1214	0.34
7	loan=no, past_marketing=unknown, call_duration=500-1k	subscribed=yes	1079	0.35
8	default_credit=no, loan=no, past_marketing=unknown, call_duration=500-1k	subscribed=yes	1061	0.35
9	default_credit=no, past_marketing=success	subscribed=yes	978	0.64
10	past_marketing=success	subscribed=yes	978	0.64

b) *Any redundant rules in each set of the rules? If yes, list them and state the reason why they are redundant.*

- Theoretically the support needs to be exact same to be redundant. Therefore, for 'Yes-class 10 rules' there is redundancy in rule 9 and rule 10 with same support and very similar confidence. As rule number 9 has more antecedents it can be removed from the list.

Task 2: Classification in Weka and Java (13 Marks)

2.1 Data Analysis in Weka (AttributeSelectedClassifier)

1) *Selecting 4 classification algorithms with specific evaluator, search method and ranker:*

Table 1: Using evaluator= **CfsSubsetEval** & search= **BestFirst**:

Classifier	Correctly Classified Instances
NaiveBayes	89.61%
NaiveBayeMuliNomial	N/A (Error)s
iBk	89.78%
PART	89.81%
OneR	89.28%
ZeroR	88.30%
J48	89.84%
RandomForest	89.78%

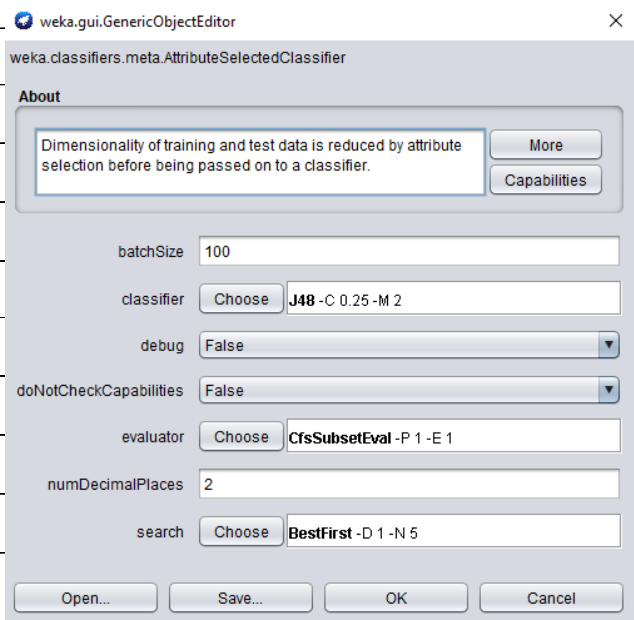
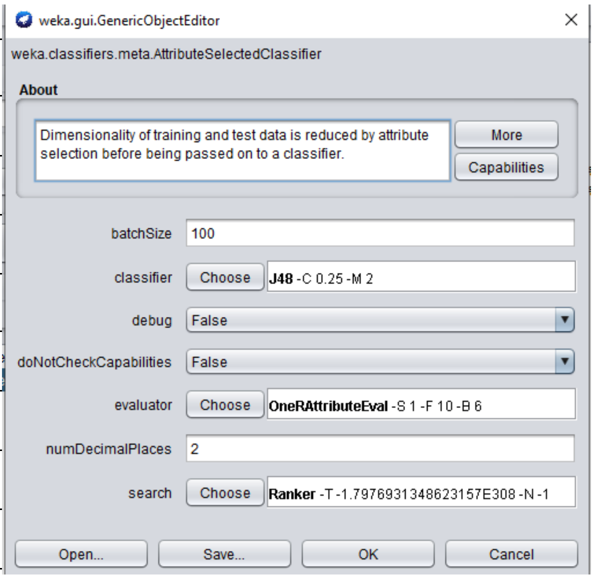


Table 2: Using evaluator= **OneRAttributeEval**, search= **Ranker**:

Classifier	Correctly Classified Instances
NaiveBayes	89.24%
NaiveBayeMuliNomial	N/A (Error)
iBk	88.83%
PART	89.19%
OneR	89.28%
ZeroR	88.30%
J48	89.77%
RandomForest	88.82%

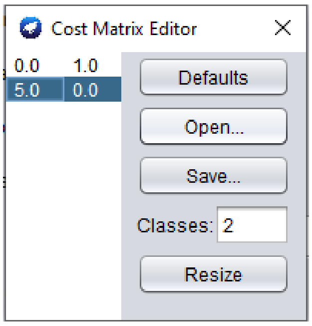


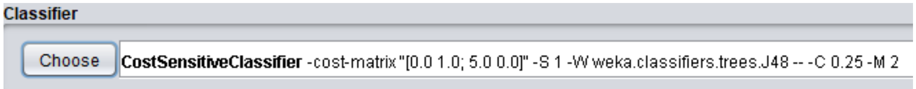
For Using evaluator= **InfoGainAttributeEval**, search= **Ranker** and Using evaluator= **GainRatioAttributeEval**, search= **Ranker** had same result as above table 2. Moreover, **WrapperSubsetEval** takes too much time because of the complexity; also, it had lower correctly classified instances compared to two tables so it was never chosen as an evaluator.

- Thus, the best 4 classification algorithms will be selected from table 1 using **CfsSubsetEval** as an evaluator and search with **BestFirst.** The selected 4 classification algorithms are the following: iBk, PART, J48 and RandomForest.

2) *Select 3 algorithms from the previous question and evaluate the cost analysis*

Considering the cost of the 4 algorithms to see which performs better accuracy the '**CostSensitiveClassifer**' has been applied on classifier. Moreover, '**subscribed = yes**' is more significant and we want to minimize the classification errors of yes-class. Thus, the cost matrix has been set are the following screenshot:





Results of the 4 algorithms with cost matrix being edited:

Classifier	Correctly Classified Instances	Incorrectly Classified Instances	Total Cost
iBk	82.60%	17.39%	15729
PART	82.74%	17.25%	14659
J48	83.61%	16.38%	12924
RandomForest	82.81%	17.18%	15598

- Referring to the costs and incorrectly classified instances, to have minimized classification error to the yes-class the 3 algorithms needs to be selected by lowest total cost and incorrect classified instances. Thus, it will be the following: **PART**, **J48** and **Random Forest**.

3) *Using* ***AttributeSelectedClassifier*** *and using chosen* ***3 algorithms*** *from question 2. Also, setting evaluator to '****GainRatioAtribueEval****' and change search to '****Ranker'*** *then setting the '****numToSelect****' to 4-8 to find the best number of attributes referring to the classification performance.*

Attributes with **8**:

Classifier	Correctly Classified Instances	Incorrectly Classified Instances
PART	89.51%	10.48%
J48	89.80%	10.19%
RandomForest	89.11%	10.88%

Attributes with **7**:

Classifier	Correctly Classified Instances	Incorrectly Classified Instances
PART	89.69%	10.30%
J48	89.79%	10.20%
RandomForest	89.40%	10.59%

Attributes with **6**:

Classifier	Correctly Classified Instances	Incorrectly Classified Instances
PART	89.83%	10.11%
J48	89.82%	10.17%
RandomForest	89.73%	10.26%

Attributes with **5**:

Classifier	Correctly Classified Instances	Incorrectly Classified Instances
PART	89.82%	10.17%
J48	89.86%	10.13%
RandomForest	89.82%	10.17%

Attributes with **4**:

Classifier	Correctly Classified Instances	Incorrectly Classified Instances
PART	89.80%	10.19%
J48	89.82%	10.17%
RandomForest	89.84%	10.15%

- According to above tables with different number of attributes, it has shown **PART** has better performance with **6** attributes, **J48** have better performance with **5** attributes and **RandomForest** have better performance with **4** attributes. The evaluator among 'GainRatioAttributeEval', 'InfoGainAttriuteEval' and 'OneRAttributeEval' the GainRatioAttributeEval had slightly better accuracy. Therefore, **GainRatioAttributeEval** has been used as the evaluator.

2.2 Java Program Classification task

1) Classification accuracy and total cost

```
IBk
Classification accuracy: 0.8260379111278229
Total Cost: 15729.0
PART
Classification accuracy: 0.827409258808697
Total Cost: 14659.0
J48
Classification accuracy: 0.8361460706465241
Total Cost: 12924.0
RandomForest
Classification accuracy: 0.8281391696710978
Total Cost: 15598.0
```

2) Number of attributes of each of the three algorithms with screenshots:

6 attributes have performed best for **PART**:

```
PART:
Correctly classified instances: 40637.0
Accuracy:0.8988299307690606
J48:
Correctly classified instances: 40611.0
Accuracy:0.8982548494835327
RandomForest:
Correctly classified instances: 40568.0
Accuracy:0.8973037535113136
```

5 attributes has performed best for **J48**:

```
PART:
Correctly classified instances: 40612.0
Accuracy:0.8982769679945146
J48:
Correctly classified instances: 40627.0
Accuracy:0.8986087456592422
RandomForest:
Correctly classified instances: 40609.0
Accuracy:0.8982106124615691
```

4 attributes have performed best for **RandomForest**:

```
PART:
Correctly classified instances: 40603.0
Accuracy:0.898077901395678
J48:
Correctly classified instances: 40610.0
Accuracy:0.898232730972551
RandomForest:
Correctly classified instances: 40620.0
Accuracy:0.8984539160823694
```

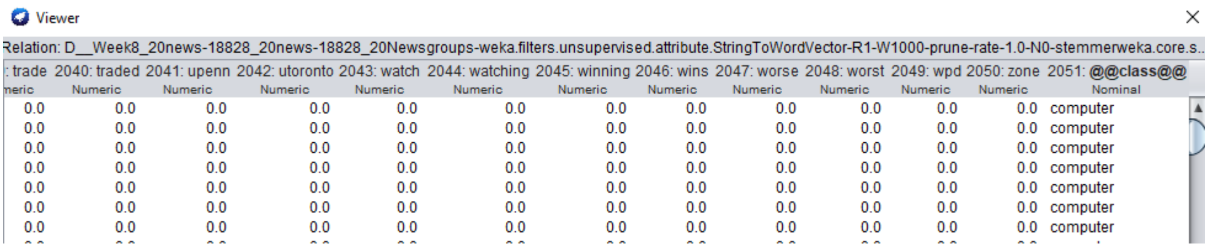
Task 3: Text classification in Weka and Java (12 marks)

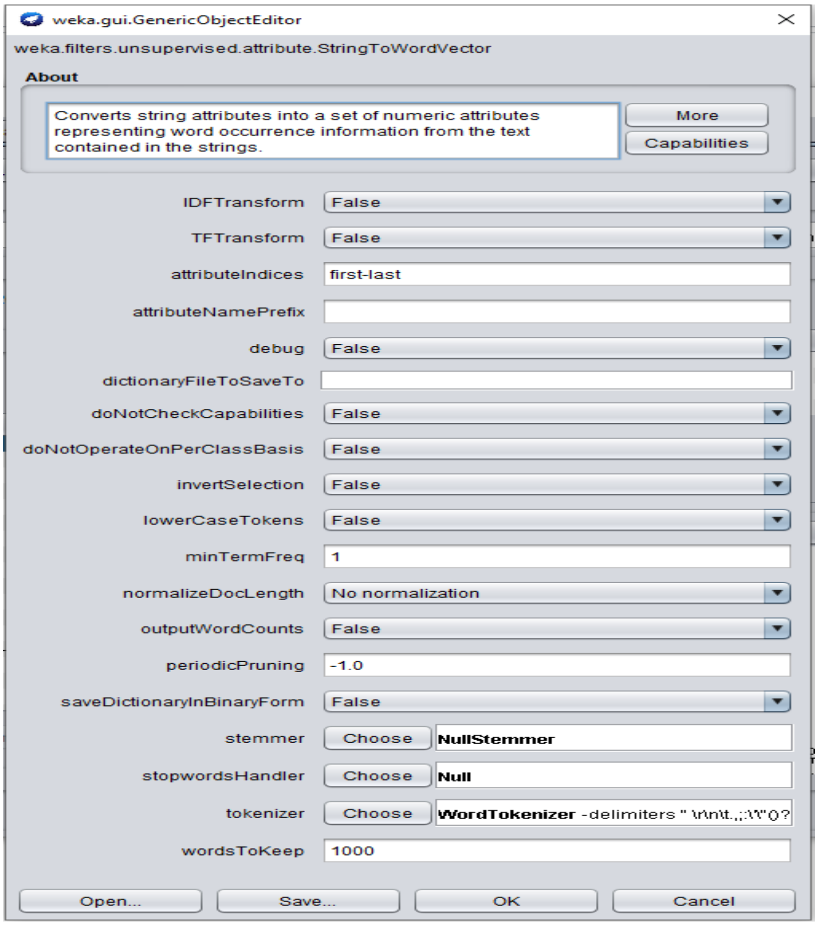
3.1 Attribute selection in Weka

1) Working process in Weka to determine the values for the parameters in the filter.

1.1 Filters -> Unsupervised > attribute -> StringToWordVector then click apply by going to Edit it shows that there are 2051 attributes.

1.2 Changed the '@@class@@' as 'attribute as class' to change the class attribute as the last entry of attributes list



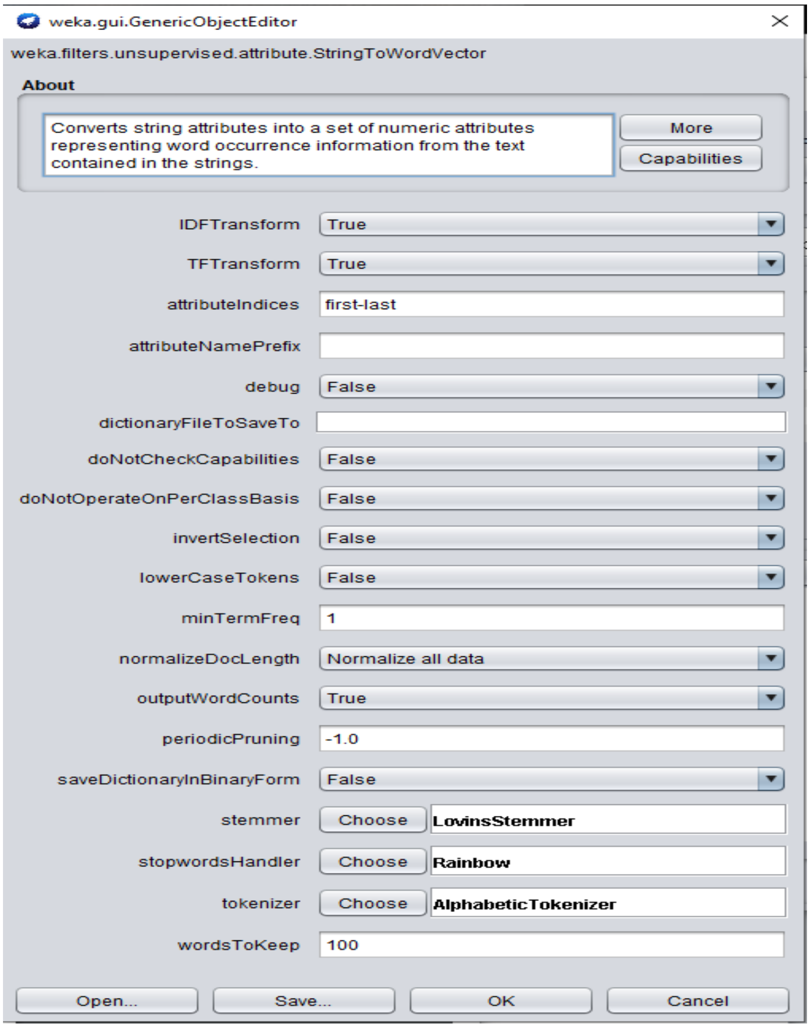


- Above screenshot is the **Default parameter** the accuracy using **J48** classifier is:

```
Correctly Classified Instances        8222               58.6532 %
Incorrectly Classified Instances      5796               41.3468 %
```

2) Parameters in the filters to be tuned and changed values:

- Need to perform the preprocessing by removing stop words, applying stemming, and removing digits etc. Thus, **IDFTransform**, **TFTransform**, **outputWordsCounts**, **stemmer**, **stopwordsHandler**, **tokenizer** and **wordsToKeep** has been changed as the following screenshot of edited parameters:



- First I have changed the **IDFTransfrom** to **true**, this changes the word frequencies in the document into fij x log.
- Changed **TFTransform** to **true**, this changes the word frequencies to transform into log (1+fij) where fij is the frequency of word i in document (instance) j.
- **normalizeDocLength** to '**Normalize all data'.** Sets whether if the word frequencies for a document (instance)
- **outputWordsCounts** changed to **true,** output word counts rather than Boolean 0 or 1(indicating presence or absence of a word).
- **Stemmer** was changed to LovinsStemmer algorithms because it is faster. It has effectively traded space for time, and with its large suffix set it needs just two major steps to remove a suffix.
- For **stopwordsHandler, Rainbow** is used. It's a program that performs statistical text classification. It is based on the *Bow* library.
- **Tokenizer** was changed to **Alphabetic Tokenizer** was used because it returns tokens that are maximal sequences of consecutive alphabetical characters.
- **wordsToKeep** is a number of words (per class if there is a class attribute assigned) to attempt to keep. So it was set to 100 to select 100 attributes. However, as there is 4 classes it still has 225 attributes

After the selecting the parameters from Meta, selected the **FilteredClassifier**. For filtered classifier, select the j48 as the classifier, and select filters -> Unsupervised > attribute -> **StringToWordVector** as the filter. Set the filter parameters with the same values as the final settings as above parameter. The approach used in FilteredClassifier is the right way to perform **text classifications** with cross validation evaluation. The evaluation results would be more accurate and more reliable. Running this classifier gives following result in screenshot:

- **8** tuned parameter accuracy filtered with **FilteredClassifer** using **J48** Classifier:

```
Correctly Classified Instances       10873               77.5646 %
Incorrectly Classified Instances      3145               22.4354 %
```

- **Filtered Classifier Weka version**

Classifier	Correctly classified instances	Accuracy
J48	10873	77.56
IBk	11548	82.37
SMO	11823	84.34
HoeffdingTree	10984	78.35

3.2 Java Program

1) *Using **tuned parameter** with 4 classification algorithms, **IBK**, **SMO**, **J48** and **HoeffdingTree***

FilteredClassifier Java version

Classifier	Correctly classified instances	Accuracy	Time(m/s)
J48	10873	77.56	545.12s
IBk	11548	82.37	119.54s
SMO	11823	84.34	227.85s
HoeffdingTree	10984	78.35	73.20

2) *Display correctly **classified instances results, accuracy** and **time taken**:*

```
J48
Correctly classified instances: 10873.0
Accuracy:0.7756455985161934
Executing J48: 545.1264635 seconds

SMO
Correctly classified instances: 11823.0
Accuracy:0.8434156085033528
Executing SMO: 227.8539976 seconds

IBk
Correctly classified instances: 11548.0
Accuracy:0.8237979740333856
Executing IBk: 119.5408583 seconds

HoeffdingTree
Correctly classified instances: 10984.0
Accuracy:0.7835639891567984
Executing HoeffdingTree: 73.2073416 seconds
```

3) *Which classifier performs the best in terms of time efficiency?*

Comparing the time efficiency the HoeffdingTree is the fastest classifier among J48, IBk and SMO. This is because HoeffdingTree is fast and operates easily on large data sets like News.arff. It also obtains significant superior accuracy on most of the largest classification datasets. This is because the data distribution is not changing over time it grows incrementally a decision tree based on the theoretical guarantees of the Hoeffding bound. A node is expanded as soon as there is sufficient statistical evidence that an optimal splitting feature exists, a decision based on the distribution-independent Hoeffding bound. The model learned by the Hoeffding tree is asymptotically nearly identical to the one built by a non-incremental learner, if the number of training instances is large enough (HUAWEI Noah's Ark Lab, 2016).