

# *COMP9313 Big Data Management*

20T2 Project2 Report



Minrui Lu  
z5277884

# 1 Evaluation

The F1 Score is 0.7483312619309965, the accuracy is 0.75375, the precision is 0.7467152515850681, the recall is 0.7537499999999999.

The evaluation of the code performance on test data is given below.

```
1 evaluator_acc = MulticlassClassificationEvaluator(predictionCol="prediction", metricName="accuracy")
2 evaluator_pre = MulticlassClassificationEvaluator(predictionCol="prediction", metricName="weightedPrecision")
3 evaluator_recall = MulticlassClassificationEvaluator(predictionCol="prediction", metricName="weightedRecall")
4 print("f1 score is:",evaluator.evaluate(pred_test, {evaluator.predictionCol:'final_prediction'}))
5 print("accuracy is: ",evaluator_acc.evaluate(pred_test, {evaluator_acc.predictionCol:'final_prediction'}))
6 print("precision is: ",evaluator_pre.evaluate(pred_test, {evaluator_pre.predictionCol:'final_prediction'}))
7 print("recall is: ",evaluator_recall.evaluate(pred_test, {evaluator_recall.predictionCol:'final_prediction'}))
8 spark.stop()

f1 score is: 0.7483312619309965
accuracy is: 0.75375
precision is: 0.7467152515850681
recall is: 0.7537499999999999
```

Figure 1. Evaluation of the original code

The formulas of accuracy, precision, recall, and F1 score are given below.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

$$F1\ Score = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall}$$

From the result above, it could be seen the F1 score, accuracy, precision and recall values are close to each other. This indicates that the proportion of correct positive predictions is similar to the proportion of correct negative predictions. This model can correctly classify three instances every four data items on average, and in four predictions, three of them are supposed to be correct.

However, compared to directly using Naive Bayes, Decision Tree Classifier, or Random Forest Classifier, the proposed method only outperforms the latter two and is inferior to the first one. Evaluation data is given below.

```

40 nb = NaiveBayes(featuresCol='features', labelCol='label', predictionCol='prediction', probabilityCol='probability', 1
41 model = nb.fit(training_set)
42 pred_test = model.transform(test_set)
43
44 evaluator = MulticlassClassificationEvaluator(predictionCol="prediction",metricName='f1')
45 evaluator_acc = MulticlassClassificationEvaluator(predictionCol="prediction", metricName="accuracy")
46 evaluator_pre = MulticlassClassificationEvaluator(predictionCol="prediction", metricName="weightedPrecision")
47 evaluator_recall = MulticlassClassificationEvaluator(predictionCol="prediction", metricName="weightedRecall")
48 print("f1 score is:",evaluator.evaluate(pred_test, {evaluator.predictionCol:'prediction'}))
49 print("accuracy is: ",evaluator_acc.evaluate(pred_test, {evaluator_acc.predictionCol:'prediction'}))
50 print("precision is: ",evaluator_pre.evaluate(pred_test, {evaluator_pre.predictionCol:'prediction'}))
51 print("recall is: ",evaluator_recall.evaluate(pred_test, {evaluator_recall.predictionCol:'prediction'}))
52 spark.stop()

```

```

f1 score is: 0.7594579098395631
accuracy is: 0.765
precision is: 0.7582808850082194
recall is: 0.765

```

Figure 2. Evaluation of using Naive Bayes

```

39 rf = RandomForestClassifier(featuresCol='features', labelCol='label', predictionCol='prediction',probabilityCol='prol
40 model = rf.fit(training_set)
41 pred_test = model.transform(test_set)
42
43 evaluator = MulticlassClassificationEvaluator(predictionCol="prediction",metricName='f1')
44 evaluator_acc = MulticlassClassificationEvaluator(predictionCol="prediction", metricName="accuracy")
45 evaluator_pre = MulticlassClassificationEvaluator(predictionCol="prediction", metricName="weightedPrecision")
46 evaluator_recall = MulticlassClassificationEvaluator(predictionCol="prediction", metricName="weightedRecall")
47 print("f1 score is:",evaluator.evaluate(pred_test, {evaluator.predictionCol:'prediction'}))
48 print("accuracy is: ",evaluator_acc.evaluate(pred_test, {evaluator_acc.predictionCol:'prediction'}))
49 print("precision is: ",evaluator_pre.evaluate(pred_test, {evaluator_pre.predictionCol:'prediction'}))
50 print("recall is: ",evaluator_recall.evaluate(pred_test, {evaluator_recall.predictionCol:'prediction'}))
51 spark.stop()

```

```

f1 score is: 0.4895136412774065
accuracy is: 0.59
precision is: 0.5080397742235643
recall is: 0.59

```

Figure 3. Evaluation of using Random Forest

```

39 dt = DecisionTreeClassifier(featuresCol='features', labelCol='label', predictionCol='prediction',probabilityCol='prol
40 model = dt.fit(training_set)
41 pred_test = model.transform(test_set)
42
43 evaluator = MulticlassClassificationEvaluator(predictionCol="prediction",metricName='f1')
44 evaluator_acc = MulticlassClassificationEvaluator(predictionCol="prediction", metricName="accuracy")
45 evaluator_pre = MulticlassClassificationEvaluator(predictionCol="prediction", metricName="weightedPrecision")
46 evaluator_recall = MulticlassClassificationEvaluator(predictionCol="prediction", metricName="weightedRecall")
47 print("f1 score is:",evaluator.evaluate(pred_test, {evaluator.predictionCol:'prediction'}))
48 print("accuracy is: ",evaluator_acc.evaluate(pred_test, {evaluator_acc.predictionCol:'prediction'}))
49 print("precision is: ",evaluator_pre.evaluate(pred_test, {evaluator_pre.predictionCol:'prediction'}))
50 print("recall is: ",evaluator_recall.evaluate(pred_test, {evaluator_recall.predictionCol:'prediction'}))
51 spark.stop()

```

```

f1 score is: 0.5950033563907392
accuracy is: 0.61875
precision is: 0.6367461614211923
recall is: 0.61875

```

Figure 4. Evaluation of using Decision Tree

Therefore, the proposed stacking model does not always work well on text classification. The introduction of Support Vector Machine and meta features could exert negative impacts on the stacking model's accuracy. The reason could be as Support Vector Machine is a binary hard-margin classifier, it may bring extra noises to the convergence in the Naive Bayes model.

## 2 Improvement

In this part, we introduce three proper enhancement methods to the stacking model.

### 2.1 Pre-processing

In the pre-processing part, we try to improve the model's performance by removing unnecessary punctuation, transforming the original text to lower case, removing white spaces on both ends of a sentence, removing accent characters, and expanding contractions in English expressions.

The associated code snippets and code result are given below.

```
from pyspark.sql.types import DoubleType,StringType
from pyspark.sql.functions import udf
import re
import unicodedata
from word2number import w2n
from pycontractions import Contractions
import gensim.downloader as api
from sklearn.feature_extraction import text as sktext
stopWords = set(sktext.ENGLISH_STOP_WORDS)
deselect_stop_words = {'no', 'not'}
model = api.load("glove-twitter-25")
cont = Contractions(kv_model=model)
cont.load_models()
def remove_accented_chars(text):
    """remove accented characters from text, e.g. café"""
    text = unicodedata.unidecode(text)
    return text
def expand_contractions(text):
    """expand shortened words, e.g. don't to do not"""
    text = list(cont.expand_texts([text], precise=True))[0]
    return text
def remove_whitespace(text):
    """remove extra whitespaces from text"""
    text = text.strip()
    return " ".join(text.split())
```

Figure 5. Pre-processing Code Snippet 01

```
def pre_processing(text):
    text = remove_whitespace(text)
    text = remove_accented_chars(text)
    text = expand_contractions(text)
    text = text.lower()
    regex = re.compile("[^a-z0-9|'\|^\|_|\\s|^(\d+\\. \d+)|^$(\\-\\-)]")
    text = regex.sub("",text)
    words = []
    for word in text.split():
        if (word[-1]=='.' or ','):
            word = word[:-1]
        if (len(word)>=1 and (word not in stopWords or word in deselect_stop_words)):
            words.append(word)
    return (" ".join(words))
pre_processing_udf = udf(pre_processing,StringType())
```

Figure 5. Pre-processing Code Snippet 02

```

update_col = 'descript'
train_data = train_data.select(*[pre_processing_udf(column).alias(update_col) \
    if column == update_col else column for column in train_data.columns])

test_data = test_data.select(*[pre_processing_udf(column).alias(update_col) \
    if column == update_col else column for column in test_data.columns])

```

Figure 6. Pre-processing Code Snippet 03

```

1 !python3 main_pre.py

[=====] 100.0% 104.8/104.8MB downloaded
/usr/local/lib/python3.6/dist-packages/smart_open/smart_open_lib.py:254: UserWarning:
'See the migration notes for details: %s' % _MIGRATION_NOTES_URL
20/08/04 06:42:20 WARN NativeCodeLoader: Unable to load native-hadoop library for you
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(new
20/08/04 06:44:52 WARN BLAS: Failed to load implementation from: com.github.fommil.ne
20/08/04 06:44:52 WARN BLAS: Failed to load implementation from: com.github.fommil.ne
f1 score is: 0.7777090582671312
accuracy is: 0.78125
precision is: 0.7767293577981651
recall is: 0.78125

```

Figure 7. Pre-processing Code Result

From Fig.7, it could be seen that the F1 score increases from 0.7483 to 0.7778. Though it has a small enhancement, the training time is far longer than the original stacking model. The fitting of *gen\_base\_pred\_pipeline* model consumed 1 hour and 43 minutes. This time consumption is not worth of the enhancement.

## 2.2 Classifier Combination

In this part, we try to enhance the stacking model's performance by using different classifier combinations in the meta feature training. We introduce Random Forest Classifier and Decision Tree Classifier in the combination trials.

The combination results are given below.

### ▼ 1.Naive Bayes + Decision Tree

```

[68] 1 !python3 main2.py

20/08/04 09:17:30 WARN NativeCodeLoader: Unable to load native-hadoop library for you
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(new
[Stage 355:]> (0 + 4) / 5]20/08/04
20/08/04 09:18:27 WARN BLAS: Failed to load implementation from: com.github.fommil.ne
f1 score is: 0.7194182810007532
accuracy is: 0.735
precision is: 0.7305846788574089
recall is: 0.735

```

Figure 8. Naive Bayes & Decision Tree

## 2. SVM+ decision tree

```
[69] 1 |python3 main3.py
20/08/04 09:18:41 WARN NativeCodeLoader: Unable to load native-hadoop library for you
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(new
20/08/04 09:19:02 WARN BLAS: Failed to load implementation from: com.github.fommil.ne
20/08/04 09:19:02 WARN BLAS: Failed to load implementation from: com.github.fommil.ne
f1 score is: 0.7516098096960818
accuracy is: 0.755
precision is: 0.7518607085346216
recall is: 0.7550000000000001
```

Figure 9. Support Vector Machine & Decision Tree

## 3. Decision Tree + RandomForest

```
[70] 1 |python3 main4.py
20/08/04 09:21:52 WARN NativeCodeLoader: Unable to load native-hadoop library for you
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(new
20/08/04 09:23:22 WARN BLAS: Failed to load implementation from: com.github.fommil.ne
20/08/04 09:23:22 WARN BLAS: Failed to load implementation from: com.github.fommil.ne
f1 score is: 0.6447867700303038
accuracy is: 0.6625
precision is: 0.6737425204014637
recall is: 0.6625
```

Figure 10. Decision Tree & Random Forest

## 4. Naive Bayes + RandomForest

```
1 |python3 main5.py
20/08/04 09:38:49 WARN NativeCodeLoader: Unable to load native-hadoop library for you
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(new
(0 + 4) / 5120/04
20/08/04 09:39:45 WARN BLAS: Failed to load implementation from: com.github.fommil.ne
f1 score is: 0.6940817878875687
accuracy is: 0.7025
recall is: 0.7025
```

Figure 11. Naive Bayes & Random Forest

## 5. SVM + RandomForest

```
1 |python3 main6.py
20/08/04 09:24:44 WARN NativeCodeLoader: Unable to load native-hadoop library for you
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(new
20/08/04 09:24:54 WARN BLAS: Failed to load implementation from: com.github.fommil.ne
20/08/04 09:24:54 WARN BLAS: Failed to load implementation from: com.github.fommil.ne
f1 score is: 0.7210851848209047
accuracy is: 0.71625
precision is: 0.7454511247144446
recall is: 0.71625
```

Figure 12. Support Vector Machine & Random Forest

From the results above, it could be seen that introducing the Random Forest Classifier into the combination is not a wise choice, and the best combination

would be “SVM + Decision Tree”, with an F1 Score of 0.7516, though the improvement is trivial.

## 2.3 Group Number

In this section we are trying to improve the performance by setting an optimal group number for the meta feature training. We change the GROUP\_NUMBER in the code:

```
training_set = training_set.withColumn('group', (rand(rseed)*GROUP_NUMBER).cast(IntegerType()))
```

The F1 Score results are given below.

### ▼ Group 3

```
[77] 1 !python3 main7.py
[20/08/04 10:12:11 WARN NativeCodeLoader:
Using Spark's default log4j profile: org.
Setting default log level to "WARN".
To adjust logging level use sc.setLogLev
20/08/04 10:12:24 WARN BLAS: Failed to l
20/08/04 10:12:24 WARN BLAS: Failed to l
0.7395556866549676
```

Figure 13. Group Number 3

### ▼ Group 4

```
[78] 1 !python3 main7.py
[20/08/04 10:21:22 WARN NativeCodeLoader
Using Spark's default log4j profile: org.
Setting default log level to "WARN".
To adjust logging level use sc.setLogLe
20/08/04 10:21:35 WARN BLAS: Failed to l
20/08/04 10:21:35 WARN BLAS: Failed to l
0.7495516561819556
```

Figure 14. Group Number 4

### ▼ Group 6

```
[81] 1 !python3 main7.py
[20/08/04 10:48:24 WARN NativeCodeLoad
Using Spark's default log4j profile: o
Setting default log level to "WARN".
To adjust logging level use sc.setLogL
20/08/04 10:48:38 WARN BLAS: Failed to
20/08/04 10:48:38 WARN BLAS: Failed to
0.7495516561819556
```

Figure 15. Group Number 6

▼ Group 7

```
[82] 1 !python3 main7.py
C: 20/08/04 10:52:20 WARN NativeCodeLoad
Using Spark's default log4j profile: org/
Setting default log level to "WARN".
To adjust logging level use sc.setLogLev
20/08/04 10:52:33 WARN BLAS: Failed to l
20/08/04 10:52:33 WARN BLAS: Failed to l
0.7395556866549676
```

Figure 16. Group Number 7

▼ Group 8

```
[83] 1 !python3 main7.py
C: 20/08/04 10:57:53 WARN NativeCodeLoader:
Using Spark's default log4j profile: org/
Setting default log level to "WARN".
To adjust logging level use sc.setLogLeve
20/08/04 10:58:05 WARN BLAS: Failed to lc
20/08/04 10:58:05 WARN BLAS: Failed to lc
0.7484123982966804
```

Figure 17. Group Number 8

▼ Group 9

```
[84] 1 !python3 main7.py
C: 20/08/04 11:51:48 WARN NativeCodeLoad
Using Spark's default log4j profile: c
Setting default log level to "WARN".
To adjust logging level use sc.setLogLe
20/08/04 11:52:02 WARN BLAS: Failed to t
20/08/04 11:52:02 WARN BLAS: Failed to t
0.7483312619309965
```

Figure 18. Group Number 9

▼ Group 10

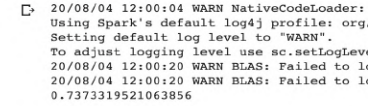
```
[85] 1 !python3 main8.py
C: 20/08/04 11:55:51 WARN NativeCodeLoader
Using Spark's default log4j profile: or
Setting default log level to "WARN".
To adjust logging level use sc.setLogLe
20/08/04 11:56:05 WARN BLAS: Failed to
20/08/04 11:56:05 WARN BLAS: Failed to
0.7342313165608007
```

Figure 19. Group Number 10



▼ Group 11

```
[86] 1 !python3 main9.py
```



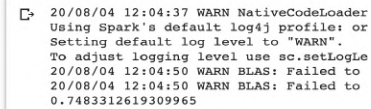
20/08/04 12:00:04 WARN NativeCodeLoader:  
Using Spark's default log4j profile: org.  
Setting default log level to "WARN".  
To adjust logging level use sc.setLogLevel(  
20/08/04 12:00:20 WARN BLAS: Failed to l  
20/08/04 12:00:20 WARN BLAS: Failed to l  
0.7373319521063856

---

Figure 20. Group Number 11

▼ Group 12

```
1 !python3 main10.py
```



20/08/04 12:04:37 WARN NativeCodeLoader:  
Using Spark's default log4j profile: or  
Setting default log level to "WARN".  
To adjust logging level use sc.setLogLe  
20/08/04 12:04:50 WARN BLAS: Failed to  
20/08/04 12:04:50 WARN BLAS: Failed to  
0.7483312619309965

---

Figure 21. Group Number 12

From the results above, it could be seen that changing the group number in the stacking model does not contribute to improve the performance.