

classification_wisc_breast_cancer

November 7, 2020

0.0.1 Classification of Wisconsin Breast Cancer Database

0.0.2 University of California, Santa Barbara

0.0.3 PSTAT 135/235: Big Data Analytics

0.0.4 Last Updated: May 30, 2020

Instructions

In this project, you will work with the Wisconsin Breast Cancer dataset. You will train a logistic regression model to predict the diagnosis. First, you will work through this example. Then you will make modifications and run the code, collecting results at the bottom of the notebook.

The following experiments should be conducted: 1. Three features were used in the model. Build the model using all features. Before training the model, apply scaling to the features using the StandardScaler transformer. Then train the model and compute the accuracy on the test set. Additionally, compute and show the confusion matrix.

2. Repeat step (1), including an intercept
3. Repeat step (1), using randomSplit([0.7, 0.3])
4. Repeat step (2), using randomSplit([0.7, 0.3])
5. Compare and discuss the results of (1) vs (2). Compare and discuss the results of (3) vs (4).

Total Possible Points: 10

```
[1]: # load modules
from pyspark.sql import SparkSession
from pyspark.mllib.classification import LogisticRegressionWithLBFGS, \
    LogisticRegressionModel
from pyspark.mllib.regression import LabeledPoint
from pyspark.ml.feature import VectorAssembler
from pyspark.mllib.linalg import Vectors
from pyspark.sql.functions import col
from pyspark.mllib.evaluation import MulticlassMetrics

import os
```

```
[2]: # param init (you will need to update data_path)
infile = 'wisc_breast_cancer_w_fields.csv'
```

```
spark = SparkSession \
    .builder \
    .appName("Wisc BRCA") \
    .getOrCreate()
```

```
[3]: # read in data
df = spark.read.csv(infile, inferSchema=True, header = True)
```

```
[4]: df.show(2)
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|   id|diagnosis|   f1|   f2|   f3|   f4|   f5|   f6|   f7|   f8|
f9|   f10|  f11|  f12|  f13|  f14|   f15|  f16|  f17|  f18|  f19|
f20|  f21|  f22|  f23|  f24|  f25|  f26|  f27|  f28|  f29|  f30|
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|842302|          M|17.99|10.38|122.8|1001.0| 0.1184| 0.2776|0.3001|
0.1471|0.2419|0.07871| 1.095|0.9053|8.589|153.4|0.006399|0.04904|0.05373|0.01587
|0.03003|0.006193|25.38|17.33|184.6|2019.0|0.1622|0.6656|0.7119|0.2654|0.4601|
0.1189|
|842517|          M|20.57|17.77|132.9|1326.0|0.08474|0.07864|0.0869|0.07017|0.1812
|0.05667|0.5435|0.7339|3.398|74.08|0.005225|0.01308| 0.0186|
0.0134|0.01389|0.003532|24.99|23.41|158.8|1956.0|0.1238|0.1866|0.2416| 0.186|
0.275|0.08902|
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
only showing top 2 rows
```

```
[5]: # use some of the fields as features
assembler = VectorAssembler(inputCols=["f1", "f2", "f3"], outputCol="features")
transformed = assembler.transform(df)
```

```
[6]: # convert to RDD
dataRdd = transformed.select(col("diagnosis"), col("features")).rdd.map(tuple)
```

```
[7]: # look at some data
dataRdd.take(2)
```

```
[7]: [('M', DenseVector([17.99, 10.38, 122.8])),
      ('M', DenseVector([20.57, 17.77, 132.9]))]
```

```
[8]: # map label to binary values, then convert to LabeledPoint
lp = dataRdd.map(lambda row: (1 if row[0]=='M' else 0, Vectors.dense(row[1])))
    ↪ \
        .map(lambda row: LabeledPoint(row[0], row[1]))
```

```
[9]: # look at some data
lp.take(2)
```

```
[9]: [LabeledPoint(1.0, [17.99,10.38,122.8]),
      LabeledPoint(1.0, [20.57,17.77,132.9])]
```

```
[10]: # Split data approximately into training (60%) and test (40%)
training, test = lp.randomSplit([0.6, 0.4], seed=314)
```

```
[11]: # count records in datasets
(training.count(), test.count(), lp.count())
```

```
[11]: (356, 213, 569)
```

```
[12]: (training.count()/lp.count(), test.count()/lp.count(), lp.count()/lp.count())
```

```
[12]: (0.6256590509666081, 0.37434094903339193, 1.0)
```

```
[13]: # Build the model
model = LogisticRegressionWithLBFGS.train(training)
```

```
[14]: # Evaluating the model on test data
labelsAndPreds_te = test.map(lambda p: (p.label, model.predict(p.features)))
accuracy_te = 1.0 * labelsAndPreds_te.filter(lambda pl: pl[0] == pl[1]).count()
    ↪ / test.count()
print('model accuracy (test): {}'.format(accuracy_te))
```

```
model accuracy (test): 0.8732394366197183
```

SOLUTIONS

For parts 1-4, compute and show for the test set: (1) accuracy (2) confusion matrix.
Each part is worth 2 POINTS.

0.1

```
[15]: ## Enter solution for Part 1
from pyspark.ml.feature import StandardScaler

assembler = VectorAssembler(inputCols=["f1", "f2", "f3"], outputCol="features")
transformed = assembler.transform(df)
scaler = StandardScaler(inputCol = "features", outputCol = "scaledFeatures")
scalerModel = scaler.fit(transformed)
```

```

scaledData = scalerModel.transform(transformed)

scaledDataRdd = scaledData.select(col("diagnosis"), col("scaledFeatures")).rdd.
    ↪map(tuple)
lp2 = scaledDataRdd.map(lambda row:(1 if row[0]=='M' else 0, Vectors.
    ↪dense(row[1]))) \
    .map(lambda row: LabeledPoint(row[0], row[1]))

training2, test2 = lp2.randomSplit([0.6, 0.4], seed = 314)
model2 = LogisticRegressionWithLBFGS.train(training2)

labelsAndPreds_te = test2.map(lambda p: (p.label, float(model2.predict(p.
    ↪features))))
accuracy_te = 1.0 * labelsAndPreds_te.filter(lambda pl: pl[0] == pl[1]).count() /
    ↪test2.count()
print('model accuracy (test): {}'.format(accuracy_te))

metrics = MulticlassMetrics(labelsAndPreds_te)
print("Confusion Matrix:\n {}".format(metrics.confusionMatrix().toArray()))

```

```

model accuracy (test): 0.8732394366197183
Confusion Matrix:
[[127.  15.]
 [ 12.  59.]]

```

0.2

```

[16]: ## Enter solution for Part 2
from pyspark.ml.feature import StandardScaler

assembler = VectorAssembler(inputCols=["f1", "f2", "f3"], outputCol="features")
transformed = assembler.transform(df)
scaler = StandardScaler(inputCol = "features", outputCol = "scaledFeatures")
scalerModel = scaler.fit(transformed)
scaledData = scalerModel.transform(transformed)

scaledDataRdd = scaledData.select(col("diagnosis"), col("scaledFeatures")).rdd.
    ↪map(tuple)
lp2 = scaledDataRdd.map(lambda row:(1 if row[0]=='M' else 0, Vectors.
    ↪dense(row[1]))) \
    .map(lambda row: LabeledPoint(row[0], row[1]))

training2, test2 = lp2.randomSplit([0.6, 0.4], seed = 314)
model3 = LogisticRegressionWithLBFGS.train(training2, intercept = True)

```

```

labelsAndPreds_te = test2.map(lambda p: (p.label, float(model3.predict(p.
    ↪features))))
accuracy_te = 1.0 * labelsAndPreds_te.filter(lambda pl: pl[0] == pl[1]).count()
    ↪ / test2.count()
print('model accuracy (test): {}'.format(accuracy_te))

metrics = MulticlassMetrics(labelsAndPreds_te)
print("Confusion Matrix:\n {}".format(metrics.confusionMatrix().toArray()))

```

model accuracy (test): 0.9014084507042254

Confusion Matrix:

```

[[129.  11.]
 [ 10.  63.]]

```

0.3

```

[17]: ## Enter solution for Part 3
from pyspark.ml.feature import StandardScaler

assembler = VectorAssembler(inputCols=["f1", "f2", "f3"], outputCol="features")
transformed = assembler.transform(df)
scaler = StandardScaler(inputCol = "features", outputCol = "scaledFeatures")
scalerModel = scaler.fit(transformed)
scaledData = scalerModel.transform(transformed)

scaledDataRdd = scaledData.select(col("diagnosis"), col("scaledFeatures")).rdd.
    ↪ map(tuple)
lp2 = scaledDataRdd.map(lambda row: (1 if row[0]=='M' else 0, Vectors.
    ↪ dense(row[1]))) \
    .map(lambda row: LabeledPoint(row[0], row[1]))

training2, test2 = lp2.randomSplit([0.7, 0.3], seed = 314)
model2 = LogisticRegressionWithLBFGS.train(training2)

labelsAndPreds_te = test2.map(lambda p: (p.label, float(model2.predict(p.
    ↪features))))
accuracy_te = 1.0 * labelsAndPreds_te.filter(lambda pl: pl[0] == pl[1]).count()
    ↪ / test2.count()
print('model accuracy (test): {}'.format(accuracy_te))

metrics = MulticlassMetrics(labelsAndPreds_te)
print("Confusion Matrix:\n {}".format(metrics.confusionMatrix().toArray()))

```

model accuracy (test): 0.8867924528301887

Confusion Matrix:

```

[[95. 12.]

```

```
[ 6. 46.]
```

0.4

```
[18]: ## Enter solution for Part 4
from pyspark.ml.feature import StandardScaler

assembler = VectorAssembler(inputCols=["f1", "f2", "f3"], outputCol="features")
transformed = assembler.transform(df)
scaler = StandardScaler(inputCol = "features", outputCol = "scaledFeatures")
scalerModel = scaler.fit(transformed)
scaledData = scalerModel.transform(transformed)

scaledDataRdd = scaledData.select(col("diagnosis"), col("scaledFeatures")).rdd.
    ↪map(tuple)
lp2 = scaledDataRdd.map(lambda row:(1 if row[0]=='M' else 0, Vectors.
    ↪dense(row[1]))) \
    .map(lambda row: LabeledPoint(row[0], row[1]))

training2, test2 = lp2.randomSplit([0.7, 0.3], seed = 314)
model3 = LogisticRegressionWithLBFGS.train(training2, intercept = True)

labelsAndPreds_te = test2.map(lambda p: (p.label, float(model3.predict(p.
    ↪features))))
accuracy_te = 1.0 * labelsAndPreds_te.filter(lambda pl: pl[0] == pl[1]).count()
    ↪ / test2.count()
print('model accuracy (test): {}'.format(accuracy_te))

metrics = MulticlassMetrics(labelsAndPreds_te)
print("Confusion Matrix:\n {}".format(metrics.confusionMatrix().toArray()))
```

```
model accuracy (test): 0.9056603773584906
```

```
Confusion Matrix:
```

```
[[96. 10.]
 [ 5. 48.]
```

0.5

Comparing part one and part two, we can see that the accuracy from the first model is 0.8732 and the second model's (including intercept) accuracy is 0.9014. We can see that the accuracy improved after adding the intercept. This stands along with something I learned in PSTAT 131 that when we include an intercept we improve accuracy and adding variance improves precision. Comparing the confusion matrix we can see that TP (True Positive) and the TN (True Negative) is higher in the second model which goes along with the accuracy being higher as well because we are avoiding Type I and II errors. The FP (False Positive) and FN (False Negative) is lower in the second model which just means we are minimizing the type I and type II error.

Comparing part three and part four where we split the data into 70% and 30%, we can see that when we add the intercept(part 4), the accuracy improves just like comparing part one and part two. It showed the exact same pattern when we add an intercept where the TP and TN is higher with the FP and FN being lower in the model with the intercept(part 4). This goes to show that adding an intercept will ultimately improve the models accuracy along with lowering the type I and type II errors.

```
[21]: # Save notebook as PDF document
!jupyter nbconvert --to pdf `pwd`/*.ipynb
```

```
[NbConvertApp] Converting notebook
/home/jovyan/assignments/M4_8/classification_wisc_breast_cancer.ipynb to pdf
[NbConvertApp] Writing 49397 bytes to ./notebook.tex
[NbConvertApp] Building PDF
[NbConvertApp] Running xelatex 3 times: ['xelatex', './notebook.tex', '-quiet']
[NbConvertApp] Running bibtex 1 time: ['bibtex', './notebook']
[NbConvertApp] WARNING | bibtex had problems, most likely because there were no
citations
[NbConvertApp] PDF successfully created
[NbConvertApp] Writing 57538 bytes to
/home/jovyan/assignments/M4_8/classification_wisc_breast_cancer.pdf
```