Solution For Twitter Gender Classification Problem

1. **Initialize Spark Session**

   a) *Set Log-level to ERROR and Spark Session creation*

   ➔ Set Spark logging level to ERROR in order to reduce the log messages.

   ➔ Create and Initialize Spark Session to run in local mode which is utilized through-out the program.

   Code Snippet

```
Logger.getLogger("org").setLevel(Level.ERROR);
Logger.getLogger("akka").setLevel(Level.ERROR);

sparkSession = SparkSession.builder()
                  .appName("TwitterGenderClassification")
                  .master("local[*]")
                  .getOrCreate();
```

2. **Load And Prepare Dataset:**

a) *Multi-line data in description and text colums of the data set resulting into more number of rows than the actual count*

➔ The multi-line data with \r character in the description and text columns of the data set causes an inadvertent increase in the number of rows resulting in inappropriate analysis of data.

➔ This issue is handled by involving the use of the escape and multiLine options when reading the csv file.

Code Snippet

```
originalDS = sparkSession.read()
             .option("header", true)
             .option("multiLine", true)
             .option("mode", "DROPMALFORMED")
             .option("inferschema", true)
             .csv(inputPath);
```

b) *Unwanted columns in the dataset are not required while processing the data*

➔ The unwanted data columns are dropped from the dataset before using the dataset for further processing

Code Snippet

```
String[] unwantedColumns = new String[20];

unwantedColumns[0] = "_unit_id";
unwantedColumns[1] = "_golden";
unwantedColumns[2] = "_last_judgment_at";
unwantedColumns[3] = "profile_yn";
unwantedColumns[4] = "profile_yn:confidence";
unwantedColumns[5] = "created";
unwantedColumns[6] = "fav_number";
unwantedColumns[7] = "gender_gold";
unwantedColumns[8] = "link_color";
unwantedColumns[9] = "name";
unwantedColumns[10] = "profile_yn_gold";
unwantedColumns[11] = "profileimage";
unwantedColumns[12] = "retweet_count";
unwantedColumns[13] = "sidebar_color";
unwantedColumns[14] = "tweet_coord";
unwantedColumns[15] = "tweet_count";
unwantedColumns[16] = "tweet_created";
unwantedColumns[17] = "tweet_id";
unwantedColumns[18] = "tweet_location";
unwantedColumns[19] = "user_timezone";
```

```
for(String col: unwantedColumns) {

      originalDS = originalDS.drop(col);

}
```

c) *Some columns in the dataset contains no values i.e. absence of values or NULL values*

➔ In order to avoid NULL values, the rows containing NULL values are dropped from the dataset

Code Snippet

```
originalDS = originalDS.na().drop();
```

d) *The gender column in the dataset contains invalid values like unknown and some times no values i.e. absence of values*

➔ Records containing no values or invalid values like **unknown** in the **gender** column are filtered out to make the base dataset contains only valid values.

Code Snippet:

```
originalDS = 
originalDS.filter(col("gender").notEqual("unknown")).filter(col("gende
r").notEqual(""));
```

3. **Feature Engineering And Encoding:**

a) *Label and Feature Selection*

➔ The **gender** column is used as **label** and the following columns are used as **features** for the model:

1. **text**
2. **description**
3. **_trusted_judgments**
4. **_unit_state**
5. **gender:confidence**

➔ The values in the columns **gender**, **_trusted_judgment**, **_unit_state** and **gender:confidence** are categorical in nature and hence are indexed using the **StringIndexer** class. The indexed values are captured in the following new columns respectively:

1. **gender --> label**
2. **_trusted_judgment --> _trusted_judgment_index**
3. **_unite_state --> _unit_state_index**
4. **gender:confidence --> gender:confidence_index**

➔ The values in the **text** and **description** columns are processed in the following order:

1. The values of these are tokenized into list of words
2. Stop words are removed from these individual list of words
3. Each word is hashed and a hash value for each word is derived
4. Frequency for each word is calculated

➔ A pipeline is created and each of the above activities are classified and ordered as stages within the pipeline

➔ A model for the pipeline is created and then the dataset is transformed to get a new dataset with all the above processed columns

Code Snippet:

```
labelIndexer = new StringIndexer()
                .setInputCol("gender")
                .setOutputCol("label")
                .fit(originalDS);

trustedJudgementsIndexer = new StringIndexer()
                            .setInputCol("_trusted_judgments")
                            .setOutputCol("_trusted_judgments_index")
                            .fit(originalDS);

unitStateIndexer = new StringIndexer()
                            .setInputCol("_unit_state")
                            .setOutputCol("_unit_state_index")
                            .fit(originalDS);
```

```java
genderConfidenceIndexer = new StringIndexer()
                        .setInputCol("gender:confidence")
                        .setOutputCol("gender:confidence_index")
                        .fit(originalDS);

textTokenizer = new Tokenizer()
                        .setInputCol("text")
                        .setOutputCol("textWords");

removerText = new StopWordsRemover()
                        .setInputCol(textTokenizer.getOutputCol())
                        .setOutputCol("filteredText");

textHTF = new HashingTF()
                        .setNumFeatures(1000)
                        .setInputCol(removerText.getOutputCol())
                        .setOutputCol("numTextFeatures");

textIDF = new IDF()
                        .setInputCol(textHTF.getOutputCol())
                        .setOutputCol("textFeatures");

descriptionTokenizer = new Tokenizer()
                        .setInputCol("description")
                        .setOutputCol("descriptionWords");

removerDescription = new StopWordsRemover()
                  .setInputCol(descriptionTokenizer.getOutputCol())
                  .setOutputCol("filteredDescription");

descriptionHTF = new HashingTF()
                  .setNumFeatures(1000)
                  .setInputCol(removerDescription.getOutputCol())
                  .setOutputCol("numDescriptionFeatures");

descriptionIDF = new IDF()
                  .setInputCol(descriptionHTF.getOutputCol())
                  .setOutputCol("descriptionFeatures");

featurePipeline = new Pipeline()
                  .setStages(new PipelineStage[] {labelIndexer,
                              trustedJudgementsIndexer,
                              unitStateIndexer,
                              genderConfidenceIndexer,
                              textTokenizer,
                              removerText,
                              textHTF,
                              textIDF,
                              descriptionTokenizer,
                              removerDescription,
                              descriptionHTF,
                              descriptionIDF});

featurePipelineModel = featurePipeline.fit(originalDS);

featuresDS = featurePipelineModel.transform(originalDS);
```

b) *One Hot Encoding – Encode the remaining 03 features*

- ➔ A one-hot encoder maps a column of category indices to a column of binary vectors, with at most a single one-value per row that indicates the input category index.
- ➔ The original labels are extracted (**gender**) against the indexed column (**label**)
- ➔ The all the features are not selected in the dataset for further processing

Code Snippet:

```
ohee    = new OneHotEncoderEstimator().setInputCols(new String []
                  {"_trusted_judgments_index",
                   "_unit_state_index",
                   "gender:confidence_index"})
                  .setOutputCols(new String[] {"categoryVec1",
                                               "categoryVec2",
                                               "categoryVec3"});

oheeModel = ohee.fit(featuresDS);

labels = new String[labelIndexer.labels().length];

labels =
Arrays.copyOf(labelIndexer.labels(),labelIndexer.labels().length);

selectedFeaturesDS = oheeModel.transform(featuresDS);
```

4. **Feature Assembling And Selection:**

a) *Feature Assembling*

- ➔ The individual feature vectors generated so far are assembled together using the **VectorAssembler** class
- ➔ All the **05** features are assembled into a vector column named **features**

Code Snippet:

```
assembler = new VectorAssembler()
                        .setInputCols(new String[]{"categoryVec1",
                                                   "categoryVec2",
                                                   "categoryVec3",
                                                   "textFeatures",
                                                "descriptionFeatures"})
                        .setOutputCol("features");

finalFeaturesDS = assembler.transform(selectedFeaturesDS)
                           .select("label","features","gender");
```

b) *Feature Selection*

- ➔ The assembled features (in the shape of a vector) are then processed to retrieve top **1000** feature points which will be used for classification.
- ➔ The **ChiSqSelector** method for top feature selection is used.
- ➔ The selected features are the final features stored in the **finalFeaturesDS** dataset.

Code Snippet:

```
selector = new ChiSqSelector();

selector.setNumTopFeatures(1000);
selector.setLabelCol("label");
selector.setFeaturesCol("features");
selector.setOutputCol("finalFeatures");

finalFeaturesDS = selector.fit(finalFeaturesDS)
                          .transform(finalFeaturesDS)
```

5. **Split The Dataset Into Training And Testing Datasets:**

   a) *A single dataset (**finalFeaturesDS**) to be used for both training the model and then later evaluating the trained model.*

      ➔ The final processed data set is divided into training dataset and testing dataset in the ratio of 80:20 respectively with a seed value of 1.

      Code Snippet:

```
splits = finalFeaturesDS.randomSplit(new double[] { 0.8, 0.2 }, 1L);

trainingDS = splits[0];

testingDS  = splits[1];
```

6. **Model Building:**

a) *Naive Bayes Classification Model*

➔ The **Naive Bayes Model** is created using the training dataset (**trainingDS**)

➔ The predictions are generated for both the training and testing (**testingDS**) dataset

Code Snippet:

```
nb = new NaiveBayes().setLabelCol("label")
                     .setFeaturesCol("finalFeatures");

nbModel = nb.fit(trainingDS);

trainingNBDS = nbModel.transform(trainingDS);

testingNBDS = nbModel.transform(testingDS);

labelConverter = new IndexToString()
                  .setInputCol("prediction")
                  .setOutputCol("predictedLabel")
                  .setLabels(labels);

testingNBDS = labelConverter.transform(testingNBDS);
```

b) *Decision Tree Classification Model*

➔ The **Decision Tree Classifier Model** is created using the training dataset (**trainingDS**)

➔ The **maxDepth** parameter is set to **30**. It is the Maximum depth of a tree. Deeper trees are more expressive (potentially allowing higher accuracy), but they are also more costly to train and are more likely to overfit.

➔ The value of **30** is derived after trying even numbers starting from **02** to **30** as values for the **maxDepth** parameter. The entire code segment was kept in a loop to identify accuracy for the training dataset and testing datasets for the models. A point where the results on training and testing data didn't had a considerable difference was chosen as value for the maxDepth parameter.

➔ The predictions are generated for both the training and testing (**testingDS**) dataset

Code Snippet:

```
dtc    = new DecisionTreeClassifier()
                  .setLabelCol("label")
                  .setFeaturesCol("finalFeatures")
                  .setMaxDepth(30);

dtcModel = dtc.fit(trainingDS);

trainingDTCDS = dtcModel.transform(trainingDS);

testingDTCDS = dtcModel.transform(testingDS);

labelConverter = new IndexToString()
                  .setInputCol("prediction")
                  .setOutputCol("predictedLabel")
                  .setLabels(labels);

testingDTCDS = labelConverter.transform(testingDTCDS);
```

c) *Random Forest Classification Model*

➔ The **Random Forest Classifier Model** is created using the training dataset (**trainingDS**)

➔ The **maxDepth** parameter is set to **30**. It is the Maximum depth of each tree in the forest. Increasing the depth makes the model more expressive and powerful. However, deep trees take longer to train and are also more prone to overfitting. In general, it is acceptable to train deeper trees when using random forests than when using a single decision tree. One tree is more likely to overfit than a random forest (because of the variance reduction from averaging multiple trees in the forest)

➔ The value of **30** is derived after trying even numbers starting from **02** to **30** as values for the **maxDepth** parameter. The entire code segment was kept in a loop to identify accuracy for the training dataset and testing datasets for the models. A point where the results on training and testing data didn’t had a considerable difference was chosen as value for the maxDepth parameter.

➔ The predictions are generated for both the training and testing (**testingDS**) dataset

Code Snippet:

```
rfc = new RandomForestClassifier()
                  .setImpurity("gini")
                  .setMaxDepth(30)
                  .setFeatureSubsetStrategy("auto")
                  .setLabelCol("label")
                  .setFeaturesCol("finalFeatures");

rfcModel = rfc.fit(trainingDS);

trainingRFCDS = rfcModel.transform(trainingDS);

testingRFCDS = rfcModel.transform(testingDS);

labelConverter = new IndexToString()
                  .setInputCol("prediction")
                  .setOutputCol("predictedLabel")
                  .setLabels(labels);

testingRFCDS = labelConverter.transform(testingRFCDS);
```

7. **Evaluation Metrics:**

a) *For all the 03 classification models, a **MulticlassClassificationEvaluator** is prepared and the following metrics are generated for each model*

➔ Accuracy

➔ Weighted Precision

➔ Weighted Recall

➔ F1 Score

➔ Confusion Matrix

b) *Naive Bayes Model Evaluation (As an example)*

Code Snippet:

```
evaluator = new MulticlassClassificationEvaluator()
                .setLabelCol("label")
                .setPredictionCol("prediction");

accuracyNBTrain = evaluator.setMetricName("accuracy")
                            .evaluate(trainingNBDS);

accuracyNBTest = evaluator.setMetricName("accuracy")
                           .evaluate(testingNBDS);

weightedPrecisionNBTest = evaluator.setMetricName("weightedPrecision")
                                    .evaluate(testingNBDS);

weightedRecallNBTest = evaluator.setMetricName("weightedRecall")
                                 .evaluate(testingNBDS);

f1ScorelNBTest = evaluator.setMetricName("f1")
                           .evaluate(testingNBDS);
```

c) *Performance Metrics Output:*

```
Naive Bayes Classifier:
-----------------------

Accuracy = 52 %

Weighted Precision  = 51 %

Weighted Recall = 52 %

F1 Score = 51 %

Confusion Matrix:
+------+--------------+-----+
|gender|predictedLabel|count|
+------+--------------+-----+
|  male|         brand|  228|
|  male|        female|  413|
|female|        female|  643|
|  male|          male|  460|
| brand|         brand|  518|
|female|          male|  298|
| brand|          male|  211|
|female|         brand|  232|
| brand|        female|  137|
+------+--------------+-----+


Decision Tree Classifier:
-------------------------

Accuracy = 45 %

Weighted Precision  = 45 %

Weighted Recall = 45 %

F1 Score = 44 %

Confusion Matrix:
+------+--------------+-----+
|gender|predictedLabel|count|
+------+--------------+-----+
|  male|         brand|  138|
|  male|        female|  575|
|female|        female|  714|
|  male|          male|  388|
| brand|         brand|  306|
|female|          male|  330|
| brand|          male|  312|
|female|         brand|  129|
| brand|        female|  248|
+------+--------------+-----+
Random Forrest Classifier:
--------------------------

Accuracy = 49 %

Weighted Precision  = 49 %

Weighted Recall = 49 %

F1 Score = 48 %

Confusion Matrix:
+------+--------------+-----+
|gender|predictedLabel|count|
+------+--------------+-----+
|  male|         brand|  135|
|  male|        female|  584|
|female|        female|  781|
|  male|          male|  382|
| brand|         brand|  360|
|female|          male|  280|
| brand|          male|  246|
|female|         brand|  112|
| brand|        female|  260|
+------+--------------+-----+
```