Assessment Submission Cover Sheet

Assignment Title	Programming for Big Data – Spark
Module	PROG9813 – Programming for Big Data
Student Name (same as Student Card)	Ethan Moran
Student Number	C19305451
Programme	MSc Computer Science(DS) – TU059
Part-Time/Full-Time	Full-Time
Year of Study (First Year, Second Year, etc)	1^{st}

Academic Integrity for assessment in TU Dublin Programmes

Each student is responsible for knowing and abiding by TU Dublin Academic Regulations and Policies. Any student in breach of these regulation/policies will be subject to action in accordance with the University's procedures for breaches of assessment regulations. Please refer to the General Assessment Regulations at

https://tudublin.libguides.com/c.php?g=674049&p=4794713

https://www.tudublinsu.ie/advice/exams/breachesofregulations/

All students are expected to complete their courses/programmes in compliance with University regulations. No student shall engage in any activity that involves attempting to receive a grade by means other than honest effort, for example:

1. No student shall complete, in part or in total, any examination or assessment for another person.
2. No student shall knowingly allow any examination or assessment to be completed, in part or in total, for themselves by another person.
3. No student shall plagiarise or copy the work of another and submit it as their own work.
4. No student shall falsify any data. Falsification is the invention of data, its alteration, its copying from any other source, or otherwise obtaining it by unfair means, or inventing quotations and/or references.
5. No student shall use aids or devices excluded by the lecturer in undertaking course work or assessments/ examinations.
6. No student shall knowingly procure, provide, or accept any materials that contain questions or answers to any examination or assessment to be given at a subsequent time.
7. No student shall provide their assignments, in part or in total, to any other student in current or future classes of this module/ programme unless authorised to do so by the lecturer.
8. No student shall submit substantially the same material in more than one module/programme without prior authorization.
9. No student shall alter graded assignments or examinations and then resubmit them for regrading, unless specifically authorised to do so by the lecturer.
10. All programming code and documentation, unless correctly referenced, submitted for assessment or existing in the student's computer accounts must be the students' original work or material specifically authorized by the lecturer.
11. Collaborating with other students to develop, complete or correct course work is limited to activities explicitly authorized by the lecturer.
12. For all group assignments, each member of the group is responsible for the academic integrity of the entire submission. Consequently, all group members must satisfy themselves that all elements of their submission adhere to the academic integrity statement points above.

By submitting coursework, either physically or electronically, you are confirming that it is your own work (or, in the case of a group submission, that it is the result of joint work undertaken by members of the group that you represent) and that you have read and understand the University's Regulations and Policies covering Academic Integrity (see General Assessment Regulations).

Coursework may be submitted to an electronic detection system in order to help ascertain if any plagiarised material is present. If you have queries about what constitutes plagiarism, please speak to your lecturer.

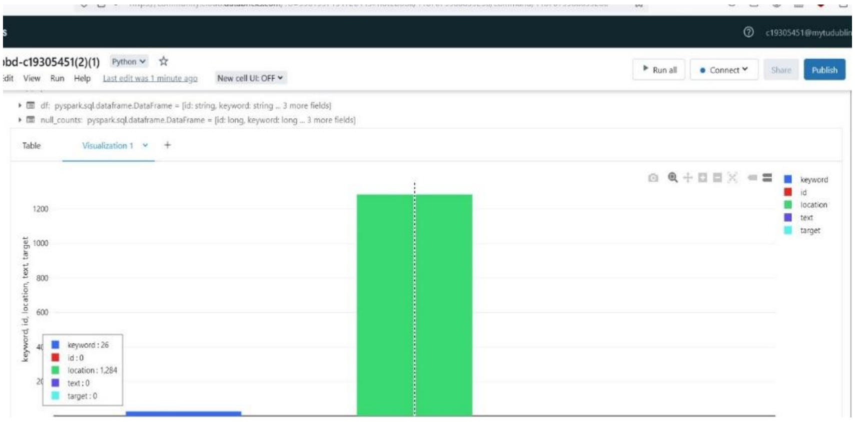
Student Signature	Ethan Moran
Date	21/03/2024

Problem Definition and dataset selected.

The Kaggle competition chosen was Natural Language Processing with disaster tweets. This is a binary classification problem where the model has to determine whether a tweet was a disaster or not. 0 is a non-disaster and 1 is a disaster. The features you could use were location, keyword and the text from the tweet. Social media nowadays and mainly Twitter is used to spread world news. As such Twitter is now often monitored by relief organisations and news channels for example. They may use crawlers to look for certain keywords or even locations(whether valid or not) to report on/help with disasters as they are happening. This can go wrong however as many words can take on different meanings depending on context. One such example given in the Kaggle competition is the word ablaze which could be used metaphorically with a picture for example backing up the context of the word. The reason this dataset was chosen was due to the NLP nature of the project as this is becoming more important in the world of machine learning due to large language models such as Chat-GPT. It was also chosen as this is a project that could genuinely help disaster relief organisations if the accuracy or f1 score, as used in this competition, can get to a good enough level. The link for the competition is https://www.kaggle.com/competitions/nlp-getting-started/data.

Data exploration, feature engineering and data preparation

The data was firstly split in half as cross validation took too long with the original dataset. The data was also under sampled. The 0 cases outnumbered the 1 case by 2233 to 1645. As such, the major class – 0 was sampled to bring it closer to 1645. It ended up being 1648 after, as the sampling is done randomly and will not always be perfectly balanced. For Data exploration, the null counts of each feature were determined to see what could be done with these.



```
df: pyspark.sql.dataframe.DataFrame = [id: string, keyword: string ... 3 more fields]
null_counts: pyspark.sql.dataframe.DataFrame = [id: long, keyword: long ... 3 more fields]
```

keyword : 26
id : 0
location : 1,284
text : 0
target : 0

Figure 1 - Null counts

As can be seen in the figure above, only location and keyword had null values. Missing location and even keywords however could be indications of disasters especially if Twitter cannot contact location services. The more "jokey" locations could also indicate a non-disaster. The lack of keywords may just indicate that keywords could not be discerned; however, the text is still important. As such these had to be left in and were replaced with placeholders. "Missing" for the location and "unknown" for the keywords. This should allow for the models to determine any patterns in the text when there is an unknown keyword/missing location. Then data preparation was started. The URLs were removed using regex(r'https?:\/\/[^\s]+?(?=\s|$)'). This matches any https or http with a string following it. The URLs in the text are simple and always follow this pattern. Then any punctuation was removed from the location, text, and keyword columns, and as it is a social media platform all hashtags and @ symbols were removed also. Non-alphanumeric characters were then removed so that valuable information could be interpreted. The keywords which had spaces contained %20 instead of spaces and as the % signs had been removed, 20's were removed in these columns. Then, the placeholders were added as mentioned above. Before the test-train split, the stopwords are removed. "Like", "amp"(ampersand) and "im" were added to the list as these came up on the word cloud as shown in figure 2 below.

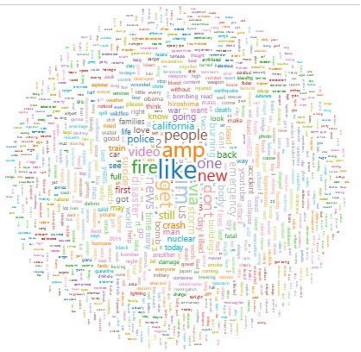


Figure 2 - Before new stopwords

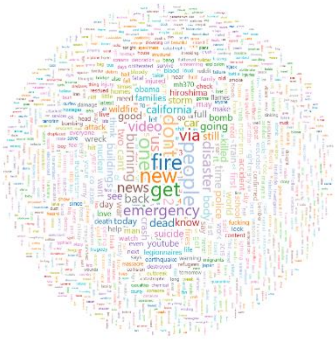


Figure 3 - After new stopwords

This became figure 3. The words were tokenized first however, so they became individual words. Finally, the location was encoded by frequency as there were too many locations to do one hot encoding. This was done before the train-test split as there is no in-built frequency encoding method in PySpark and adding this to the pipeline would as such be too difficult.

After the train-test split, one-hot encoding was performed on the keywords. They were first indexed using the StringIndexer in PySpark and then encoded using the in-built OneHotEncoder. This ensures they are in vector form and can be processed by the machine learning algorithms.

After this, count vectorizer was used to convert the tokenized text into vector form. TF-IDF was considered. This was a mix of hashingTF and DF in PySpark. This however was too resource heavy and as such, the cross validation would not run. Count vectorizer was chosen instead with minimal effects on accuracy, AUC and f1 score. A vector assembler was then used to combine the location, keyword, and text vectors to be used in the models.

Model Creation

Three classification techniques were used in this Logistic Regression, Naïve Bayes and Linear SVC.

Logistic Regression

Logistic regression was used due to its simplicity and effectiveness. It was implemented simply with `lr = LogisticRegression(featuresCol="features", labelCol="target", regParam=1.0)`

The regularization parameter was first set at 1. This is relatively strong but as it is not a very large dataset, it is important to try and combat overfitting from the start. The cross validation however would allow for a more in depth evaluation. Cross validation was the validation method chosen for this as it allowed for a grid to be made to search through the hyperparameters, even if it may be more computationally expensive than train validation split for example. The hyperparameters checked were `paramGrid_lr = ParamGridBuilder().addGrid(lr.regParam, [0.5, 1.0, 1.5]).addGrid(lr.elasticNetParam, [0.0, 0.3, 0.5, 1.0]).build()`

The regularization parameter was checked for 0.5, 1 and 1.5 and elastic net parameter was checked for 0.0, 0.3, 0.5 and 1.0. This checks whether ridge regression(0), lasso regression(1) or a combination of the two ensures the best model. Lasso uses variable selection and regularization (*What is lasso regression?* n.d) whereas ridge regression corrects for multicollinearity (*What is ridge regression?* n.d). The results of both are discussed in the next section Model Evaluation.

Naïve Bayes

Naïve Bayes was used due to its ease to implement alongside the fact that it is regularly used in sentiment analysis and text classification. It was implemented with `nb = NaiveBayes(featuresCol="features", modelType="complement", labelCol="target")`

The base case of Naïve Bayes was used, although there is only one hyperparameter that could be changed in the cross validation, which is smoothing. Smoothing refers to Laplace smoothing that stops the issue of classifications having a zero probability. The cross validation grid looked like `paramGrid_nb = ParamGridBuilder().addGrid(nb.smoothing, [0.0, 0.3, 0.5, 1.0, 2.0]).build()`

The smoothing checked for the values of 0.0, 0.3, 0.5, 1.0 and 2.0, to see whether a lower or higher alpha value resulted in the best model. The results of both are discussed in further detail in the Model Evaluation section.

Linear SVC

Linear SVC is a linear SVM model used in PySpark. It was used due to it being a good algorithm for binary classification. It was implemented with the code `svc = LinearSVC(featuresCol="features", labelCol="target", maxIter=10, regParam=0.1)`. A low regParam was used to start with. Max iterations refers to number of operations performed to find the optimal hyperplane that separates the 1s and 0s. 10 is the base number used here and attempts to find a balance between the model running too long and underfitting. Too low of a value may cause underfitting in this case. The cross validation grid looked like `paramGrid_svc = ParamGridBuilder().addGrid(svc.regParam, [0.0, 0.3, 0.5, 1.0]).addGrid(svc.fitIntercept, [True, False]).build()`

The maxIter was left out of this as the data bricks compute power wouldn't allow for more iterations however the regParam was set from 0.0, 0.3, 0.5 to 1.0. It wasn't set any higher as this was checked and lower values were picked so values like 0.0 and 0.3 were added instead of 1.5 and 2.0. The fit intercept is also checked although unless the intercept definitely passes through the origin and is centred it generally should be True. (*Sklearn.svm.LinearSVC*, n.d)

The results of both are discussed in further detail in the Model Evaluation section.

Model Evaluation

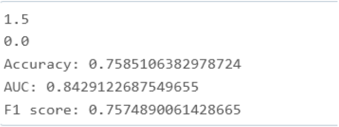
In each of these models, the accuracy, the AUC and the f1 score were found. The accuracy because it is the most commonly used and well-rounded metric, AUC because it is a good indicator of model performance as it shows the true positive/false positive rate against different thresholds(Bhandari, 2024) and the f1 score as it gives a better idea of incorrect predictions compared to overall accuracy.

Logistic Regression

In Logistic Regression, the base linear regression model without any hyperparameter tunings give the results

```
Accuracy: 0.7553191489361702
AUC: 0.8426535013051865
F1 score 0.7545916846122009
```

These are fairly good scores that show that the model has learned something and is at least not randomly guessing, especially since the dataset has been balanced as mentioned in the Data exploration, feature engineering and data preparation section. The hyperparameter tuning was then done using cross validation and the results were

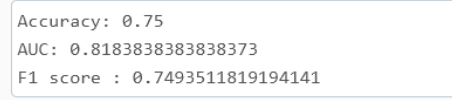


```
1.5
0.0
Accuracy: 0.7585106382978724
AUC: 0.8429122687549655
F1 score: 0.7574890061428665
```

1.5 is the regParam and the 0 the elastic net param. There is a small improvement. If there were more compute resources, perhaps more parameters could be checked.

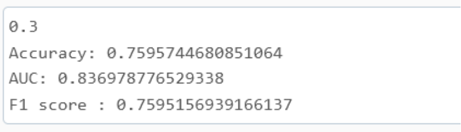
Naïve Bayes

In Naïve Bayes, the naïve bayes with just the default Laplace smoothing gave the results



```
Accuracy: 0.75
AUC: 0.8183838383838373
F1 score : 0.7493511819194141
```

Again, it shows the model is not randomly guessing but is not an improvement on the logistic regression thus far. With LaPlace smoothing of 0.3, the results are higher

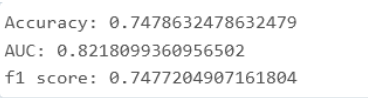


```
0.3
Accuracy: 0.7595744680851064
AUC: 0.836978776529338
F1 score : 0.7595156939166137
```

The accuracy and the f1 score are higher although the AUC is lower than the logistic regression indicating that the logistic regression model generalises better than the naïve bayes one.

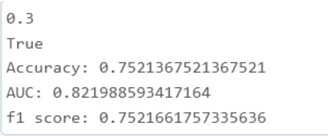
Linear SVC

In the linear SVC model, the SVC model starts with a small regularization parameter which is built upon and then a max iteration that stays the same due to the restricted resources. The results are



```
Accuracy: 0.7478632478632479
AUC: 0.8218099360956502
f1 score: 0.7477204907161804
```

This is a bit worse than both the logistic regression and the naïve bayes model but generalises a bit better than the base Naïve Bayes model. Then with the cross validation, the results are



```
0.3
True
Accuracy: 0.7521367521367521
AUC: 0.821988593417164
f1 score: 0.7521661757335636
```

With regParam at 0.3 and fit intercept as True, the model doesn't outperform either of the last two although it is still promising.

The model for the Kaggle competition that was used was the Naïve Bayes model with the hyperparameter tuning as it has the best f1 score which is what the Kaggle competition was looking for. However overall, the Logistic regression may be the best due to the high AUC score.

Discussion of work, Competition and Features used

The PySpark ML API along with the dataframe API is used throughout the code. The dataframe API allows the mixing of python with the sql code rather than using pure python or sql. This allows for the looping through of columns to make replacing rows in columns for example

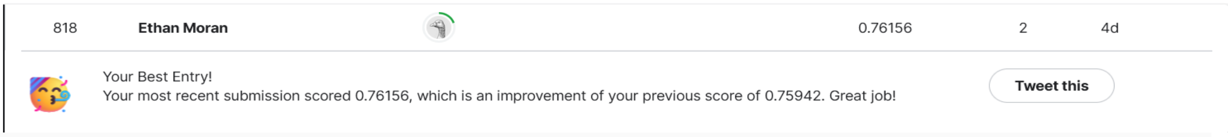
```
for column in columns:
    df_new = df_new.withColumn(column, regexp_replace(df_new[column], "[^a-zA-Z0-9\s]", ''))
df_new.display()
```

It also allows for the use of regex replace as shown above and the explode function that easily allows the counting of words in an array as used in the word cloud. Only features within the PySpark API were used which sometimes limited certain aspects of the exploration such as an .na() in scikit-learn equivalent not existing in PySpark. .isNull() alongside !="" had to be used as below for certain placeholders.

```
df_pl = df_no20.withColumn('keyword', when(col('keyword').isNull(),
'unknown').otherwise(col('keyword')))
```

```
df_pl = df_pl.withColumn('location', when(trim(col('location')).isNull() | (trim(col('location')) ==
""), 'missing').otherwise(col('location')))
```

The Kaggle competition result was an f1 score of 0.76156 for the best Naïve Bayes model. The previous iteration below was logistic regression which was believed to be the best until Complement naïve Bayes was used.



818 Ethan Moran 0.76156 2 4d

Your Best Entry!
Your most recent submission scored 0.76156, which is an improvement of your previous score of 0.75942. Great job!

It is on a leaderboard and is 818/977 at the time of writing. This doesn't seem too impressive, however the model has shown that it can generalise fairly well and there are some perfect scores on the leaderboard. If this assignment was to be done again with more resources, hashingTF and IDF would be used as would Spark NLP to implement some embedding for the NLP side. Cross validation with more specific hyperparameters would be done and neural networks would be looked into as this could learn better and give a higher accuracy, AUC and f1 score.

Code

```
#Setup for ML and SQL functions
from pyspark.ml import Pipeline
from pyspark.ml.feature import RegexTokenizer, StopWordsRemover
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.linalg import Vectors, VectorUDT
from pyspark.ml.feature import StringIndexer, OneHotEncoder, CountVectorizer
from pyspark.ml.classification import LogisticRegression, NaiveBayes, LinearSVC
from pyspark.ml.evaluation import MulticlassClassificationEvaluator, BinaryClassificationEvaluator,
RegressionEvaluator
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
from pyspark.sql.functions import col,isnan, when, count, trim, regexp_replace, explode, udf
from pyspark.sql.types import StringType, StructType, StructField, IntegerType
path = "/FileStore/shared_uploads/c19305451@mytudublin.ie/train_assignment.csv"
#this ensures that multiline text is not read as two texts
df = spark.read.option("multiLine", "true").csv(path, header=True)
#samples half the dataset as it took too long to run. Takes half of the dataset
df = df.sample(withReplacement=False, fraction=0.5, seed=42)
#ensuring the target is an integer instead of text
df = df.withColumn("target", df["target"].cast(IntegerType()))
#checking and displaying null counts
null_counts = df.select([count(when(col(c).isNull() | (col(c) == ""), c)).alias(c) for c in
df.columns])
display(null_counts)
#this gets the number of 0's and 1's
amounts_targets_0 = df.select("target").where(df.target == 0)
amounts_targets_1 = df.select("target").where(df.target == 1)
print(amounts_targets_0.count())
print(amounts_targets_1.count())
#undersampling
#sets the 0 as the larger target(non-disaster) and 1 as the smaller target
```

```
larger_df = df.filter(col("target") == 0)
smaller_df = df.filter(col("target") == 1)
#gets the ratio of 0 to 1 to be able to undersample accordingly
ratio = larger_df.count()/smaller_df.count()
sampled_df = larger_df.sample(False, 1/ratio)
df_us = sampled_df.unionAll(smaller_df)
#shows the new amount of targets(0 and 1)
amounts_targets_0_s= df_us.select("target").where(df_us.target == 0)
amounts_targets_1_s  = df_us.select("target").where(df_us.target == 1)
print(amounts_targets_0_s.count())
print(amounts_targets_1_s.count())
#removing urls using a simple regex that looks for a string that starts with http/https
regex = r'https?:\/\/[^\s]+?(?=\s|$)'
df_new = df_us.withColumn('text', regexp_replace(df_us.text, regex, ''))
df_new.display()
#all the columns that need to be checked
columns = ['text', 'location', 'keyword']
#removing punctuation from location, keywords and text
for column in columns:
    df_new = df_new.withColumn(column, regexp_replace(df_new[column], "[_():';,.!?\\-]", ''))
df_new.display()
#removing hashtags and @s - set as a list to make it easier
to_replace = ["@", "#"]
for column in columns:
    for replace in to_replace:
        df_new = df_new.withColumn(column, regexp_replace(df_new[column], replace, ''))
df_new.display()
columns = ['text', 'location', 'keyword']
#removing non-alphanumeric characters with regex
for column in columns:
   df_new = df_new.withColumn(column, regexp_replace(df_new[column], "[^a-zA-Z0-9\s]", ''))
df_new.display()
#make 20s to an underscore as they represent spaces in keyword
df_no20 = df_new.withColumn('keyword', regexp_replace(df_new.keyword, "20", '_'))
df_no20.display()
#inputting placeholders
df_pl = df_no20.withColumn('keyword', when(col('keyword').isNull(),
'unknown').otherwise(col('keyword')))
df_pl = df_pl.withColumn('location', when(trim(col('location')).isNull() | (trim(col('location')) ==
""), 'missing').otherwise(col('location')))
df_pl.display()
#removing stopwords from text - new stopwords added as these don’t necessarily mean anything but are
added
stopword_list = ["im", "amp", "like"]
stopword_list.extend(StopWordsRemover().getStopWords())
stopword_list = list(set(stopword_list))
```

```
#tokenize first
tokenizer = RegexTokenizer(inputCol="text", outputCol="words", pattern="\\W+", gaps=True)
df_pl = tokenizer.transform(df_pl)
remover = StopWordsRemover(inputCol="words", outputCol="words_filtered", stopWords=stopword_list)
df_pl = remover.transform(df_pl)
#df_pl.show()
#frequency encoding -location(too many for one hot encoding)
frequency_df = df_pl.groupBy("location").count().withColumnRenamed("count", "location_encoded_freq")
df_pl = df_pl.join(frequency_df, on="location", how="left")
df_pl.display()
#splitting dataset
train_df, test_df = df_pl.randomSplit(weights=[0.7,0.3], seed=42)
#there are too many values for location to be one hot encoded
location_distinctcheck = train_df.select("location").distinct()
print(location_distinctcheck.count())
keyword_distinctcheck = train_df.select("keyword").distinct()
print(keyword_distinctcheck.count())
#one hot encoding
#indexing the keywords first
keyw_index = StringIndexer(inputCol="keyword", outputCol="keyword_indexed", handleInvalid='keep')
keyindexmodel = keyw_index.fit(train_df)
indexed_df = keyindexmodel.transform(train_df)
indexed_df.display()
#applying one hot encoding
keyw_encoder = OneHotEncoder(inputCol="keyword_indexed", outputCol="keyword_encoded")
keyw_encoded_df = keyw_encoder.fit(indexed_df).transform(indexed_df)
keyw_encoded_df.display()
#word cloud
#this counts each word that was tokenised before and adds them to a new column
exploded_df = keyw_encoded_df.withColumn("words_filtered", explode(col("words_filtered")))
exploded_df.display()
word_freq = exploded_df.groupBy("words_filtered").count()
#only showing the top 1000 words
list_freq = word_freq.orderBy(col("count").desc()).head(1000)
word_freq_list = spark.createDataFrame(list_freq)
word_freq_list.display()
#count vectorizer
countVect = CountVectorizer(inputCol="words_filtered", outputCol="features_cv")
countVectModel = countVect.fit(keyw_encoded_df)
data = countVectModel.transform(keyw_encoded_df)
data.display()
#Setting up the features and targets
#setting up the encoded location and making it so it can be added to the pipeline.
assembler_loc = VectorAssembler(inputCols=["location_encoded_freq"], outputCol = "location_encoded")
data = assembler_loc.transform(data)
#these are the features now - added to one column for ease of use
```

```python
input = ["location_encoded", "keyword_encoded", "features_cv"]
vecAssembler = VectorAssembler(inputCols=input, outputCol="features")
data_ml = vecAssembler.transform(data)
data_ml.display()
#logistic regression
lr = LogisticRegression(featuresCol="features", labelCol="target", regParam=1.0)
#pipeline
pipeline_lr = Pipeline(stages=[keyw_index, keyw_encoder, countVect, assembler_loc, vecAssembler, lr])
pipelinemodel = pipeline_lr.fit(train_df)
train_predictions = pipelinemodel.transform(train_df)
test_pred = pipelinemodel.transform(test_df)
f1Evaluate = MulticlassClassificationEvaluator(labelCol="target", predictionCol="prediction", metricName="f1")
binaryEvaluate = BinaryClassificationEvaluator(metricName="areaUnderROC", labelCol="target")
accuracyEvaluate = MulticlassClassificationEvaluator(metricName="accuracy", labelCol="target")
print(f"Accuracy: {accuracyEvaluate.evaluate(test_pred)}")
print(f"AUC: {binaryEvaluate.evaluate(test_pred)}")
print(f"F1 score {f1Evaluate.evaluate(test_pred)}")
#cross validation to choose the best model - building a grid to see which hyperparameters give the best model
paramGrid_lr = ParamGridBuilder().addGrid(lr.regParam, [0.5, 1.0, 1.5]).addGrid(lr.elasticNetParam, [0.0, 0.3, 0.5, 1.0]).build()
validator_lr = CrossValidator(estimator=pipeline_lr, estimatorParamMaps=paramGrid_lr, evaluator=binaryEvaluate, numFolds=3, parallelism=2)
#training the best model
train_df.cache() #caches the train data to speed up the process
crossModel = validator_lr.fit(train_df)
train_df.unpersist()
#tests best model
cv_pred = crossModel.transform(test_df)
cv_pred.select("features", "target", "prediction", "probability").display()
#shows accuracy, AUC and f1 score of best model
bestModel = crossModel.bestModel
lrModel = bestModel.stages[-1]
print(lrModel.getRegParam())
print(lrModel.getElasticNetParam())
print(f"Accuracy: {accuracyEvaluate.evaluate(cv_pred)}")
print(f"AUC: {binaryEvaluate.evaluate(cv_pred)}")
print(f"F1 score: {f1Evaluate.evaluate(cv_pred)}")
#naive bayes
nb = NaiveBayes(featuresCol="features", modelType="complement", labelCol="target")
pipeline_nb = Pipeline(stages=[keyw_index, keyw_encoder, countVect, assembler_loc, vecAssembler, nb])
pipelinemodelnb = pipeline_nb.fit(train_df)
train_predictions_nb = pipelinemodelnb.transform(train_df)
test_pred_nb = pipelinemodelnb.transform(test_df)
test_pred_nb.select("features", "target", "prediction", "probability").show()
```

```
print(f"Accuracy: {accuracyEvaluate.evaluate(test_pred_nb)}")
print(f"AUC: {binaryEvaluate.evaluate(test_pred_nb)}")
print(f"F1 score : {f1Evaluate.evaluate(test_pred_nb)}")
#cross validation
paramGrid_nb = ParamGridBuilder().addGrid(nb.smoothing, [0.0, 0.3, 0.5, 1.0, 2.0]).build()
validator_nb = CrossValidator(estimator=pipeline_nb, estimatorParamMaps=paramGrid_nb,
evaluator=binaryEvaluate, numFolds=3, parallelism=2)
train_df.cache()
crossModel_nb = validator_nb.fit(train_df)
train_df.unpersist()
cv_pred_nb = crossModel_nb.transform(test_df)
cv_pred_nb.select("features", "target", "prediction", "probability").display()
bestModel_nb = crossModel_nb.bestModel
nbModel = bestModel_nb.stages[-1]
print(nbModel.getSmoothing())
print(f"Accuracy: {accuracyEvaluate.evaluate(cv_pred_nb)}")
print(f"AUC: {binaryEvaluate.evaluate(cv_pred_nb)}")
print(f"F1 score : {f1Evaluate.evaluate(cv_pred_nb)}")
#SVM - Linear SVC
svc = LinearSVC(featuresCol="features", labelCol="target", maxIter=10, regParam=0.1)
pipeline_svc = Pipeline(stages=[keyw_index, keyw_encoder, countVect, assembler_loc, vecAssembler,
svc])
pipelinemodelsvc = pipeline_svc.fit(train_df)
train_predictions_svc = pipelinemodelsvc.transform(train_df)
test_pred_svc = pipelinemodelsvc.transform(test_df)
test_pred_svc.select("features", "target", "prediction").show()
print(f"Accuracy: {accuracyEvaluate.evaluate(test_pred_svc)}")
print(f"AUC: {binaryEvaluate.evaluate(test_pred_svc)}")
print(f"f1 score: {f1Evaluate.evaluate(test_pred_svc)}")
paramGrid_svc = ParamGridBuilder().addGrid(svc.regParam, [0.0, 0.3, 0.5,
1.0]).addGrid(svc.fitIntercept, [True, False]).build()
validator_svc = CrossValidator(estimator=pipeline_svc, estimatorParamMaps=paramGrid_svc,
evaluator=binaryEvaluate, numFolds=3, parallelism=2)
train_df.cache()
crossModel_svc = validator_svc.fit(train_df)
train_df.unpersist()
cv_pred_svc = crossModel_svc.transform(test_df)
cv_pred_svc.select("features", "target", "prediction").display()
bestModel_svc = crossModel_svc.bestModel
svcModel = bestModel_svc.stages[-1]
print(svcModel.getRegParam())
print(svcModel.getFitIntercept())
print(f"Accuracy: {accuracyEvaluate.evaluate(cv_pred_svc)}")
print(f"AUC: {binaryEvaluate.evaluate(cv_pred_svc)}")
print(f"f1 score: {f1Evaluate.evaluate(cv_pred_svc)}")
#doing the same pre-processing to the kaggle test data
```

```
path_kaggle = "/FileStore/shared_uploads/c19305451@mytudublin.ie/test.csv"
test_kg = spark.read.option("multiLine", "true").csv(path_kaggle, header=True)
#removing urls
regex = r'https?:\/\/[^\s]+?(?=\s|$)'test_kg = test_kg.withColumn('text', regexp_replace(test_kg.text,
regex, ''))
test_kg.display()
columns = ['text', 'location', 'keyword']
#removing punctuation from location, keywords and text
for column in columns:
    test_kg = test_kg.withColumn(column, regexp_replace(test_kg[column], "[_():';,.!?\\-]", ''))
test_kg.display()
#removing hashtags and @s
to_replace = ["@", "#"]
for column in columns:
    for replace in to_replace:
        test_kg = test_kg.withColumn(column, regexp_replace(test_kg[column], replace, ''))
test_kg.display()
columns = ['text', 'location', 'keyword']
for column in columns:
   test_kg = test_kg.withColumn(column, regexp_replace(test_kg[column], "[^a-zA-Z0-9\s]", ''))
test_kg.display()
#make 20s to an underscore as they seem to represent spaces in keyword
kg_no20 = test_kg.withColumn('keyword', regexp_replace(test_kg.keyword, "20", '_'))
kg_no20.display()
#inputting placeholders
kg_pl = kg_no20.withColumn('keyword', when(col('keyword').isNull(),
'unknown').otherwise(col('keyword')))
kg_pl = kg_pl.withColumn('location', when(trim(col('location')).isNull() | (trim(col('location')) ==
""), 'missing').otherwise(col('location')))
df_pl.display()
#removing stopwords from text
stopword_list = list(set(stopword_list))
stopword_list = ["like", "im", "amp"]
stopword_list.extend(StopWordsRemover().getStopWords())
#tokenize first
tokenizer = RegexTokenizer(inputCol="text", outputCol="words", pattern="\\W+", gaps=True)
kg_pl = tokenizer.transform(kg_pl)
remover = StopWordsRemover(inputCol="words", outputCol="words_filtered", stopWords=stopword_list)
kg_pl = remover.transform(kg_pl)
#df_pl.show()
#frequency encoding -location(too many for one hot encoding)
frequency_kg = kg_pl.groupBy("location").count().withColumnRenamed("count", "location_encoded_freq")
kg_pl = kg_pl.join(frequency_kg, on="location", how="left")
kg_pl.display()
#preparing the best model on the cleaned kaggle dataset and displaying it so it can be downloaded –
bestModel does pipeline automatically.
kg_pred = bestModel_nb.transform(kg_pl)
kg_predictions = kg_pred.withColumnRenamed("prediction", "target")
kg_predictions.select("id", "target").display()
```

References

Bhandari, A. (2024, January 8). *Guide to AUC ROC curve in machine learning : What is specificity?*. Analytics Vidhya. https://www.analyticsvidhya.com/blog/2020/06/auc-roc-curve-machine-learning/

Sklearn.svm.LinearSVC. scikit. (n.d.). https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html

What is lasso regression?. IBM. (n.d.-a). https://www.ibm.com/topics/lasso-regression

What is ridge regression?. IBM. (n.d.-b). https://www.ibm.com/topics/ridge-regression