BDAT 1008 Data Collection and Curation — Final Group Project

Network Intrusion Detection System — August 14, 2022

Project Team 2 – Shubham Chawla (200493036), Archit Sinha (200505416), Chintan Piyush Vajani (200508118)

Technical Design Document

Project Overview

Any unauthorized activity on a computer network which could threaten users' privacy or harm the function and infrastructure of the whole network is known as network intrusion. Due to the exponential growth of computer networks and web applications, it has become a critical security requirement to have the ability to protect against the potential threats that can be caused by network attacks.

The main purpose of this system is to ensure that the user is notified when an attack or intrusion takes place so as to have a quick response to malicious traffic.

This report describes the implementation and performance evaluation of a Big Data project whose goal is to use Apache Spark and MLlib in order to perform network attack classification on KDD99 Dataset.

About Dataset - http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html

- Developed by University of California, Irvine 1998-99
- A connection is a sequence of TCP packets starting and ending at some well-defined times, between which data flows to and from a source IP address to a target IP address and use some well-defined application protocol.
- It consists of a single CSV file containing different types of network attacks (Neptune, smurf, DoS etc.) along with normal samples
- Dataset size: **4,898,431 rows and 42 features**
- The derived features are divided into two main categories:
    - Content based features
    - Time based features
- Attacks fall into four main categories:
    - DOS: denial-of-service, e.g. syn flood
    - R2L: unauthorized access from a remote machine, e.g., guessing password
    - U2R: unauthorized access to local superuser (root) privileges, e.g., various ''buffer overflow'' attacks
    - probing: surveillance and other probing, e.g., port scanning.

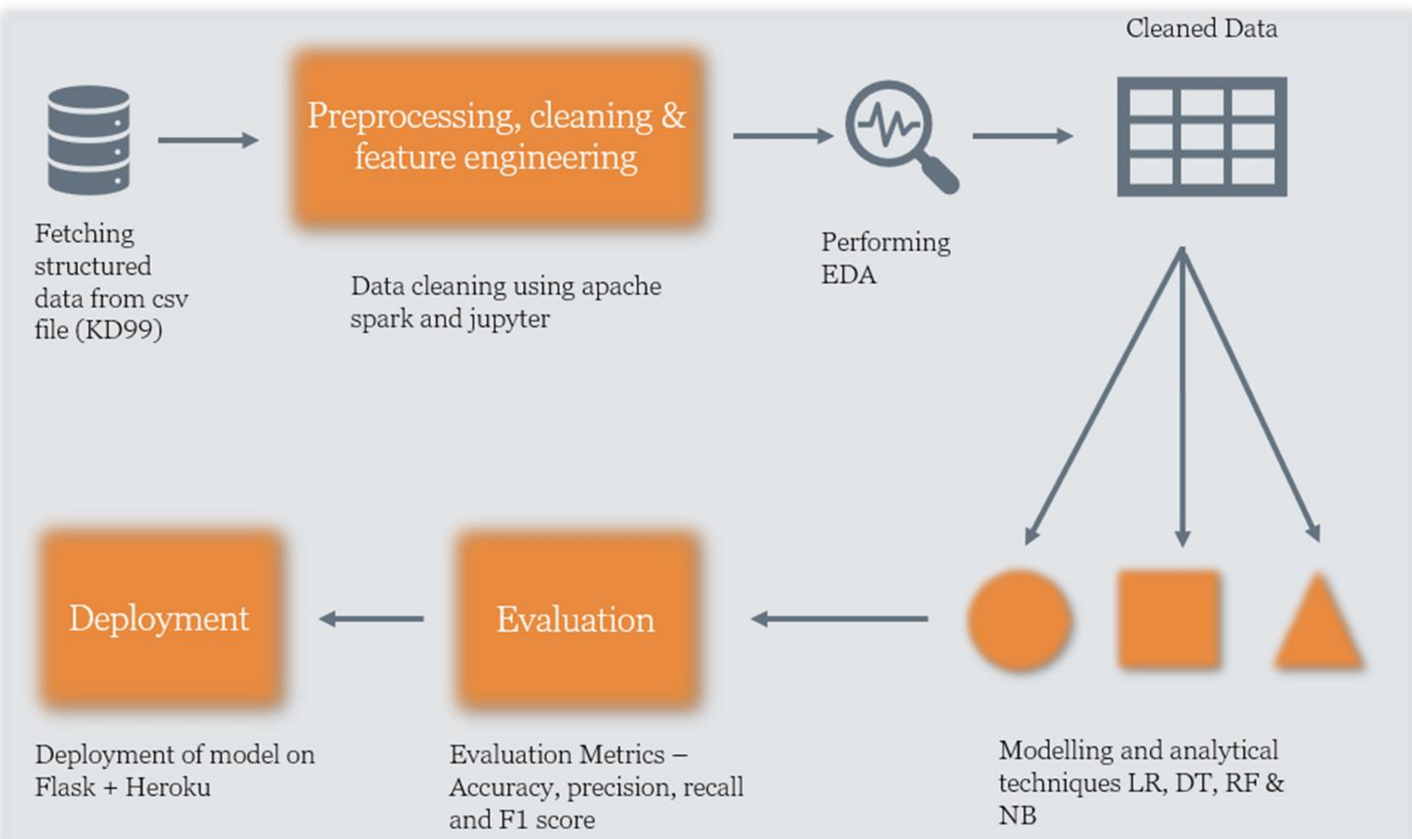
Programming languages Used

- Pyspark
- Python
- Html
- CSS

Tools Used

- Jupyter Notebook
- Apache Spark
- Apache Spark Mllib
- Visual Studio Code
- Flask Framework
- Heroku

Data Flow Diagram



Project Implementation

1. Loading Data and defining attribute names as they were not present in the data csv file.

In [4]:
```
# Defining attribute names as they were not present in the data file
columns = ["duration", "protocol_type", "service", "flag", "src_bytes", "dst_bytes",
           "land", "wrong_fragment", "urgent", "hot", "num_failed_logins", "logged_in",
           "num_compromised", "root_shell", "su_attempted", "num_root", "num_file_creations",
           "num_shells", "num_access_files", "num_outbound_cmds", "is_host_login",
           "is_guest_login", "count", "srv_count", "serror_rate", "srv_serror_rate", "rerror_rate",
           "srv_rerror_rate", "same_srv_rate", "diff_srv_rate", "srv_diff_host_rate", "dst_host_count",
           "dst_host_srv_count", "dst_host_same_srv_rate", "dst_host_diff_srv_rate",
           "dst_host_same_src_port_rate", "dst_host_srv_diff_host_rate", "dst_host_serror_rate",
           "dst_host_srv_serror_rate", "dst_host_rerror_rate", "dst_host_srv_rerror_rate", "label"]

# Reading csv file into spark dataframe
sparkdf = spark.read.csv("kddcup.data.corrected", inferSchema=True, header=False)

# Adding the list of features to our dataset
sparkdf = sparkdf.toDF(*columns)
sparkdf = sparkdf.withColumn("label", regexp_replace("label", "\.", ""))

print("Dataset sizes: {row} rows, {cols} columns".format(row=sparkdf.count(), cols=len(sparkdf.columns)))
```

```
Dataset sizes: 4898431 rows, 42 columns
```

2. Data Preprocessing and Cleaning

```
# Checking nulls in all the columns
from pyspark.sql.functions import col,isnan, when, count
sparkdf.select([count(when(isnan(c) | col(c).isNull(), c)).alias(c) for c in sparkdf.columns]
   ).toPandas()
```

	duration	protocol_type	service	flag	src_bytes	dst_bytes	land	wrong_fragment	urgent	hot	...	dst_host_srv_count	ds
0	0	0	0	0	0	0	0	0	0	0	...	0	0

1 rows × 42 columns

As we can see, there are no null values present in the dataset. So there is no need of cleaning with respect to null or missing values.

No nulls were present in the data.

Using Pyspark SQL to encode attack categories into 5 different categories

0- normal, 1- DOS, 2- Probe, 3 - R2L (remote to user attack), 4 - U2R (User to Root attack)

```
sparkdf.createOrReplaceTempView("network")

# adding label_num column
df_sql = spark.sql("""
SELECT *,
CASE
    WHEN label = 'normal' THEN 0
    WHEN label = 'back' OR label = 'smurf' OR label = 'pod' OR label = 'land' OR label = 'neptune' OR label
    WHEN label = 'satan' OR label = 'ipsweep' OR label = 'nmap' OR label = 'portsweep' THEN 2
    WHEN label = 'guess_passwd' OR label = 'ftpwrite' OR label = 'imap' OR label = 'phf' OR label = 'multih
    ELSE 4
END AS label_num,
CASE
    WHEN label = 'normal' THEN 'normal'
    WHEN label = 'back' OR label = 'smurf' OR label = 'pod' OR label = 'land' OR label = 'neptune' OR label
    WHEN label = 'satan' OR label = 'ipsweep' OR label = 'nmap' OR label = 'portsweep' THEN 'Probe'
    WHEN label = 'guess_passwd' OR label = 'ftpwrite' OR label = 'imap' OR label = 'phf' OR label = 'multih
    ELSE 'U2L'
END AS label_cat
FROM network;
""")
df_sql.select(['label_num','label_cat']).distinct().orderBy('label_num',ascending=True).show()
```

```
+---------+---------+
|label_num|label_cat|
+---------+---------+
|        0|   normal|
|        1|      DOS|
|        2|    Probe|
|        3|      R2L|
|        4|      U2L|
+---------+---------+
```

Checking Count of encoded categories

In [14]:
```
# Checking count of new label categories
df_sql.createOrReplaceTempView("label")

df_sql1 = spark.sql("""
SELECT label_cat, count(label_cat)
FROM label
group by label_cat
order by count(label_cat)
;
""")
#df_sql.select(['label_cat']).distinct().orderBy('label_num',ascending=True).show()
df_sql1.show()
```

Output

```
+---------+----------------+
|label_cat|count(label_cat)|
+---------+----------------+
|      R2L|              98|
|      U2L|            1080|
|    Probe|           41102|
|   normal|          972781|
|      DOS|         3883370|
+---------+----------------+
```

3. Data Exploration – Used Pandas API on spark and Plotly

1. Label Categories Bar Chart

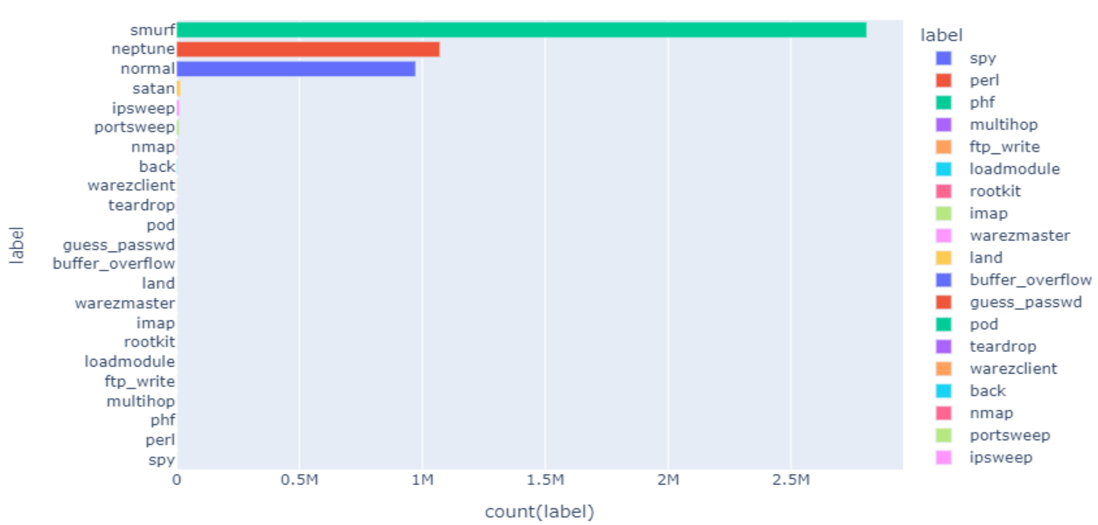
```
In [11]:
# Distribution of label - attack type class
from pyspark.sql import functions as F
df1 = sparkdf.groupby('label').agg(F.count('label')).orderBy('count(label)', ascending= False)
df1.show(23)
```

Output

```
+--------------+------------+
|         label|count(label)|
+--------------+------------+
|         smurf|     2807886|
|       neptune|     1072017|
|        normal|      972781|
|         satan|       15892|
|       ipsweep|       12481|
|     portsweep|       10413|
|          nmap|        2316|
|          back|        2203|
|   warezclient|        1020|
|      teardrop|         979|
|           pod|         264|
|   guess_passwd|         53|
|buffer_overflow|         30|
|          land|          21|
|    warezmaster|         20|
|          imap|          12|
|       rootkit|          10|
|    loadmodule|           9|
|     ftp_write|           8|
|      multihop|           7|
|           phf|           4|
```

```
In [12]:
psdf = df1.to_pandas_on_spark()
psdf.plot.bar(y = 'label', x = 'count(label)', color = 'label').update_yaxes(categoryorder="total
                                                                                   ascending")
```

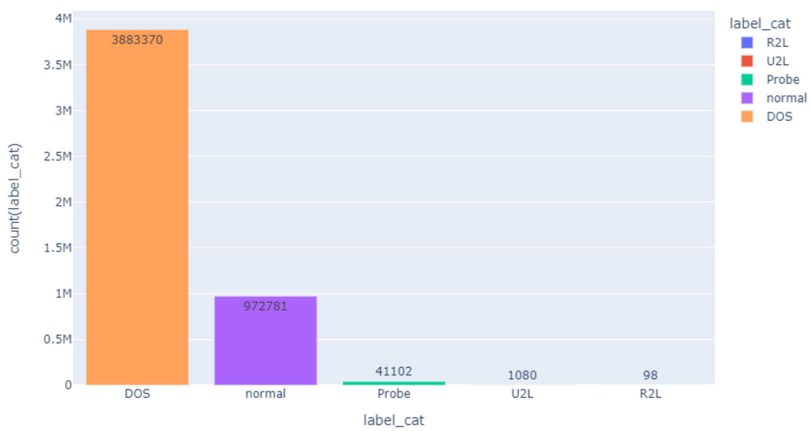
Output



2. Encoded Label Categories Bar Chart

```
In [15]: psdf1 = df_sql1.to_pandas_on_spark()
         psdf1.plot.bar(y = 'count(label_cat)', x = 'label_cat', color = 'label_cat',
                        text = 'count(label_cat)').update_xaxes(categoryorder="total descending")
```

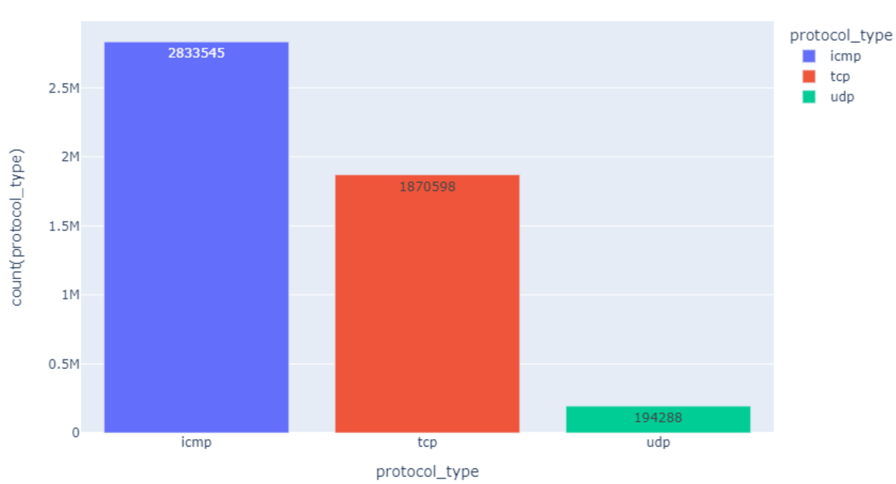
Output



3. Protocol Types Bar Chart

```
In [16]: ### Plotting Bar chart for protocol types

         df3 = df_sql.groupby('protocol_type').agg(F.count('protocol_type')).orderBy('protocol_type',
                                                                                       ascending= True)
         psdf2 = df3.to_pandas_on_spark()
         psdf2.plot.bar(y = 'count(protocol_type)', x = 'protocol_type', color = 'protocol_type',
                        text = 'count(protocol_type)')
```

Output



4. Protocol Types vs Label Bar Chart

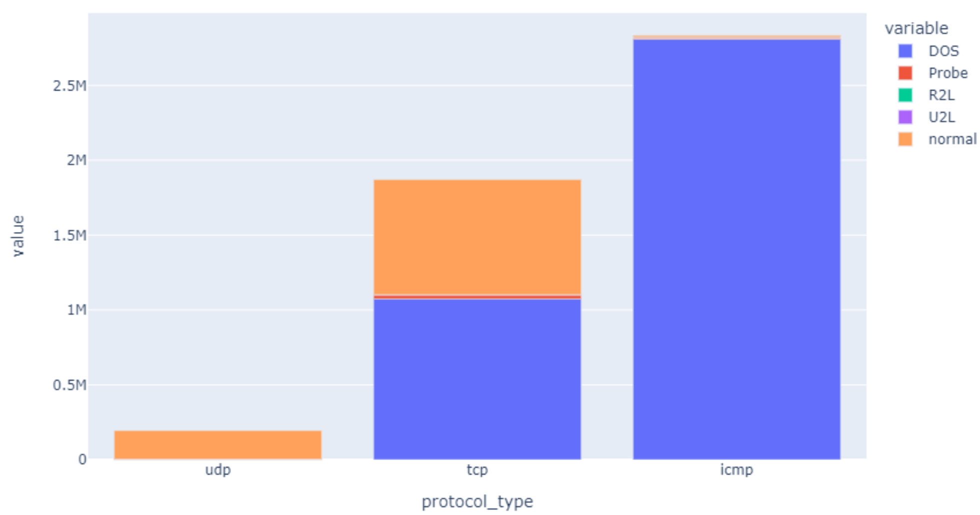
```
In [17]: # Creating pivot table to check relationship between label and protocol type
         df3 = df_sql.groupby('protocol_type').pivot('label_cat').agg(F.count('protocol_type')).
         orderBy('protocol_type', ascending= False)
         psdf2 = df3.to_pandas_on_spark()
         psdf2 = psdf2.fillna(0)
         #psdf2.head()
         df_pd = psdf2.set_index('protocol_type')
         df_pd
```

Output

	DOS	Probe	R2L	U2L	normal
protocol_type					
udp	979	1958	0	3	191348
tcp	1074241	26512	98	1077	768670
icmp	2808150	12632	0	0	12763

```
In [18]: # Plotting protocol type against our target variable
         df_pd.plot.bar(barmode = 'stack')
```

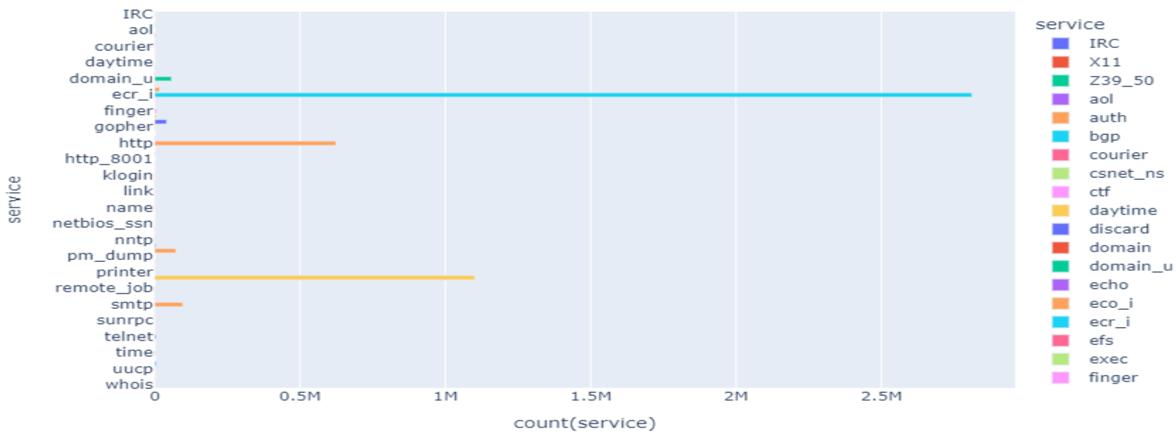
Output:



5. Service Types Bar Chart

```
In [19]: ### Plotting Bar chart for service type

         df3 = df_sql.groupby('service').agg(F.count('service')).orderBy('service', ascending= True)
         psdf2 = df3.to_pandas_on_spark()
         psdf2.plot.bar(y = 'service', x = 'count(service)', color = 'service')
```

Output:

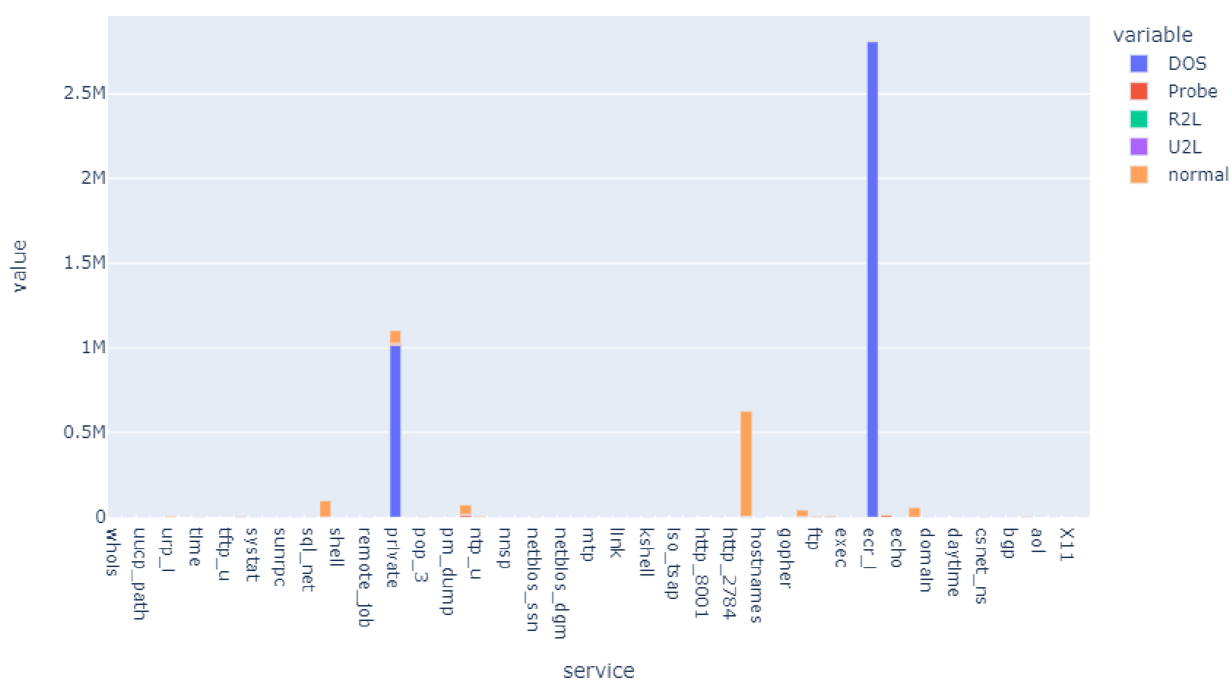


6. Service Types vs Label Bar Chart

In [20]:

```
# Creating pivot table to check relationship between label and service

df3 = df_sql.groupby('service').pivot('label_cat').agg(F.count('service')).orderBy('service', ascending= Fa
psdf2 = df3.to_pandas_on_spark()
psdf2 = psdf2.fillna(0)
#psdf2.head()
df_pd = psdf2.set_index('service')
df_pd.head(3)
```

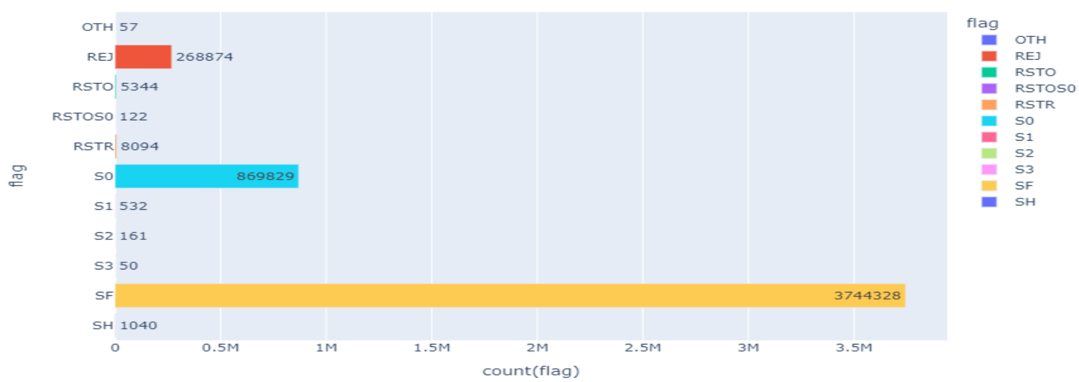
Output:



7. Flag Types Bar Chart

```
In [22]: ### Plotting Bar chart for flag type

         df3 = df_sql.groupby('flag').agg(F.count('flag')).orderBy('flag', ascending= True)
         psdf2 = df3.to_pandas_on_spark()
         psdf2.plot.bar(y = 'flag', x = 'count(flag)', color = 'flag', text = 'count(flag)')
```

Output:



8. Flag Type vs Label Bar Chart

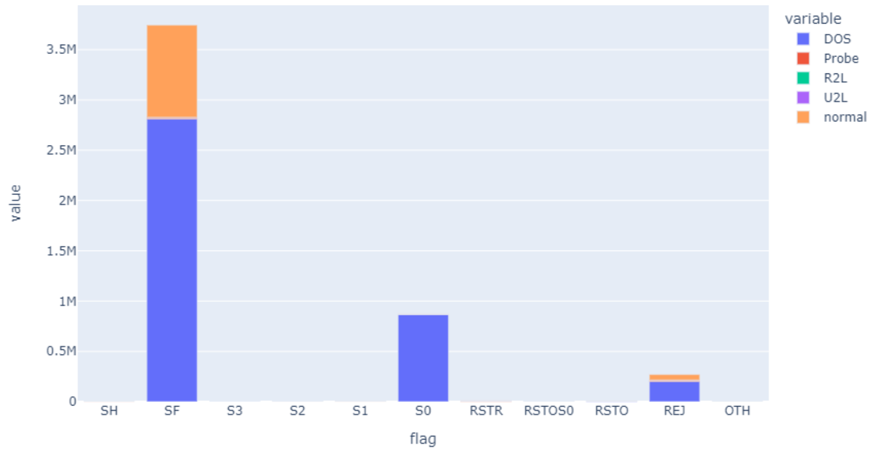
```
In [23]: # Creating pivot table to check relationship between label and flag
         df3 = df_sql.groupby('flag').pivot('label_cat').agg(F.count('flag')).orderBy('flag', ascending= False)
         psdf2 = df3.to_pandas_on_spark()
         psdf2 = psdf2.fillna(0)
         #psdf2.head()
         df_pd = psdf2.set_index('flag')
         df_pd.head(3)
```

Output:

	DOS	Probe	R2L	U2L	normal
flag					
SH	0	1034	4	0	2
SF	2811235	14769	41	1075	917208
S3	0	1	2	1	46

```
In [24]: # Plotting flag against our target variable
         df_pd.plot.bar(barmode = 'stack')
```

Output:



9. Mean table for all numerical variables

```
In [25]: psdf4 = df_sql.to_pandas_on_spark()
         psdf4.groupby('label_num').mean().T
```

Output:

label_num	4	2	1	0	3
duration	585.213889	590.519464	0.000074	217.824724	25.091837
src_bytes	283585.966667	109376.218870	707.446083	1477.846250	157.132653
dst_bytes	966.800000	51350.530655	4.670547	3234.650111	822842.908163
land	0.000000	0.000000	0.000005	0.000007	0.000000
wrong_fragment	0.000000	0.000000	0.000818	0.000000	0.000000
urgent	0.003704	0.000000	0.000000	0.000036	0.000000
hot	7.673148	0.000462	0.001114	0.049535	1.091837
num_failed_logins	0.000926	0.000097	0.000000	0.000099	0.571429
logged_in	0.992593	0.002141	0.000568	0.719268	0.132653
num_compromised	0.059259	0.000170	0.000548	0.038389	0.775510
root_shell	0.024074	0.000000	0.000000	0.000310	0.061224
su_attempted	0.000000	0.000000	0.000000	0.000184	0.010204
num_root	0.039815	0.000170	0.000000	0.064970	1.112245
num_file_creations	0.039815	0.000487	0.000000	0.005887	0.336735
num_shells	0.006481	0.000000	0.000000	0.000363	0.040816
num_access_files	0.003704	0.000000	0.000000	0.005131	0.071429
num_outbound_cmds	0.000000	0.000000	0.000000	0.000000	0.000000
is_host_login	0.000000	0.000000	0.000000	0.000002	0.000000
is_guest_login	0.286111	0.000024	0.000000	0.003882	0.051020
count	1.520370	171.770936	416.668688	8.159029	1.438776
srv_count	1.331481	7.349496	369.633827	10.912790	12.887755
serror_rate	0.004148	0.070904	0.223364	0.001483	0.105510
srv_serror_rate	0.002444	0.073539	0.223360	0.001725	0.095408
rerror_rate	0.002787	0.567238	0.052708	0.055941	0.500000
srv_rerror_rate	0.002778	0.565939	0.052737	0.056206	0.507959
same_srv_rate	0.993620	0.586935	0.743029	0.985257	1.000000
diff_srv_rate	0.009241	0.407238	0.017760	0.018535	0.000000
srv_diff_host_rate	0.011565	0.220565	0.000119	0.132494	0.046735
dst_host_count	82.085185	169.054182	254.867245	148.498415	44.816327
dst_host_srv_count	38.618519	52.952825	187.495878	202.014806	27.724490
dst_host_same_srv_rate	0.738676	0.318464	0.735483	0.844879	0.931327
dst_host_diff_srv_rate	0.021824	0.589753	0.018337	0.056500	0.001837
dst_host_same_src_port_rate	0.658407	0.589580	0.722970	0.134940	0.368163
dst_host_srv_diff_host_rate	0.098639	0.189274	0.000026	0.024340	0.010204
dst_host_serror_rate	0.010352	0.070218	0.223382	0.002039	0.133571
dst_host_srv_serror_rate	0.003537	0.073449	0.223338	0.001050	0.131429
dst_host_rerror_rate	0.005648	0.547672	0.052784	0.057784	0.477857
dst_host_srv_rerror_rate	0.001407	0.564952	0.052707	0.056016	0.475510

Observations:

- Duration of connection for attack is higher than normal except for DOS and R2L.
- Count of outbound commands in an ftp session are 0 for normal and attack connections.
- Wrong fragments in the connection are only present in DOS attack.

10. Correlation Matrix

```
In [28]: # Creating Correlation Matrix
         psdf_corr = df_sql.to_pandas_on_spark()
         corrm=psdf_corr[numerical_variables].corr()
         corrm
```

Output:

	duration	src_bytes	dst_bytes	land	wrong_fragment	urgent	hot	num_failed_logi
duration	1.000000	4.122055e-02	0.020392	-0.000160	-0.001012	3.765465e-03	0.004450	0.007412
src_bytes	0.041221	1.000000e+00	0.000239	-0.000005	-0.000027	-8.677487e-08	0.000782	-0.000007
dst_bytes	0.020392	2.393376e-04	1.000000	-0.000004	-0.000026	1.645208e-04	0.000126	0.000632
land	-0.000160	-4.659182e-06	-0.000004	1.000000	-0.000036	-2.638263e-06	-0.000063	-0.000010
wrong_fragment	-0.001012	-2.714937e-05	-0.000026	-0.000036	1.000000	-1.670585e-05	-0.000402	-0.000066
urgent	0.003765	-8.677487e-08	0.000165	-0.000003	-0.000017	1.000000e+00	0.003591	0.031005
hot	0.004450	7.822053e-04	0.000126	-0.000063	-0.000402	3.590666e-03	1.000000	0.004475
num_failed_logins	0.007412	-6.921906e-06	0.000632	-0.000010	-0.000066	3.100532e-02	0.004475	1.000000
logged_in	-0.020624	1.998519e-04	0.002119	-0.000979	-0.006197	2.534183e-03	0.064579	0.001792
num_compromised	0.027126	4.832481e-06	0.001307	-0.000005	-0.000032	1.767953e-02	0.002688	0.019542
root_shell	0.026378	-5.620801e-06	0.000988	-0.000020	-0.000125	8.908420e-02	0.017916	0.023673
[illegible]	[illegible]	-4.122588e-	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

4. Data Preparation

Building a pipeline to encode the qualitative variables which are of String datatype namely protocol_type, service, flag, and label into variables of datatype Double by using StringIndexer Transformers.

This step is important as Logistic Regression and NaiveBayes classifiers needs to be trained on numerical values.

```
In [26]: # Defining Categorical Variables
         qualitative_variables = ["protocol_type", "service", "flag"]

         # Using String Indexer to encode Categorical variables to Numerical Variables
         indexers = [StringIndexer(inputCol=column, outputCol=column + "_num") for column in qualitative_variables]

         #Creating a pipeline
         pipeline = Pipeline(stages=indexers)
         #Transforming our spark dataframe
         df_sql = pipeline.fit(df_sql).transform(df_sql)

         # Excluding Non-Numeric Attributes
         not_needed = qualitative_variables + ["label", "label_num","label_cat"]
         #print(not_needed)
```

Then we used Spark's Vector Assembler to create a feature vector for each numerical variable as we need to combine all the input columns into a single vector which would essentially act as the input feature for the classifiers.

```
In [29]:
df_assembler = VectorAssembler(inputCols=numerical_variables, outputCol="features")
df_sql = df_assembler.transform(df_sql)
#df_sql.printSchema()
```

Then we created our final dataframe ***ml_df*** containing two columns - **features** and **label_num**. Furthermore, we will randomly split it into the training dataset and testing dataset in 75:25 ratio respectively.

```
In [30]:
# Selecting vectorized features column and label_num
ml_df = df_sql.select(["features","label_num"])
ml_df.printSchema()

train_set, test_set = ml_df.randomSplit([0.75, 0.25], seed=3000)
print("Training dataset count: " + str(train_set.count()))
print("Test dataset count: " + str(test_set.count()))
```

```
root
 |-- features: vector (nullable = true)
 |-- label_num: integer (nullable = false)

Training dataset count: 3675127
Test dataset count: 1223304
```

5. Machine Learning Models

At this point we instantiate the classifiers for each model, we fit it using the training set, we make predictions on the test set and finally we evaluate and print in output the performance by considering accuracy, weighted precision, weighted recall and F1-score as performance metrics.

Defining metrics

```
In [35]:
metrics = ["accuracy", "weightedPrecision", "weightedRecall", "f1"]
```

- **Logistic Regression**

```
In [88]:
# Logistic Regression model
lr = LogisticRegression(maxIter=20, regParam=0.3, elasticNetParam=0.8, featuresCol="features",
                        labelCol="label_num", family="multinomial")

lrlist = []

print("\nLogistic Regression Model Evaluation:")
print("-"*30)
lr_model = lr.fit(train_set)

# make predictions
predictions = lr_model.transform(test_set)
predictions.cache()

evaluator = MulticlassClassificationEvaluator(labelCol="label_num", predictionCol="prediction")

for m in metrics:
    evaluator.setMetricName(m)
    metric = evaluator.evaluate(predictions)
    print("{name} = {value:.2f}".format(name=m, value=metric))
    lrlist.append(metric)

predictions.show(5)
```

➢ Decision Tree

```
In [89]:
# Decision Tree model
dt = DecisionTreeClassifier(labelCol="label_num", featuresCol="features",  maxBins=70)

dtlist = []

print("\nDecision Tree Model Evaluation:")
print("{:-<30}".format(""))
dt_model = dt.fit(train_set)

# make predictions
predictions = dt_model.transform(test_set)
predictions.cache()

evaluator = MulticlassClassificationEvaluator(labelCol="label_num", predictionCol="prediction")

for m in metrics:
    evaluator.setMetricName(m)
    metric = evaluator.evaluate(predictions)
    print("{name} = {value:.2f}".format(name=m, value=metric))
    dtlist.append(metric)

predictions.show(5)
```

➢ Random Forest

```
In [90]:
# Random Forest model
rf = RandomForestClassifier(labelCol="label_num", featuresCol="features", numTrees=20, maxBins=70)

rflist = []

print("\nRandom Forest Classifier Model Evaluation:")
print("{:-<30}".format(""))
rf_model = rf.fit(train_set)

# make predictions
predictions = rf_model.transform(test_set)
predictions.cache()

evaluator = MulticlassClassificationEvaluator(labelCol="label_num", predictionCol="prediction")

for m in metrics:
    evaluator.setMetricName(m)
    metric = evaluator.evaluate(predictions)
    print("{name} = {value:.2f}".format(name=m, value=metric))
    rflist.append(metric)

predictions.show(5)
```

➢ Multinomial Naïve Bayes

```
In [91]:
# Naive Bayes Multinomial
nb = NaiveBayes(labelCol="label_num", featuresCol="features", smoothing=1.0, modelType="multinomial")

nblist = []

print("\nNaive Bayes Multinomial Model Evaluation:")
print("{:-<30}".format(""))
nb_model = nb.fit(train_set)

# make predictions
predictions = nb_model.transform(test_set)
predictions.cache()

evaluator = MulticlassClassificationEvaluator(labelCol="label_num", predictionCol="prediction")

for m in metrics:
    evaluator.setMetricName(m)
    metric = evaluator.evaluate(predictions)
    print("{name} = {value:.2f}".format(name=m, value=metric))
    nblist.append(metric)

predictions.show(5)
```

6. Model Performance Comparison

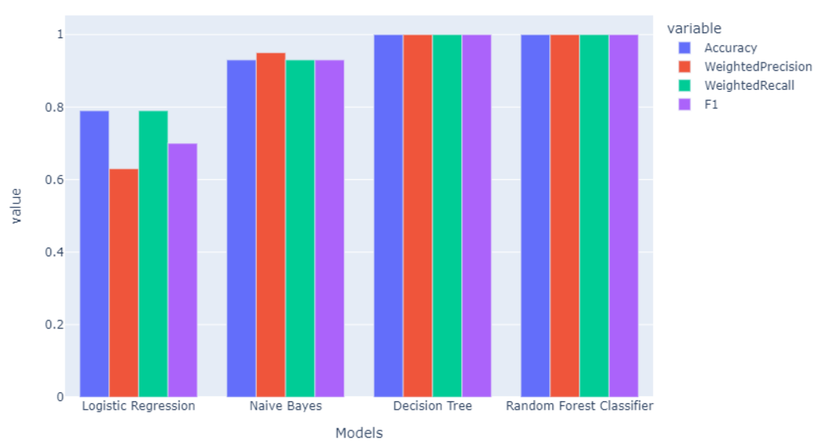
```
In [92]: l = [round(item, 2) for item in lrlist]
         l.insert(0, 'Logistic Regression')
         d = [round(item, 2) for item in dtlist]
         d.insert(0, 'Decision Tree')
         r = [round(item, 2) for item in rflist]
         r.insert(0, 'Random Forest Classifier')
         n = [round(item, 2) for item in nblist]
         n.insert(0, 'Naive Bayes')

         data = [l,d,r,n]

         #giving column names of dataframe
         columns = ["Models","Accuracy", "WeightedPrecision", "WeightedRecall","F1"]
         #creating a dataframe
         dataframe = spark.createDataFrame(data, columns)

         #show data frame
         dataframe.show()
```

Output:

```
+-------------------+--------+-----------------+--------------+----+
|             Models|Accuracy|WeightedPrecision|WeightedRecall|  F1|
+-------------------+--------+-----------------+--------------+----+
| Logistic Regression|    0.79|             0.63|          0.79| 0.7|
|      Decision Tree|     1.0|              1.0|           1.0| 1.0|
|Random Forest Cla...|     1.0|              1.0|           1.0| 1.0|
|        Naive Bayes|    0.93|             0.95|          0.93|0.93|
+-------------------+--------+-----------------+--------------+----+
```

```
In [93]: psdf_results = dataframe.to_pandas_on_spark()
         psdf_results = psdf_results.set_index('Models')
         psdf_results.head()
```

Output:



7. Saving our model

```
In [380]:
#!pip install prophet
from pyspark.sql.functions import *
from pyspark.sql.types import *
from prophet import Prophet
import pickle

import pickle
pkl_path = "model.pkl"
with open(pkl_path, "wb") as f:
    #Saving our decision tree model
    pickle.dump(dt_model, f)

# save the model
print("*** Data Saved ***")

*** Data Saved ***
```

8. Results Summary

Observations

- The best performance was achieved by Decision Tree and Random Forests classifiers which guarantees high performance scores for all the metrics.
- Naïve Bayes multinomial predicts our target variable with 93% accuracy which is not bad.
- As for the other evaluation metrics, Logistic Regression is characterized by quite good results except for the weighted precision which is rather low (63%)