CW_7145

April 21, 2021

0.1 Development of a Machine Learning model with RAPIDS

0.2 Preprocessing

0.2.1 Required Import of Libraries

[1]:
```
#cudf and cuml libraries
import cudf
import cupy as cp
from cuml.preprocessing.model_selection import train_test_split
from cuml.experimental.preprocessing import StandardScaler
from cuml.decomposition import PCA
from cuml.ensemble import RandomForestClassifier
from cuml.linear_model import LogisticRegression
from cuml.svm import SVC
from cuml.neighbors import KNeighborsClassifier

#cuml metric libraries
from cuml.metrics import roc_auc_score,accuracy_score,confusion_matrix


#plotting libraries
import cuxfilter
from cuxfilter import themes, layouts, DataFrame
from cuxfilter.assets.custom_tiles import get_provider, Vendors
from cuxfilter.charts import scatter
import matplotlib.pyplot as plt
%matplotlib inline
from matplotlib.legend_handler import HandlerLine2D
import seaborn as sns

#time libraries
from datetime import datetime
```

Allow cudf to access RAPIDS Memory Manager to fit dataframe

[2]:
```
cudf.set_allocator('managed')
```

0.2.2 Load data into a DataFrame

[3]:
```
df = cudf.read_csv('Partical.csv',header=None)
```

[4]:
```
df.head()
```

[4]:
```
     0         1         2         3         4         5         6         7  \
0  1.0  0.869293 -0.635082  0.225690  0.327470 -0.689993  0.754202 -0.248573
1  1.0  0.907542  0.329147  0.359412  1.497970 -0.313010  1.095531 -0.557525
2  1.0  0.798835  1.470639 -1.635975  0.453773  0.425629  1.104875  1.282322
3  0.0  1.344385 -0.876626  0.935913  1.992050  0.882454  1.786066 -1.646778
4  1.0  1.105009  0.321356  1.522401  0.882808 -1.205349  0.681466 -1.070464

          8         9  …        19        20        21        22        23  \
0 -1.092064  0.000000  … -0.010455 -0.045767  3.101961  1.353760  0.979563
1 -1.588230  2.173076  … -1.138930 -0.000819  0.000000  0.302220  0.833048
2  1.381664  0.000000  …  1.128848  0.900461  0.000000  0.909753  1.108330
3 -0.942383  0.000000  … -0.678379 -1.360356  0.000000  0.946652  1.028704
4 -0.921871  0.000000  … -0.373566  0.113041  0.000000  0.755856  1.361057

         24        25        26        27        28
0  0.978076  0.920005  0.721657  0.988751  0.876678
1  0.985700  0.978098  0.779732  0.992356  0.798343
2  0.985692  0.951331  0.803252  0.865924  0.780118
3  0.998656  0.728281  0.869200  1.026736  0.957904
4  0.986610  0.838085  1.133295  0.872245  0.808487

[5 rows x 29 columns]
```

0.2.3 Description and info about Data

[5]:
```
df.shape
```

[5]: (11000000, 29)

There are 11 million data points with 29 attributes

[6]:
```
df.info()
```

```
<class 'cudf.core.dataframe.DataFrame'>
RangeIndex: 11000000 entries, 0 to 10999999
Data columns (total 29 columns):
 #   Column  Dtype
---  ------  -----
 0   0       float64
 1   1       float64
 2   2       float64
 3   3       float64
```

```
 4   4        float64
 5   5        float64
 6   6        float64
 7   7        float64
 8   8        float64
 9   9        float64
 10  10       float64
 11  11       float64
 12  12       float64
 13  13       float64
 14  14       float64
 15  15       float64
 16  16       float64
 17  17       float64
 18  18       float64
 19  19       float64
 20  20       float64
 21  21       float64
 22  22       float64
 23  23       float64
 24  24       float64
 25  25       float64
 26  26       float64
 27  27       float64
 28  28       float64
dtypes: float64(29)
memory usage: 2.4 GB
```

There are 29 columns. All the columns are of dtype float 64,target variable has to be changed to int and other indepedant variables to float 32 because cuml supports only float 32 in their algorithms

[7]:
```
df.describe().transpose()
```

[7]:
```
         count      mean       std       min       25%       50%       75%  \
0   11000000.0  0.529920  0.499104  0.000000  0.000000  1.000000  1.000000
1   11000000.0  0.991466  0.565378  0.274697  0.590753  0.853371  1.236226
2   11000000.0 -0.000008  1.008827 -2.434976 -0.738322 -0.000054  0.738214
3   11000000.0 -0.000013  1.006346 -1.742508 -0.871931 -0.000241  0.870994
4   11000000.0  0.998536  0.600018  0.000237  0.576816  0.891628  1.293056
5   11000000.0  0.000026  1.006326 -1.743944 -0.871208  0.000213  0.871471
6   11000000.0  0.990915  0.474975  0.137502  0.678993  0.894819  1.170740
7   11000000.0 -0.000020  1.009303 -2.969725 -0.687245 -0.000025  0.687194
8   11000000.0  0.000008  1.005901 -1.741237 -0.868096  0.000058  0.868313
9   11000000.0  0.999969  1.027808  0.000000  0.000000  1.086538  2.173076
10  11000000.0  0.992729  0.499994  0.188981  0.656461  0.890138  1.201875
11  11000000.0 -0.000010  1.009331 -2.913090 -0.694472  0.000060  0.694592
12  11000000.0 -0.000021  1.006154 -1.742372 -0.870179  0.000351  0.869873
13  11000000.0  1.000008  1.049398  0.000000  0.000000  0.000000  2.214872
```

14	11000000.0	0.992259	0.487662	0.263608	0.650853	0.897249	1.221798
15	11000000.0	0.000015	1.008747	-2.729663	-0.699808	0.000173	0.700154
16	11000000.0	0.000004	1.006305	-1.742069	-0.871134	-0.000752	0.871395
17	11000000.0	1.000011	1.193676	0.000000	0.000000	0.000000	2.548224
18	11000000.0	0.986109	0.505778	0.365354	0.617767	0.868233	1.220930
19	11000000.0	-0.000006	1.007694	-2.497265	-0.714190	0.000372	0.714102
20	11000000.0	0.000017	1.006366	-1.742691	-0.871479	-0.000264	0.871605
21	11000000.0	1.000000	1.400209	0.000000	0.000000	0.000000	3.101961
22	11000000.0	1.034290	0.674635	0.075070	0.790610	0.894930	1.024730
23	11000000.0	1.024805	0.380807	0.198676	0.846227	0.950685	1.083493
24	11000000.0	1.050554	0.164576	0.083049	0.985752	0.989780	1.020528
25	11000000.0	1.009742	0.397445	0.132006	0.767573	0.916511	1.142226
26	11000000.0	0.972960	0.525406	0.047862	0.673817	0.873380	1.138439
27	11000000.0	1.033036	0.365256	0.295112	0.819396	0.947345	1.140458
28	11000000.0	0.959812	0.313338	0.330721	0.770390	0.871970	1.059248

	max
0	1.000000
1	12.098914
2	2.434868
3	1.743236
4	15.396821
5	1.743257
6	9.940391
7	2.969674
8	1.741454
9	2.173076
10	11.647081
11	2.913210
12	1.743175
13	2.214872
14	14.708989
15	2.730009
16	1.742884
17	2.548224
18	12.882567
19	2.498009
20	1.743372
21	3.101961
22	40.192368
23	20.372782
24	7.992739
25	14.262439
26	17.762852
27	11.496522
28	8.374498

[8]:
```
df.columns
```

[8]:
```
Index(['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '11', '12',
       '13', '14', '15', '16', '17', '18', '19', '20', '21', '22', '23', '24',
       '25', '26', '27', '28'],
      dtype='object')
```

The column names are from '0' to '28'

0.2.4 checking for missing values

[9]:
```
df.isnull().sum()
```

[9]:
```
0     0
1     0
2     0
3     0
4     0
5     0
6     0
7     0
8     0
9     0
10    0
11    0
12    0
13    0
14    0
15    0
16    0
17    0
18    0
19    0
20    0
21    0
22    0
23    0
24    0
25    0
26    0
27    0
28    0
dtype: uint64
```

There are no missing values in data

Target Value counts

[10]:
```
df['0'].value_counts()
```

[10]:
```
1.0    5829123
0.0    5170877
Name: 0, dtype: int32
```

Since only 0 has values of 0 and 1 from Data Description and also from above, we are assuming this as Target Variable

0.3 Exploratory Data Analysis (EDA)

Load DF into a Cux DataFrame for plotting

[11]:
```
cux_df = cuxfilter.DataFrame.from_dataframe(df)
```

Target Value Distribution

[12]:
```
bar_chart = cuxfilter.charts.bar('0')
d = cux_df.dashboard([bar_chart])
bar_chart.view()
```

[12]:
```
Column(sizing_mode='scale_both', width=400)
    [0] Bokeh(Figure)
    [1] RangeSlider(sizing_mode='scale_width', step=0, width=400)
```

0.3.1 Although Target class is not strictly evenly distributed but it looks close

0.3.2 Correlation Matrix

[13]:
```
df.corr()
```

[13]:

	0	1	2	3	4	5	6	\
0	1.000000	-0.048599	-0.000134	0.000643	-0.099999	-0.000539	0.056908	
1	-0.048599	1.000000	-0.000153	-0.000175	-0.139528	0.000232	0.165798	
2	-0.000134	-0.000153	1.000000	0.000418	-0.000438	0.000161	-0.000396	
3	0.000643	-0.000175	0.000418	1.000000	-0.000012	-0.044518	-0.000135	
4	-0.099999	-0.139528	-0.000438	-0.000012	1.000000	-0.000232	0.199157	
5	-0.000539	0.000232	0.000161	-0.044518	-0.000232	1.000000	0.000118	
6	0.056908	0.165798	-0.000396	-0.000135	0.199157	0.000118	1.000000	
7	-0.000003	-0.000135	0.264797	0.000793	-0.000119	-0.000642	-0.000329	
8	0.000431	-0.000335	-0.000187	-0.167880	-0.000036	-0.154905	-0.000110	
9	-0.009731	-0.006265	0.000275	-0.000178	-0.030368	0.000001	-0.015637	
10	0.021891	0.004612	0.000123	-0.000241	0.039498	0.000399	0.487611	
11	-0.000184	0.000031	0.264615	0.000419	0.000177	-0.000119	0.000027	
12	-0.000734	0.000071	-0.000418	-0.093680	0.000013	-0.064373	-0.000063	
13	-0.049985	-0.005632	-0.000263	0.000027	-0.005196	-0.000313	-0.032107	
14	0.014865	-0.011190	0.000451	-0.000072	0.019677	0.000067	0.267882	
15	-0.000208	0.000571	0.226182	0.000674	-0.000530	-0.000435	-0.000210	
16	-0.000268	0.000149	-0.000340	-0.082511	0.000094	-0.052710	0.000206	

17	-0.023926	0.005832	0.000095	-0.000160	0.010641	0.000337	-0.025106
18	0.037140	-0.019608	0.000150	-0.000109	0.004578	0.000025	0.164582
19	-0.000344	0.000090	0.177698	0.000983	-0.000017	-0.000475	0.000085
20	0.000432	-0.000047	0.000301	-0.065012	-0.000165	-0.038633	-0.000173
21	0.015057	0.000139	-0.000146	-0.000227	0.009673	0.000204	-0.005552
22	0.012852	0.026513	0.000306	0.000358	0.034129	-0.000024	0.186939
23	0.025545	0.017842	-0.000133	0.000176	0.032766	0.000355	0.261443
24	0.010999	0.272327	-0.000272	-0.000788	0.171896	0.000583	0.018275
25	-0.030911	0.132228	0.000061	0.000409	0.280523	0.000060	0.278144
26	-0.152094	0.007636	0.000145	-0.000249	0.025929	0.000662	0.335090
27	-0.065590	0.095841	-0.000011	0.000207	0.213948	0.000427	0.480738
28	-0.123266	0.141168	0.000072	0.000321	0.298656	0.000329	0.450244

	7	8	9	…	19	20	21	22	\
0	-0.000003	0.000431	-0.009731	…	-0.000344	0.000432	0.015057	0.012852	
1	-0.000135	-0.000335	-0.006265	…	0.000090	-0.000047	0.000139	0.026513	
2	0.264797	-0.000187	0.000275	…	0.177698	0.000301	-0.000146	0.000306	
3	0.000793	-0.167880	-0.000178	…	0.000983	-0.065012	-0.000227	0.000358	
4	-0.000119	-0.000036	-0.030368	…	-0.000017	-0.000165	0.009673	0.034129	
5	-0.000642	-0.154905	0.000001	…	-0.000475	-0.038633	0.000204	-0.000024	
6	-0.000329	-0.000110	-0.015637	…	0.000085	-0.000173	-0.005552	0.186939	
7	1.000000	0.000106	-0.000009	…	0.191889	-0.000280	-0.000070	0.000508	
8	0.000106	1.000000	0.000057	…	-0.000052	-0.101777	0.000275	-0.000067	
9	-0.000009	0.000057	1.000000	…	-0.000170	-0.000275	-0.234233	-0.115780	
10	-0.000416	-0.000312	-0.136351	…	-0.000044	-0.000107	-0.027348	0.183869	
11	0.246482	-0.000135	0.000220	…	0.174811	-0.000217	-0.000493	0.000037	
12	0.000434	-0.198034	-0.000757	…	0.000097	-0.070836	0.000095	-0.000281	
13	-0.000105	0.000613	-0.259127	…	0.000152	0.000156	-0.248957	-0.019811	
14	0.000331	0.000065	-0.148108	…	0.000102	0.000198	-0.034229	0.155333	
15	0.230847	-0.000187	-0.000132	…	0.149607	0.000069	-0.000146	-0.000071	
16	-0.000059	-0.133385	0.000277	…	0.000327	-0.064490	-0.001002	0.000074	
17	0.000356	-0.000606	-0.256531	…	-0.000602	0.000016	-0.254475	0.063079	
18	0.000186	0.000179	-0.147732	…	0.000147	0.000362	0.195825	0.106715	
19	0.191889	-0.000052	-0.000170	…	1.000000	-0.000602	0.000110	-0.000555	
20	-0.000280	-0.101777	-0.000275	…	-0.000602	1.000000	-0.000239	-0.000053	
21	-0.000070	0.000275	-0.234233	…	0.000110	-0.000239	1.000000	0.119757	
22	0.000508	-0.000067	-0.115780	…	-0.000555	-0.000053	0.119757	1.000000	
23	0.000547	-0.000127	-0.070605	…	-0.000292	-0.000176	0.050959	0.795835	
24	0.000585	-0.000199	0.000428	…	-0.000028	0.000208	-0.003100	0.011689	
25	0.000233	-0.000553	0.132221	…	0.000223	0.000129	-0.073656	0.104761	
26	0.000027	-0.000424	0.270451	…	0.000062	0.000030	-0.203769	0.012772	
27	0.000528	-0.000167	0.111227	…	0.000180	-0.000056	-0.060376	0.457981	
28	0.000329	-0.000164	0.003050	…	0.000097	-0.000072	-0.000448	0.461672	

	23	24	25	26	27	28
0	0.025545	0.010999	-0.030911	-0.152094	-0.065590	-0.123266
1	0.017842	0.272327	0.132228	0.007636	0.095841	0.141168

```
2  -0.000133 -0.000272  0.000061  0.000145 -0.000011  0.000072
3   0.000176 -0.000788  0.000409 -0.000249  0.000207  0.000321
4   0.032766  0.171896  0.280523  0.025929  0.213948  0.298656
5   0.000355  0.000583  0.000060  0.000662  0.000427  0.000329
6   0.261443  0.018275  0.278144  0.335090  0.480738  0.450244
7   0.000547  0.000585  0.000233  0.000027  0.000528  0.000329
8  -0.000127 -0.000199 -0.000553 -0.000424 -0.000167 -0.000164
9  -0.070605  0.000428  0.132221  0.270451  0.111227  0.003050
10  0.262688  0.001760  0.204045  0.371731  0.432508  0.383743
11 -0.000330 -0.000239  0.000771 -0.000020  0.000151  0.000057
12 -0.000017  0.000122 -0.000024  0.000285 -0.000052 -0.000008
13  0.023675  0.003526  0.042714  0.148366  0.064296  0.038195
14  0.243738 -0.002181  0.125042  0.257692  0.291775  0.277848
15 -0.000264 -0.000227 -0.000059  0.000121  0.000033 -0.000144
16 -0.000217 -0.000019  0.000078  0.000527  0.000347  0.000102
17  0.069390  0.001129 -0.024881 -0.052367 -0.007966  0.023796
18  0.175381 -0.005030  0.062586  0.136933  0.165062  0.170389
19 -0.000292 -0.000028  0.000223  0.000062  0.000180  0.000097
20 -0.000176  0.000208  0.000129  0.000030 -0.000056 -0.000072
21  0.050959 -0.003100 -0.073656 -0.203769 -0.060376 -0.000448
22  0.795835  0.011689  0.104761  0.012772  0.457981  0.461672
23  1.000000  0.010250  0.117861  0.142874  0.613328  0.589181
24  0.010250  1.000000  0.122068  0.000446  0.035578  0.046834
25  0.117861  0.122068  1.000000  0.289727  0.566684  0.546700
26  0.142874  0.000446  0.289727  1.000000  0.556250  0.413663
27  0.613328  0.035578  0.566684  0.556250  1.000000  0.895267
28  0.589181  0.046834  0.546700  0.413663  0.895267  1.000000

[29 rows x 29 columns]
```

0.3.3 Get list of highly correalted columns with correlated values since its hard to check from the large data sets

[14]:
```python
def correlation(dataset, threshold):
    col_names = []
    col_corr = set() # Set of all the names of deleted columns
    corr_matrix = dataset.corr()
    for i in range(len(corr_matrix.columns)):
        for j in range(i):
            if (corr_matrix.iloc[i, j] >= threshold) and (corr_matrix.
↪columns[j] not in col_corr):
                colname = corr_matrix.columns[i] # getting the name of column
                corr_columns = (corr_matrix.columns[i],corr_matrix.
↪columns[j],corr_matrix.iloc[i, j])
                print(corr_columns)
```

[15]:
```
correlation(df, 0.70)
```

```
('23', '22', 0.795835168196029)
('28', '27', 0.8952667066077536)
```

scatter plot for columns '27' and '28'

[16]:
```
scatter_chart = scatter(x='27',y='28', pixel_shade_type="linear",title='Scatter␣
↪plot for columns 27 and 28')

d = cux_df.dashboard([scatter_chart])
scatter_chart.view()
```

[16]:
```
Column(sizing_mode='scale_both', width=800)
    [0] Bokeh(Figure)
```

[17]:
```
scatter_chart = scatter(x='22',y='23', pixel_shade_type="linear",title='Scatter␣
↪plot for columns 22 and 23')
d = cux_df.dashboard([scatter_chart])
scatter_chart.view()
```

[17]:
```
Column(sizing_mode='scale_both', width=800)
    [0] Bokeh(Figure)
```

Although pair(27,28) looks highly correlated with O.89, they don't look much correlated when the value is close to zero. Same goes with (22,23). Hence I am not dropping any variables

0.3.4 Dimensionality Reduction

Because Dimentionality Reduction finds for important basis vectors from data, I am splitting data into train and test to avoid diluting data integrity.

0.3.5 Splitting data into train and test

[18]:
```
X_train, X_test, y_train, y_test = train_test_split(df, '0', train_size=0.8,␣
↪random_state=0)
```

train_size is set to 0.8 to divide data in ratio of 80:20 for training and testing random_state is set for reproducibility

converting variables in X_train and X_test to float32 Also Converting variables in y_train and y_test to int for fitting models in cuml.

[19]:
```
X_train = X_train.astype(cp.float32)
X_test = X_test.astype(cp.float32)
y_train = y_train.astype(cp.int32)
y_test = y_test.astype(cp.int32)
```

```
[20]: print('Training Features Shape:', X_train.shape)
print('Training Labels Shape:', y_train.shape)
print('Testing Features Shape:', X_test.shape)
print('Testing Labels Shape:', y_test.shape)
```

```
Training Features Shape: (8800000, 28)
Training Labels Shape: (8800000,)
Testing Features Shape: (2200000, 28)
Testing Labels Shape: (2200000,)
```

0.3.6 Scaling

To avoid giving importance to attributes with higher values in PCA, standardization is applied on data

```
[21]: scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

0.4 PCA

```
[22]: pca = PCA(n_components = 23, whiten=False)
X_train_pca = pca.fit_transform(X_train_scaled)
print(f'Explained variance\n {pca.explained_variance_ratio_.sum()}')
X_test_pca = pca.transform(X_test_scaled)
```

```
Explained variance
 0.9592921137809753
```

Using 23 Pca components, the data can explain 95.9 percent of variance.

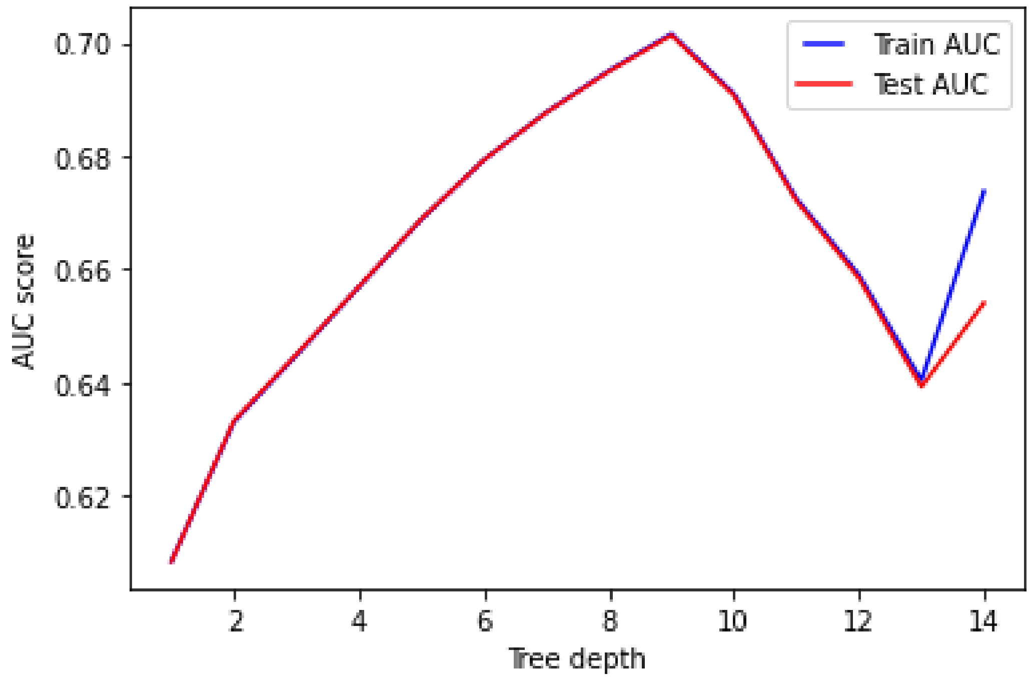
0.4.1 Random Forrest

Depth and no. of estimators are important hyper parameters for Random Forrest, to get the best depth and no. of trees, I will run different parameters in loop.

0.4.2 Hyper-parameters selection

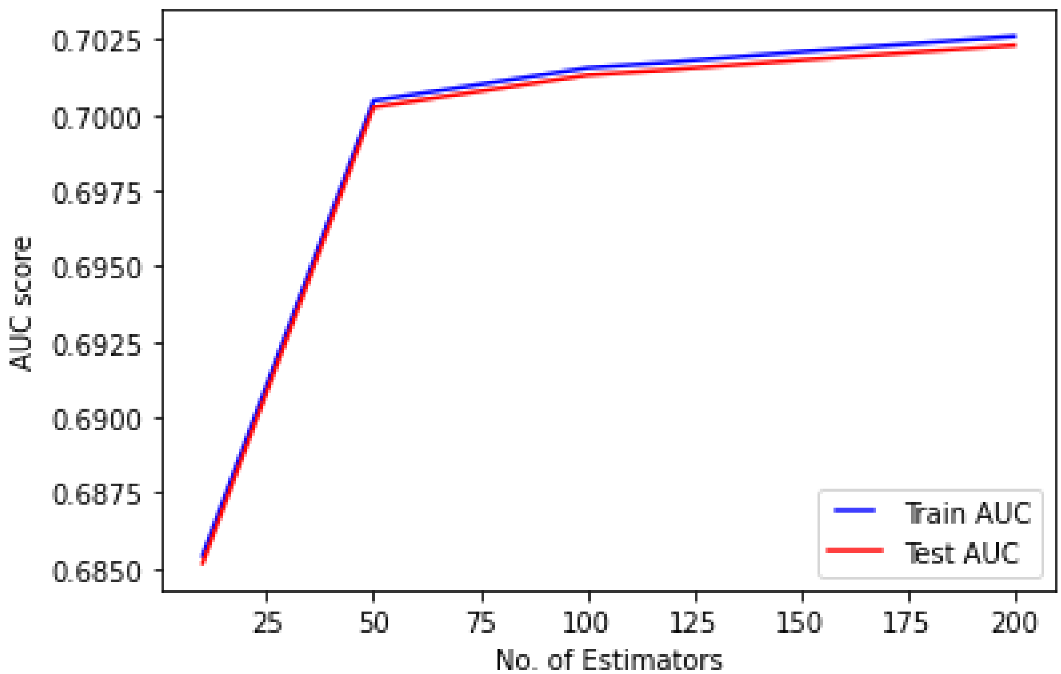
```
[23]: max_depths = range(1,15)
train_results = []
test_results = []
for depth in max_depths:
    rf = RandomForestClassifier(max_depth = depth)
    rf.fit(X_train_pca, y_train)
    train_pred = rf.predict_proba(X_train_pca)
    roc_auc = roc_auc_score(y_train, train_pred[1])
    train_results.append(roc_auc)
    y_pred = rf.predict_proba(X_test_pca)
    roc_auc = roc_auc_score(y_test, y_pred[1])
```

```
    test_results.append(roc_auc)



line1, = plt.plot(max_depths, train_results, 'b', label="Train AUC")
line2, = plt.plot(max_depths, test_results, 'r', label="Test AUC")
plt.legend(handler_map={line1: HandlerLine2D(numpoints=2)})
plt.ylabel('AUC score')
plt.xlabel('Tree depth')
plt.show()
```



[24]:
```
estimators = [10,50,100,120,150,200]
max_depth = 9
train_results = []
test_results = []
for estimator in estimators:
    rf = RandomForestClassifier(max_depth = max_depth, n_estimators = estimator)
    rf.fit(X_train_pca, y_train)
    train_pred = rf.predict_proba(X_train_pca)
    roc_auc = roc_auc_score(y_train, train_pred[1])
    train_results.append(roc_auc)
    y_pred = rf.predict_proba(X_test_pca)
    roc_auc = roc_auc_score(y_test, y_pred[1])
```

```
    test_results.append(roc_auc)


from matplotlib.legend_handler import HandlerLine2D
line1, = plt.plot(estimators, train_results, 'b', label="Train AUC")
line2, = plt.plot(estimators, test_results, 'r', label="Test AUC")
plt.legend(handler_map={line1: HandlerLine2D(numpoints=2)})
plt.ylabel('AUC score')
plt.xlabel('No. of Estimators')
plt.show()
```



Hyperparameters for RandomForest from the above graphs, selection depth as 9 , and number of trees as 100, since it pretty much same after 100

[25]:
```
max_depth = 9
n_trees =100
```

[26]:
```
cuml_rf = RandomForestClassifier(max_depth=max_depth, n_estimators=n_trees)
start=datetime.now()
cuml_rf.fit(X_train_pca, y_train)
cuml_rf_runtime = (datetime.now() - start).seconds
```

Accuracy Score

[27]:
```
cuml_rf_y_train_pred = cuml_rf.predict(X_train_pca)
cuml_rf_y_test_pred = cuml_rf.predict(X_test_pca)
cuml_rf_accuracy_score = accuracy_score(y_test, cuml_rf_y_test_pred)
print("CuML RF Train accuracy:     ", accuracy_score(y_train,␣
↪cuml_rf_y_train_pred))
print("CuML RF Test accuracy:     ", cuml_rf_accuracy_score)
```

```
CuML RF Train accuracy:      0.6484019160270691
CuML RF Test accuracy:      0.6481136083602905
```

confusionmatrix

[28]:
```
cnf_rf = confusion_matrix(y_test,cuml_rf_y_test_pred.astype(cp.int32))
print(cnf_rf)
```

```
        0       1
0  507002  526727
1  247423  918848
```

ROC score

[29]:
```
cuml_rf_y_test_pred = cuml_rf.predict_proba(X_test_pca)
roc_score_rf = roc_auc_score(y_test,cuml_rf_y_test_pred[1])
print('ROC score is: ',roc_score_rf)
```

```
ROC score is:  0.7013164758682251
```

0.4.3 function to get precision,sensitivity, specificity and f1-score

[30]:
```
def get_metrics(cnf):
    TP = cnf.iloc[1,1]
    TN = cnf.iloc[0,0]
    FP = cnf.iloc[1,0]
    FN = cnf.iloc[0,1]
    precision = (TP)/(TP+FP)
    sensitivity = (TP)/(TP+FN)
    specificity = (TN)/(TN+FP)
    f1_score = (2*TP)/((2*TP)+FP+FN)
    return precision,sensitivity,specificity,f1_score
```

precision,sensitivity, specificity and f1_score

[31]:
```
precision,sensitivity,specificity,f1_score = get_metrics(cnf_rf)
print('precision is ',precision)
print('sensitivity is ',sensitivity)
print('specificity is ', specificity)
print('f1_score is ',f1_score)
```

```
precision is  0.7878511941049722
sensitivity is  0.635628037286201
```

```
specificity is  0.6720376445637406
f1_score is  0.7036004419862427
```

[32]:
```
score_dict = {'algorithm':[],
              'implementation':[],
              'Accuracy':[],
              'run_time':[],
              'precison':[],
              'sensitivity':[],
              'roc_auc_score':[],
               'f1_score':[]
              }
```

record results

[33]:
```
score_dict['algorithm'].append('Random Forest')
score_dict['implementation'].append('GPU')
score_dict['Accuracy'].append(cuml_rf_accuracy_score)
score_dict['run_time'].append(cuml_rf_runtime)
score_dict['precison'].append(precision)
score_dict['sensitivity'].append(sensitivity)
score_dict['roc_auc_score'].append(roc_score_rf)
score_dict['f1_score'].append(f1_score)
```

Logistic Regression

[34]:
```
cuml_logreg = LogisticRegression(penalty='l2',fit_intercept=False,max_iter=1000)
start=datetime.now()
cuml_logreg.fit(X_train_pca,y_train)
cuml_log_reg_runtime = (datetime.now()-start).seconds
```

```
[E] [10:09:14.290411] L-BFGS line search failed
```

Accuracy score

[35]:
```
cuml_logreg_y_train_pred = cuml_logreg.predict(X_train_pca)
cuml_logreg_y_test_pred = cuml_logreg.predict(X_test_pca)
cuml_logreg_accuracy_score = accuracy_score(y_test, cuml_logreg_y_test_pred)
print("CuML Log Reg Train accuracy:     ", accuracy_score(y_train,␣
 ↪cuml_logreg_y_train_pred))
print("CuML Log Reg Test accuracy:     ", cuml_logreg_accuracy_score)
```

```
CuML Log Reg Train accuracy:      0.5944775938987732
CuML Log Reg Test accuracy:      0.5944663882255554
```

confusion Matrix

[36]:
```
cnf_logreg = confusion_matrix(y_test,cuml_logreg_y_test_pred.astype(cp.int32))
print(cnf_logreg)
```

```
        0       1
0  620034  413695
1  478479  687792
```

0.4.4 roc_auc_score

[37]:
```
cuml_logreg_y_test_pred = cuml_logreg.predict_proba(X_test_pca)
roc_score_logreg = roc_auc_score(y_test,cuml_logreg_y_test_pred[1])
print('ROC score is: ',roc_score_logreg)
```

```
ROC score is:  0.6393469572067261
```

precison,sensitivity and f1_score

[38]:
```
precision,sensitivity,specificity,f1_score = get_metrics(cnf_logreg)
print('precision is ',precision)
print('sensitivity is ',sensitivity)
print('specificity is ', specificity)
print('f1_score is ',f1_score)
```

```
precision is  0.5897360047536122
sensitivity is  0.6244213504108537
specificity is  0.5644302798419317
f1_score is  0.6065832421272463
```

record results

[39]:
```
score_dict['algorithm'].append('Logistic')
score_dict['implementation'].append('GPU')
score_dict['Accuracy'].append(cuml_logreg_accuracy_score)
score_dict['run_time'].append(cuml_log_reg_runtime)
score_dict['precison'].append(precision)
score_dict['sensitivity'].append(sensitivity)
score_dict['roc_auc_score'].append(roc_score_logreg)
score_dict['f1_score'].append(f1_score)
```

K-Nearest-Neighbors

[40]:
```
cuml_knn = KNeighborsClassifier(n_neighbors=7)
cuml_knn.fit(X_train_pca, y_train)
```

[40]:
```
KNeighborsClassifier(weights='uniform')
```

Accuracy Score

[41]:
```
start = datetime.now()
cuml_knn_y_test_pred = cuml_knn.predict(X_test_pca)
cuml_knn_runtime = (datetime.now()-start).seconds
cuml_knn_accuracy_score = accuracy_score(y_test, cuml_knn_y_test_pred)
```

```
#print("CuML KNN Train accuracy:     ", accuracy_score(y_train,␣
↪cuml_knn_y_train_pred))
print("CuML KNN Test accuracy:     ", cuml_rf_accuracy_score)
```

```
CuML KNN Test accuracy:      0.6481136083602905
```

confusion matrix

```
[42]: cnf_knn = confusion_matrix(y_test,cuml_knn_y_test_pred.astype(cp.int32))
print(cnf_knn)
```

```
        0       1
0  636015  397714
1  307998  858273
```

ROC score

```
[43]: cuml_knn_y_test_pred = cuml_knn.predict_proba(X_test_pca)
roc_score_knn = roc_auc_score(y_test,cuml_knn_y_test_pred[1])
print('ROC score is: ',roc_score_knn)
```

```
ROC score is:  0.7356789708137512
```

precision, sensitivity and f1__score

```
[44]: precision,sensitivity,specificity,f1_score = get_metrics(cnf_knn)
print('precision is ',precision)
print('sensitivity is ',sensitivity)
print('specificity is ', specificity)
print('f1_score is ',f1_score)
```

```
precision is  0.7359121507779924
sensitivity is  0.6833454486391977
specificity is  0.6737354252536777
f1_score is  0.7086553125224481
```

record results

```
[45]: score_dict['algorithm'].append('KNN')
score_dict['implementation'].append('GPU')
score_dict['Accuracy'].append(cuml_knn_accuracy_score)
score_dict['run_time'].append(cuml_knn_runtime)
score_dict['precison'].append(precision)
score_dict['sensitivity'].append(sensitivity)
score_dict['roc_auc_score'].append(roc_score_knn)
score_dict['f1_score'].append(f1_score)
```

Running same algorithms on CPU with sklearn

import sklearn libraries

[46]:
```python
#pandas
import pandas as pd

#Machine Learning learning libraries
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier

#Machine Learning metrics
from sklearn.metrics import confusion_matrix,accuracy_score,roc_auc_score
```

Converting data to pandas for fitting on sklearn

[47]:
```python
X_train_pd = X_train_pca.to_pandas()
y_train_pd = y_train.to_pandas()
X_test_pd = X_test_pca.to_pandas()
y_test_pd = y_test.to_pandas()
```

Randomforest with same parameters as above on CPU

[48]:
```python
sklearn_rf = RandomForestClassifier(max_depth=max_depth, n_estimators=n_trees)
start=datetime.now()
sklearn_rf.fit(X_train_pca.to_pandas(), y_train.to_pandas())
sklearn_rf_runtime = (datetime.now() - start).seconds
```

Accuracy scores

[49]:
```python
sklearn_rf_y_train_pred = sklearn_rf.predict(X_train_pd)
sklearn_rf_y_test_pred = sklearn_rf.predict(X_test_pd)
rf_accuracy_score = accuracy_score(y_test_pd, sklearn_rf_y_test_pred)
print("SKLEARN RF Train accuracy:     ", accuracy_score(y_train_pd,␣
 ↪sklearn_rf_y_train_pred))
print("sklearn RF Test accuracy:     ", rf_accuracy_score)
```

```
SKLEARN RF Train accuracy:      0.6531452272727273
sklearn RF Test accuracy:      0.6518759090909091
```

confusion matrix

[50]:
```python
cnf_rf_sk = confusion_matrix(y_test_pd,sklearn_rf_y_test_pred)
print(cnf_rf_sk)
```

```
[[502813 530916]
 [234957 931314]]
```

precision, sensitivity and f1_score

[51]:
```python
precision,sensitivity,specificity,f1_score = get_metrics(pd.
 ↪DataFrame(cnf_rf_sk))
```

```
print('precision is ',precision)
print('sensitivity is ',sensitivity)
print('specificity is ', specificity)
print('f1_score is ',f1_score)
```

```
precision is  0.798539961981392
sensitivity is  0.6369134814632444
specificity is  0.6815308293912737
f1_score is  0.7086274648554443
```

ROC score

```
[52]: sk_rf_y_test_pred = sklearn_rf.predict_proba(X_test_pd)
sk_rf_auc_roc = roc_auc_score(y_test_pd,sk_rf_y_test_pred[:,1])
print('ROC Score is ',sk_rf_auc_roc)
```

```
ROC Score is  0.7105721661200627
```

record results

```
[53]: score_dict['algorithm'].append('Random Forest')
score_dict['implementation'].append('CPU')
score_dict['Accuracy'].append(rf_accuracy_score)
score_dict['run_time'].append(sklearn_rf_runtime)
score_dict['precison'].append(precision)
score_dict['sensitivity'].append(sensitivity)
score_dict['roc_auc_score'].append(sk_rf_auc_roc)
score_dict['f1_score'].append(f1_score)
```

SKlearn Logistic Regression

```
[54]: sk_logreg = LogisticRegression(penalty='l2',fit_intercept=False,max_iter=1000)
start=datetime.now()
sk_logreg.fit(X_train_pd,y_train_pd)
sk_log_reg_runtime = (datetime.now() - start).seconds
```

Accuracy Score

```
[55]: sklearn_logreg_y_train_pred = sk_logreg.predict(X_train_pd)
sklearn_logreg_y_test_pred = sk_logreg.predict(X_test_pd)
logreg_accuracy_score = accuracy_score(y_test_pd, sklearn_logreg_y_test_pred)
print("SKLEARN Log Reg Train accuracy:     ", accuracy_score(y_train_pd,␣
↪sklearn_logreg_y_train_pred))
print("sklearn Log Reg Test accuracy:     ", logreg_accuracy_score)
```

```
SKLEARN Log Reg Train accuracy:      0.5944726136363636
sklearn Log Reg Test accuracy:      0.594470909090909
```

Confusion Matrix

[56]:
```
cnf_logreg_sk = confusion_matrix(y_test_pd,sklearn_logreg_y_test_pred)
print(cnf_logreg_sk)
```

```
[[619994 413735]
 [478429 687842]]
```

precison, sensitvity and f1_score

[57]:
```
precision,sensitivity,specificity,f1_score = get_metrics(pd.
 ↪DataFrame(cnf_logreg_sk))
print('precision is ',precision)
print('sensitivity is ',sensitivity)
print('specificity is ', specificity)
print('f1_score is ',f1_score)
```

```
precision is  0.5897788764360942
sensitivity is  0.6244157240029521
specificity is  0.564440110959075
f1_score is  0.6066032644163101
```

AUC ROC score

[58]:
```
sk_logreg_y_test_pred = sk_logreg.predict_proba(X_test_pd)
sk_logreg_auc_roc = roc_auc_score(y_test_pd,sk_logreg_y_test_pred[:,1])
print('ROC Score is ',sk_logreg_auc_roc)
```

```
ROC Score is  0.6393415834218121
```

record results

[59]:
```
score_dict['algorithm'].append('Logistic')
score_dict['implementation'].append('CPU')
score_dict['Accuracy'].append(logreg_accuracy_score)
score_dict['run_time'].append(sk_log_reg_runtime)
score_dict['precison'].append(precision)
score_dict['sensitivity'].append(sensitivity)
score_dict['roc_auc_score'].append(sk_logreg_auc_roc)
score_dict['f1_score'].append(f1_score)
```

K Nearest Neighbors sk_knn = KNeighborsClassifier(n_neighbors=7) sk_knn.fit(X_train_pd, y_train_pd)

Accuracy score start = datetime.now() sk_knn_y_test_pred = sk_knn.predict(X_test_pd) sk_knn_runtime = (datetime.now()-start).seconds sk_knn_accuracy_score = accuracy_score(y_test_pd, sk_knn_y_test_pred) #print("CuML KNN Train accuracy:", accuracy_score(y_train, cuml_knn_y_train_pred)) print("CuML KNN Test accuracy:", sk_knn_accuracy_score)

Confusion Matrix cnf_knn_sk = confusion_matrix(y_test_pd,sk_knn_y_test_pred) print(cnf_knn_sk)

ROC score sk_knn_y_test_pred = sk_knn.predict_proba(X_test_pd) sk_knn_auc_roc = roc_auc_score(y_test_pd,sk_knn_y_test_pred[:,1]) print('ROC Score is',sk_knn_auc_roc)

Sensitivity Precision and F1_score precision,sensitivity,specificity,f1_score = get_metrics(pd.DataFrame(cnf_knn_sk)) print('precision is',precision) print('sensitivity is',sensitivity) print('specificity is', specificity) print('f1_score is',f1_score)

record results score_dict['algorithm'].append('KNN') score_dict['implementation'].append('CPU') score_dict['Accuracy'].append(sk_knn_accuracy_score) score_dict['run_time'].append(sk_knn_runtime) score_dict['precison'].append(precision) score_dict['sensitivity'].append(sensitivity) score_dict['roc_auc_score'].append(sk_knn_auc_roc) score_dict['f1_score'].append(f1_score)

creating dataframe for results

[60]:
```
results_df = pd.DataFrame(data=score_dict)
```

Comparing CUML Results

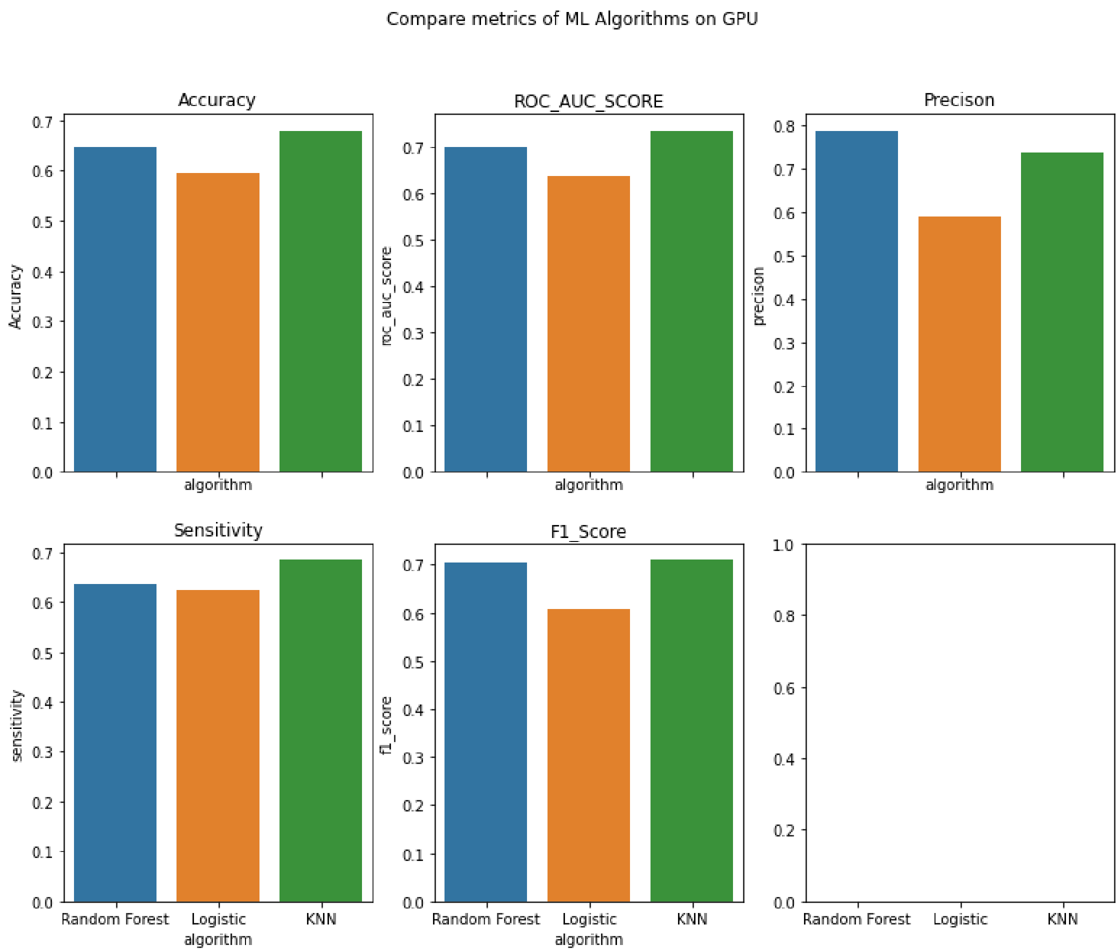
[61]:
```
cuml_scores_df = results_df[results_df['implementation']=='GPU']
```

[62]:
```
cuml_scores_df.head()
```

[62]:
```
       algorithm implementation  Accuracy  run_time  precison  sensitivity  \
0  Random Forest            GPU  0.648114         3  0.787851     0.635628
1       Logistic            GPU  0.594466         0  0.589736     0.624421
2            KNN            GPU  0.679222       370  0.735912     0.683345

   roc_auc_score  f1_score
0       0.701316  0.703600
1       0.639347  0.606583
2       0.735679  0.708655
```

[70]:
```
fig, axes = plt.subplots(2, 3,sharex=True, figsize=(13,10))
fig.suptitle('Compare metrics of ML Algorithms on GPU')

# Accuracy plot
sns.barplot(ax=axes[0,0], x='algorithm', y='Accuracy',data=cuml_scores_df)
axes[0,0].set_title('Accuracy')

# AUC ROC plot
sns.barplot(ax=axes[0,1], x='algorithm', y='roc_auc_score',data=cuml_scores_df)
axes[0,1].set_title('ROC_AUC_SCORE')

# precison plot
sns.barplot(ax=axes[0,2], x='algorithm', y='precison',data=cuml_scores_df)
```

```
axes[0,2].set_title('Precison')

# sensitivity plot
sns.barplot(ax=axes[1,0], x='algorithm', y='sensitivity',data=cuml_scores_df)
axes[1,0].set_title('Sensitivity')

# f1_score plot
sns.barplot(ax=axes[1,1], x='algorithm', y='f1_score',data=cuml_scores_df)
axes[1,1].set_title('F1_Score')

plt.show()
```

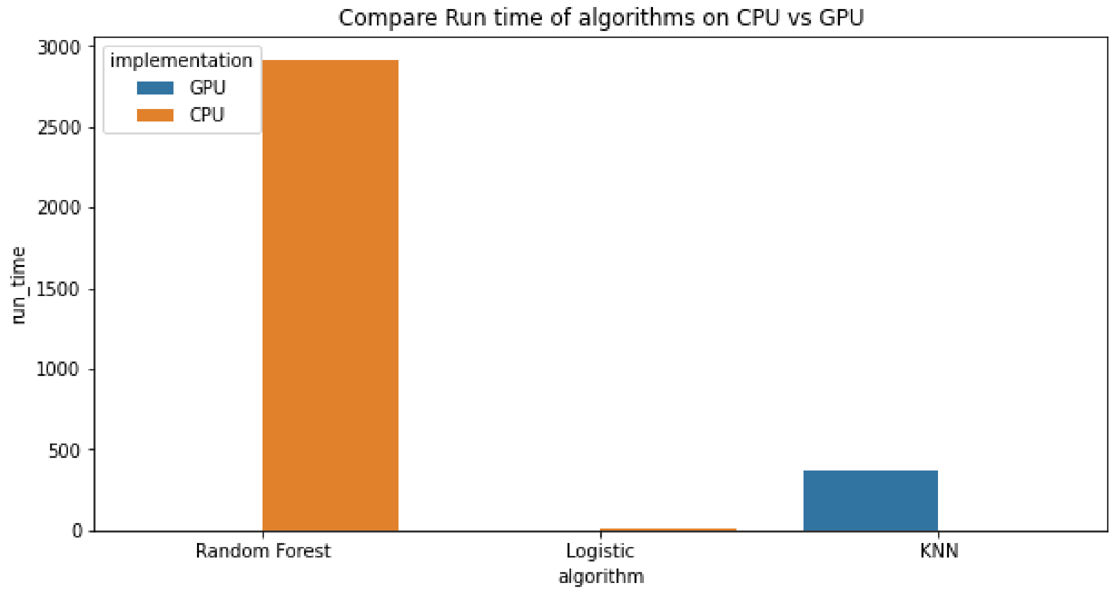


From the above results, KNN is doing better at every metric except precision over Random Forrest. Unless precision is a very important metric, I would choose KNN.

comparision of metrics of times of algorithms run on GPU vs CPU

[71]:

```
plt.figure(figsize=(10,5))
sns.barplot(x='algorithm',y='run_time',data = results_df,hue='implementation')
plt.title('Compare Run time of algorithms on CPU vs GPU')
plt.show()
```

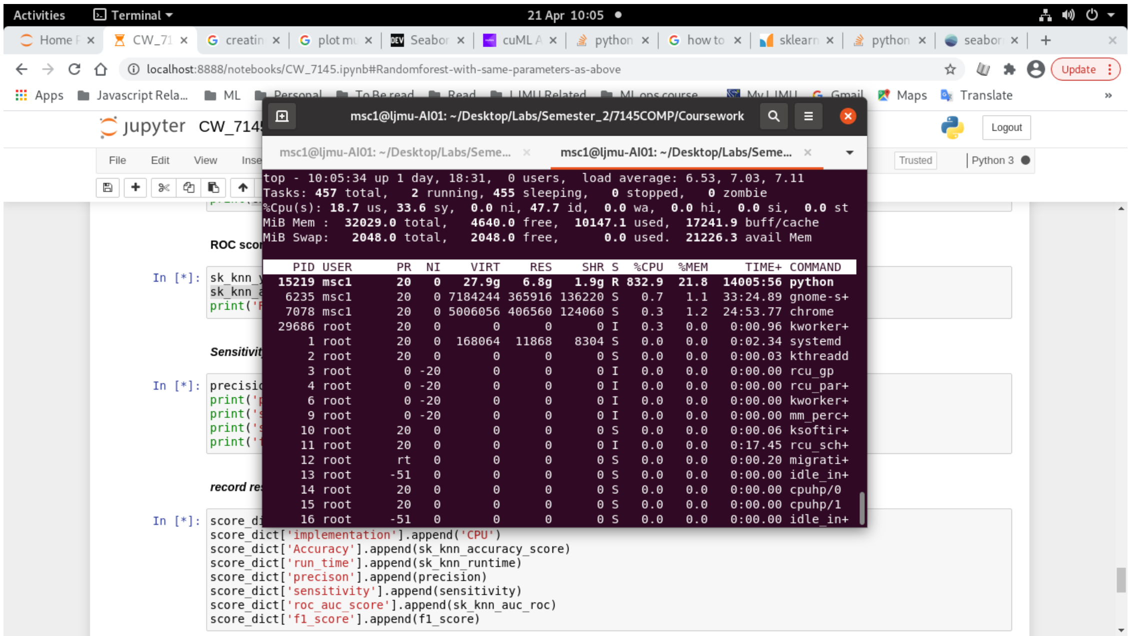


Our GPU is 10 GB RAM and 33 GB CPU RAM.
Random Forrest with Rapids took 3 seconds to run but on SKLearn it has taken 58 minutes to run the same iteration. Logistic is fairly fast on CPU taking 4 seconds comapared to under 1 seconds on Rapids but Perfomance of logisitic is poor. To run KNN predictions, it took 6 minutes 10 seconds on RAPIDS. When I tried to run on SKlearn, after 33 hours I gave up. Below is a screen shot with process time on ubuntu. To keep monitoring every 1 hr to see if there are any process breakdowns has been exhausting.

[73]:

```
from IPython.display import Image
Image(filename='Screenshot from 2021-04-21 10-05-36.png')
```

[73]:

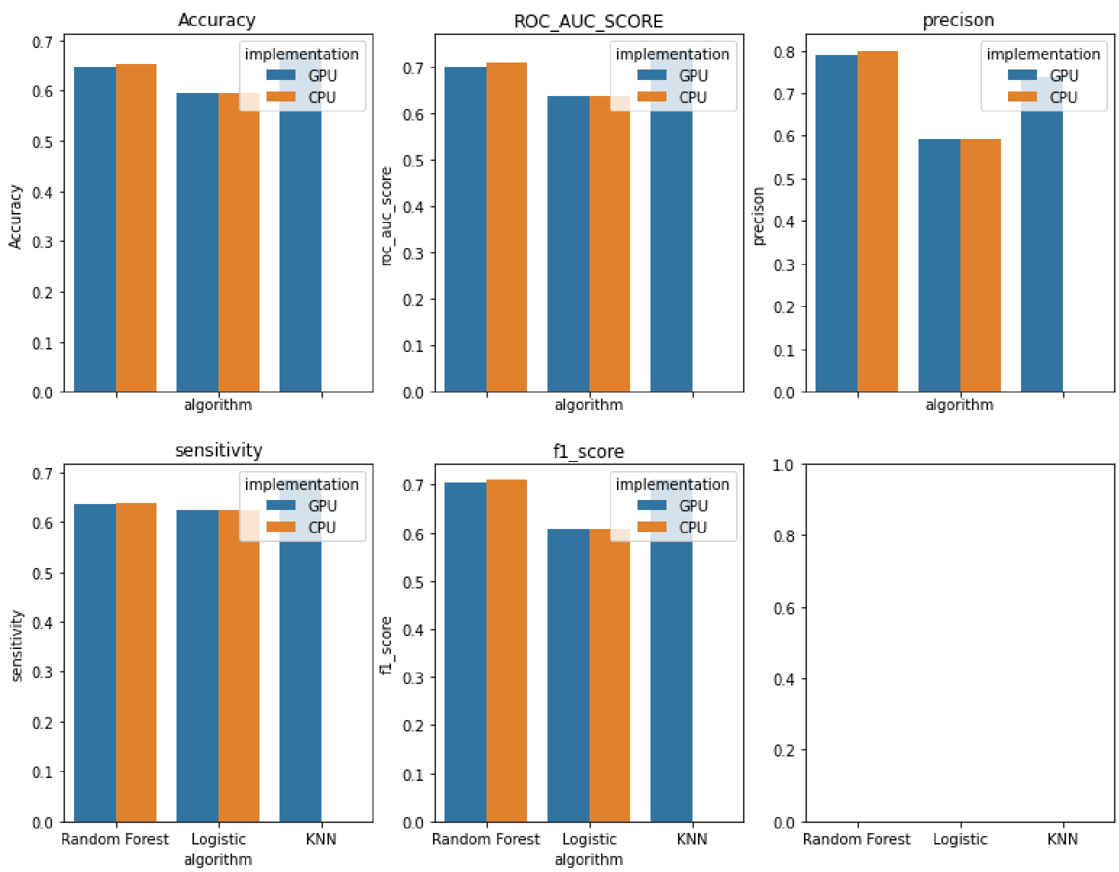


comparing algorithms results

[67]:
```
fig, axes = plt.subplots(2, 3,sharex=True, figsize=(13,10))
fig.suptitle('Compare metrics of ML Algorithms on GPU')

# Accuracy plot
sns.barplot(ax=axes[0,0], x='algorithm',␣
↪y='Accuracy',data=results_df,hue='implementation')
axes[0,0].set_title('Accuracy')

# AUC ROC plot
sns.barplot(ax=axes[0,1], x='algorithm',␣
↪y='roc_auc_score',data=results_df,hue='implementation')
axes[0,1].set_title('ROC_AUC_SCORE')

# precison plot
sns.barplot(ax=axes[0,2], x='algorithm',␣
↪y='precison',data=results_df,hue='implementation')
axes[0,2].set_title('precison')

# sensitivity plot
sns.barplot(ax=axes[1,0], x='algorithm',␣
↪y='sensitivity',data=results_df,hue='implementation')
axes[1,0].set_title('sensitivity')

# f1_score plot
```

```
sns.barplot(ax=axes[1,1], x='algorithm',␣
 ↪y='f1_score',data=results_df,hue='implementation')
axes[1,1].set_title('f1_score')

plt.show()
```

Compare metrics of ML Algorithms on GPU



Although SKlearn performaned slightly better than Rapics on same data for the same parameters, its only in decimal of 3rd order which is not very significant given the time taken to run on SKlearn/CPU.

[68]:
```
results_df
```

[68]:
```
         algorithm implementation  Accuracy  run_time  precison  sensitivity  \
0  Random Forest             GPU  0.648114         3  0.787851     0.635628
1       Logistic             GPU  0.594466         0  0.589736     0.624421
2            KNN             GPU  0.679222       370  0.735912     0.683345
3  Random Forest             CPU  0.651876      2916  0.798540     0.636913
4       Logistic             CPU  0.594471         4  0.589779     0.624416
```

	roc_auc_score	f1_score
0	0.701316	0.703600
1	0.639347	0.606583
2	0.735679	0.708655
3	0.710572	0.708627
4	0.639342	0.606603

```
[ ]:
```