Physical Activity Classification

Dương Đức Khải - 21110775
Thái Minh Bằng - 21110008
Phạm Anh Khoa - 21110776

Data Source: https://archive.ics.uci.edu/static/public/231/pamap2+physical+activity+monitoring.zip

Milestone presentation: https://youtu.be/nZcVQI0BaDY

Final report presentation: https://youtu.be/kDZoi9pfJyw

Goals:

- **Analyze the data in PAMAP2 dataset**

1. Classify which activities a person is performing (walking, running, lying, ...)
2. Determine important features for physical activity recognition

- **Classification methods**

1. Logistic regression
2. Random forest
3. K-NN
4. Other algorithms

Motivations

- Monitor seniors, and the mentally disabled from performing harmful or unusual activities

- Gaming: track the player's activities in real life and translate into actions in virtual reality games

Problems Addressed

1. **Filtering noise data**
2. **Selecting important data for model training**
3. **Train models with different algorithms**
4. **Evaluate models' performance**
5. **Hyperparameters tuning models**

DATA

Subject Information

Subject ID	Sex	Age (years)	Height (cm)	Weight (kg)	Resting HR (bpm)	Max HR (bpm)	Dominant hand
101	Male	27	182	83	75	193	right
102	Female	25	169	78	74	195	right
103	Male	31	187	92	68	189	right
104	Male	24	194	95	58	196	right
105	Male	26	180	73	70	194	right
106	Male	26	183	69	60	194	right
107	Male	23	173	86	60	197	right
108	Male	32	179	87	66	188	left
109	Male	31	168	65	54	189	right

DATA

1.timestamp

2.activityID

3.heartrate

4.handTempeture

5-54. Other raw sensory data

```
colNames = ["timestamp", "activityID","heartrate"]

IMUhand = ['handTemperature',
           'handAcc16_1', 'handAcc16_2', 'handAcc16_3',
           'handAcc6_1', 'handAcc6_2', 'handAcc6_3',
           'handGyro1', 'handGyro2', 'handGyro3',
           'handMagne1', 'handMagne2', 'handMagne3',
           'handOrientation1', 'handOrientation2', 'handOrientation3', 'handOrientation4']

IMUchest = ['chestTemperature',
           'chestAcc16_1', 'chestAcc16_2', 'chestAcc16_3',
           'chestAcc6_1', 'chestAcc6_2', 'chestAcc6_3',
           'chestGyro1', 'chestGyro2', 'chestGyro3',
           'chestMagne1', 'chestMagne2', 'chestMagne3',
           'chestOrientation1', 'chestOrientation2', 'chestOrientation3', 'chestOrientation4']

IMUankle = ['ankleTemperature',
           'ankleAcc16_1', 'ankleAcc16_2', 'ankleAcc16_3',
           'ankleAcc6_1', 'ankleAcc6_2', 'ankleAcc6_3',
           'ankleGyro1', 'ankleGyro2', 'ankleGyro3',
           'ankleMagne1', 'ankleMagne2', 'ankleMagne3',
           'ankleOrientation1', 'ankleOrientation2', 'ankleOrientation3', 'ankleOrientation4']
```

DATA

- 1 lying
- 2 sitting
- 3 standing
- 4 walking
- 5 running
- 6 cycling
- 7 Nordic walking
- 9 watching TV
- 10 computer work
- 11 car driving
- 12 ascending stairs
- 13 descending stairs
- 16 vacuum cleaning
- 17 ironing
- 18 folding laundry
- 19 house cleaning
- 20 playing soccer
- 24 rope jumping
- 0 other (transient activities)

DATA CLEANING

- There were some wireless disconnections in data collection. Therefore the missing data has to be accounted for and made up in a way that our data analysis will not be impacted.
- activityID 0 must be removed completely from our dataset since this is transient period where the subject was not doing any particular activity.

DATA CLEANING

Activities performed by subjects (in seconds)

	subject101	subject102	subject103	subject104	subject105	subject106	subject107	subject108	subject109	Sum	Nr. of subjects
1 – lying	271.86	234.29	220.43	230.46	236.98	233.39	256.1	241.64	0	1925.15	8
2 – sitting	234.79	223.44	287.6	254.91	268.63	230.4	122.81	229.22	0	1851.8	8
3 – standing	217.16	255.75	205.32	247.05	221.31	243.55	257.5	251.59	0	1899.23	8
4 – walking	222.52	325.32	290.35	319.31	320.32	257.2	337.19	315.32	0	2387.53	8
5 – running	212.64	92.37	0	0	246.45	228.24	36.91	165.31	0	981.92	6
6 – cycling	235.74	251.07	0	226.98	245.76	204.85	226.79	254.74	0	1645.93	7
7 – Nordic walking	202.64	297.38	0	275.32	262.7	266.85	287.24	288.87	0	1881	7
9 – watching TV	836.45	0	0	0	0	0	0	0	0	836.45	1
10 – computer work	0	0	0	0	1108.82	617.76	0	687.24	685.49	3099.31	4
11 – car driving	545.18	0	0	0	0	0	0	0	0	545.18	1
12 – ascending stairs	158.88	173.4	103.87	166.92	142.79	132.89	176.44	116.81	0	1172	8
13 – descending stairs	148.97	152.11	152.72	142.83	127.25	112.7	116.16	96.53	0	1049.27	8
16 – vacuum cleaning	229.4	206.82	203.24	200.36	244.44	210.77	215.51	242.91	0	1753.45	8
17 – ironing	235.72	288.79	279.74	249.94	330.33	377.43	294.98	329.89	0	2386.82	8
18 – folding laundry	271.13	0	0	0	0	217.85	0	236.49	273.27	998.74	4
19 – house cleaning	540.88	0	0	0	284.87	287.13	0	416.9	342.05	1871.83	5
20 – playing soccer	0	0	0	0	0	0	0	181.24	287.88	469.12	2
24 – rope jumping	129.11	132.61	0	0	77.32	2.55	0	88.05	63.9	493.54	6
Labeled total	4693.07	2633.35	1743.27	2314.08	4117.97	3623.56	2327.63	4142.75	1652.59	27248.27	
Total	6957.67	4469.99	2528.32	3295.75	5295.54	4917.78	3135.98	5884.41	2019.47	38504.91	

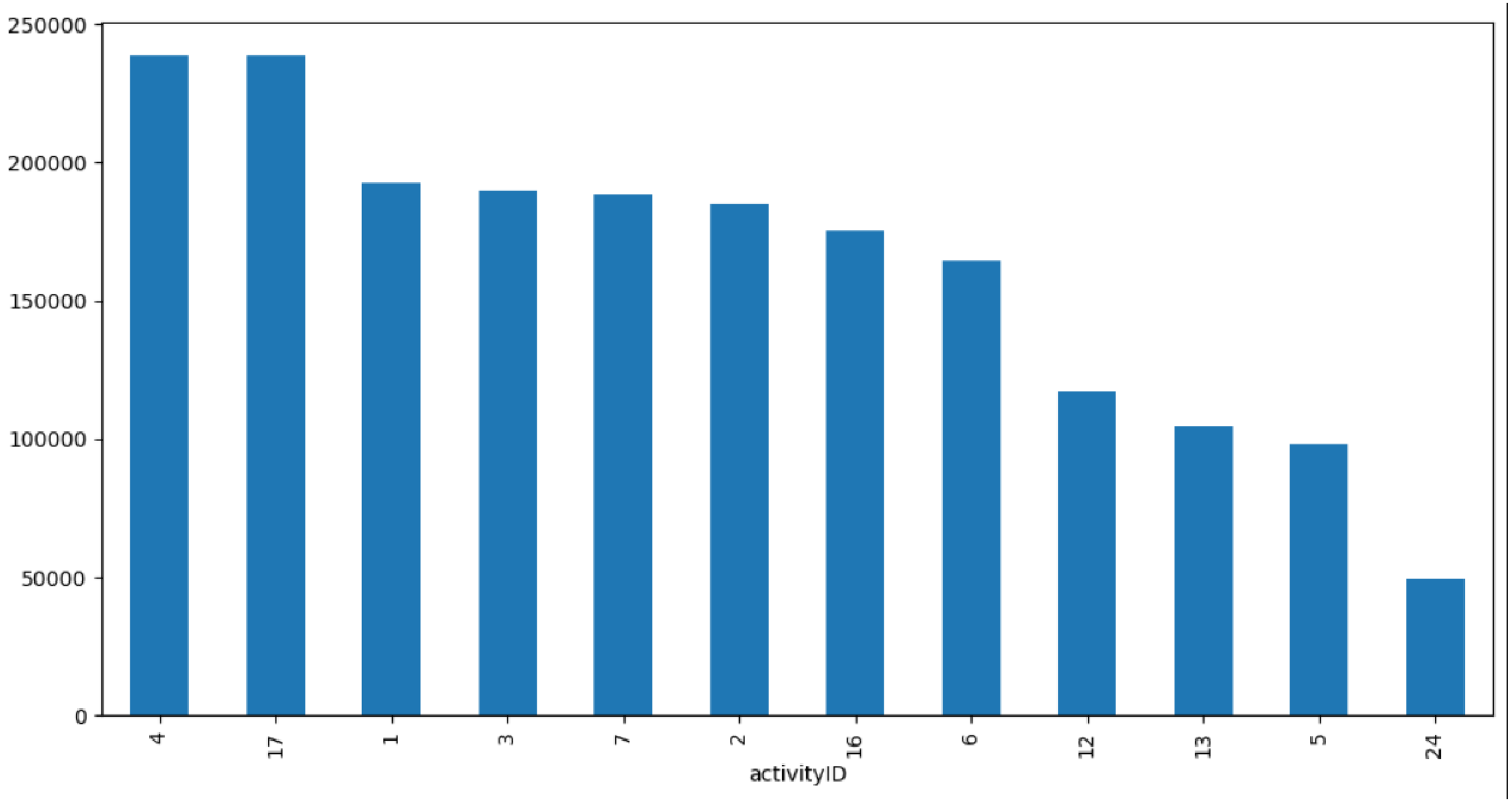
DATA CLEANING

	timestamp	activityID	heartrate
0	37.66	1	NaN
1	37.67	1	NaN
2	37.68	1	NaN
3	37.69	1	NaN
4	37.70	1	100.0
5	37.71	1	100.0
6	37.72	1	100.0
7	37.73	1	100.0
8	37.74	1	100.0
9	37.75	1	100.0

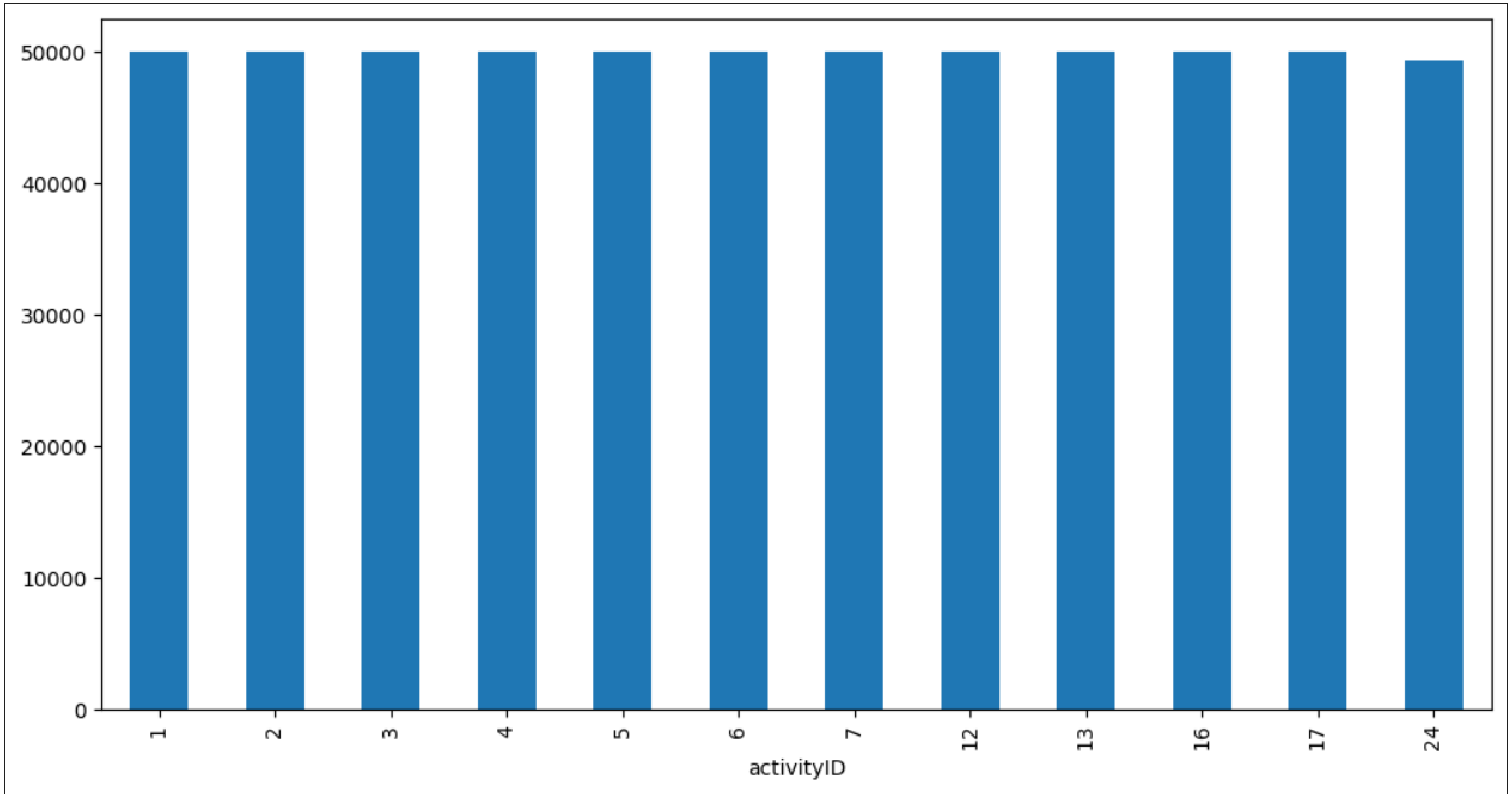
heartrate still has NaN values

```
for i in range(0,4):
    dataCol["heartrate"].iloc[i]=100
```

DATA CLEANING



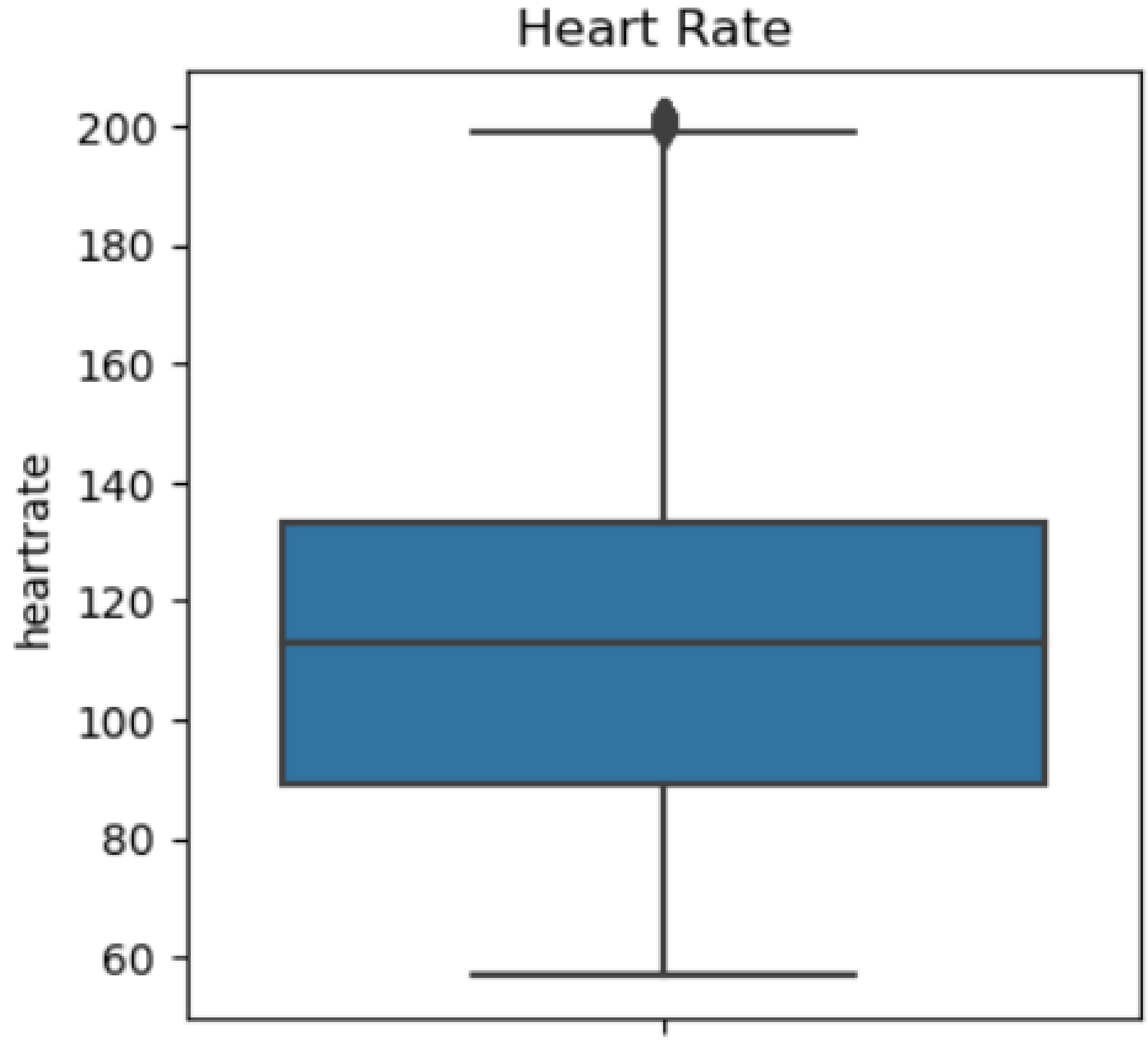
DATA CLEANING



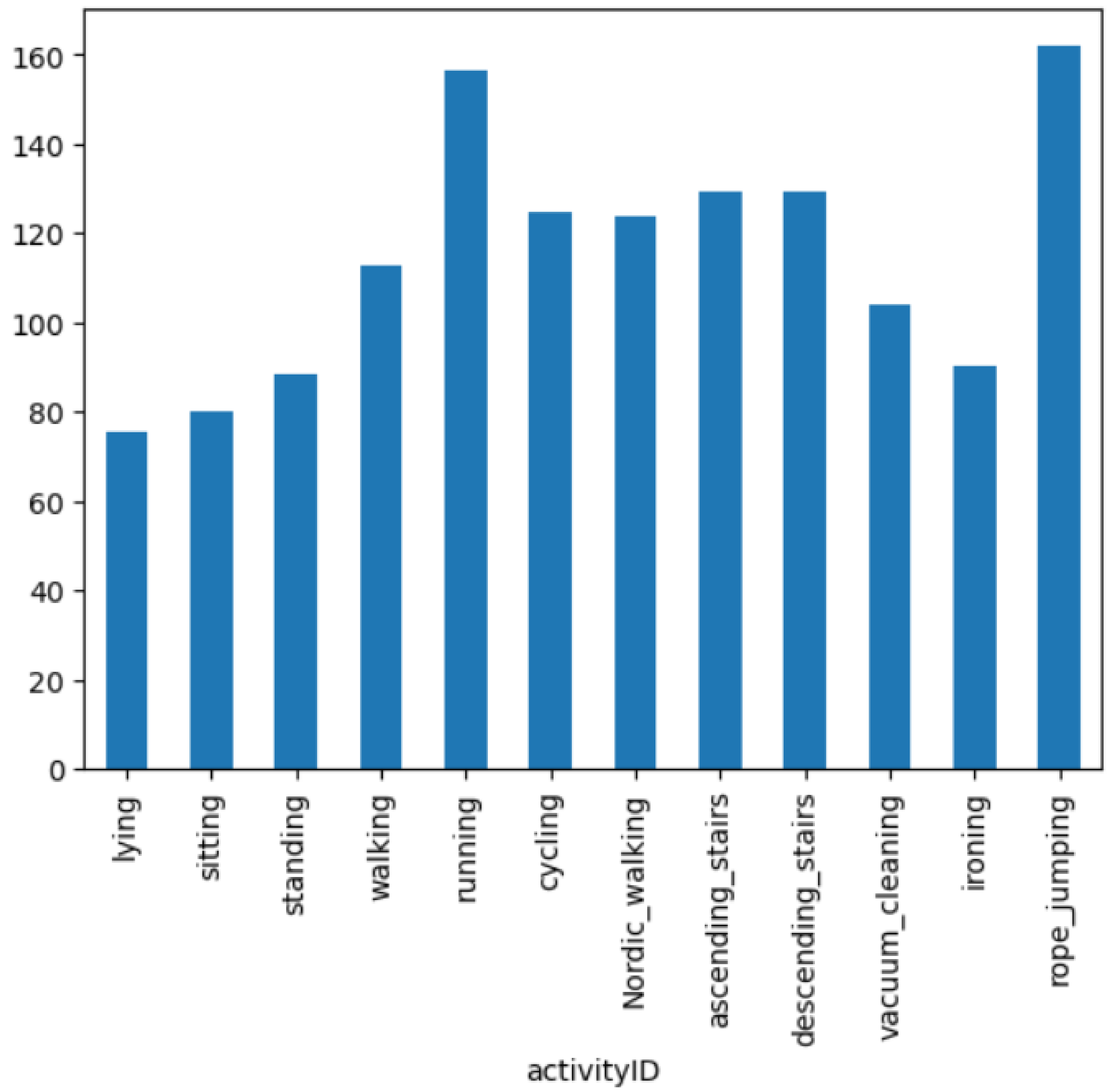
DATA CLEANING

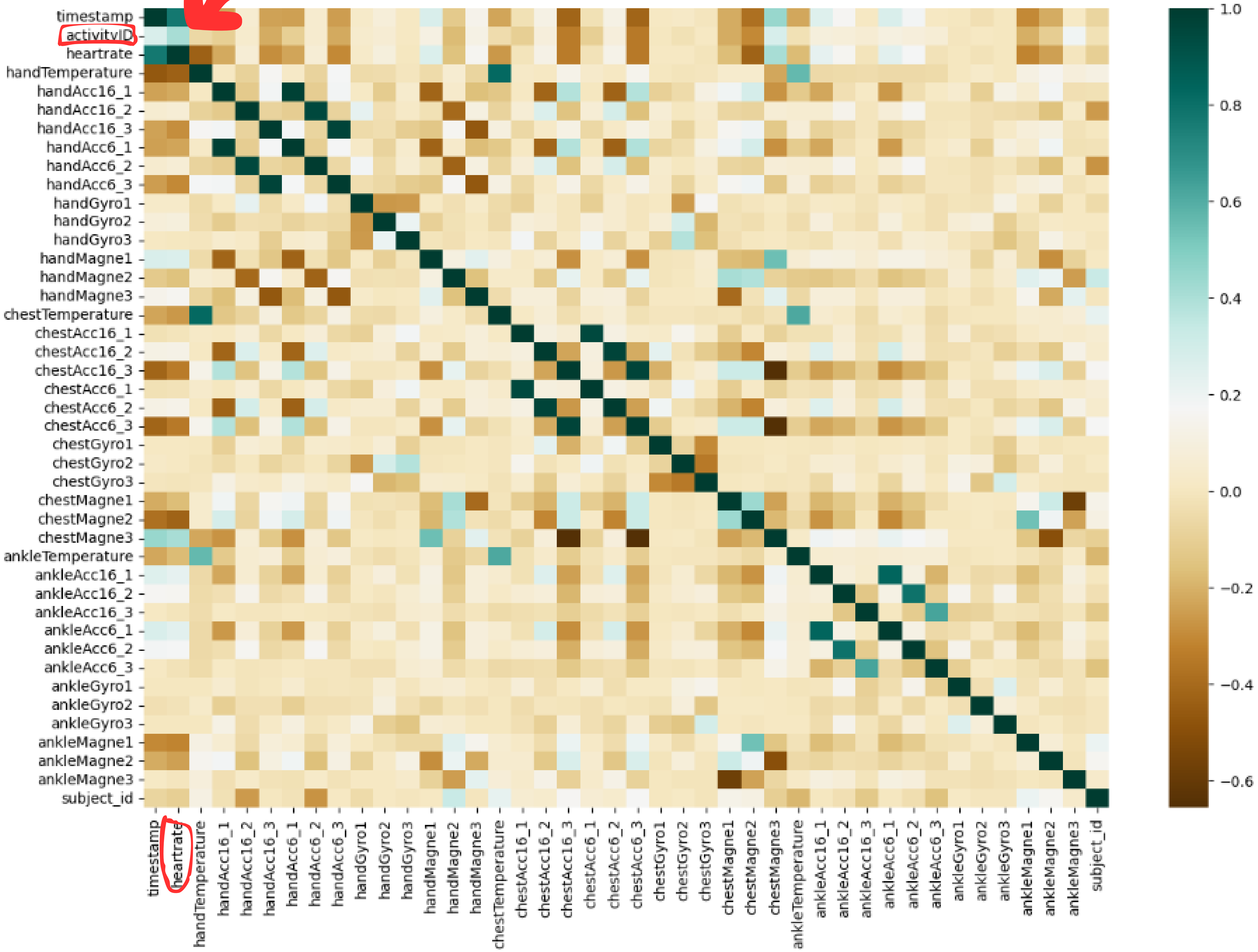
	timestamp	activityID	heartrate	handTemperature	handAcc16_1
count	479488.000000	479488.000000	479488.000000	479488.000000	479488.000000
mean	1899.408018	9.144890	114.700567	32.507942	-5.079252
std	1175.267993	6.852498	30.977854	2.008550	6.745207
min	31.200000	1.000000	57.000000	24.875000	-106.957000
25%	834.097500	3.000000	89.000000	31.062500	-9.049547
50%	1804.400000	6.000000	113.000000	33.062500	-5.472365
75%	2951.350000	13.000000	133.000000	33.937500	-0.823944
max	4245.670000	24.000000	202.000000	35.500000	60.912600

DATA CLEANING



DATA CLEANING




timestamp
activityID
heartrate
handTemperature
handAcc16_1
handAcc16_2
handAcc16_3
handAcc6_1
handAcc6_2
handAcc6_3
handGyro1
handGyro2
handGyro3
handMagne1
handMagne2
handMagne3
chestTemperature
chestAcc16_1
chestAcc16_2
chestAcc16_3
chestAcc6_1
chestAcc6_2
chestAcc6_3
chestGyro1
chestGyro2
chestGyro3
chestMagne1
chestMagne2
chestMagne3
ankleTemperature
ankleAcc16_1
ankleAcc16_2
ankleAcc16_3
ankleAcc6_1
ankleAcc6_2
ankleAcc6_3
ankleGyro1
ankleGyro2
ankleGyro3
ankleMagne1
ankleMagne2
ankleMagne3
subject_id
timestamp
heartrate
handTemperature
handAcc16_1
handAcc16_2
handAcc16_3
handAcc6_1
handAcc6_2
handAcc6_3
handGyro1
handGyro2
handGyro3
handMagne1
handMagne2
handMagne3
chestTemperature
chestAcc16_1
chestAcc16_2
chestAcc16_3
chestAcc6_1
chestAcc6_2
chestAcc6_3
chestGyro1
chestGyro2
chestGyro3
chestMagne1
chestMagne2
chestMagne3
ankleTemperature
ankleAcc16_1
ankleAcc16_2
ankleAcc16_3
ankleAcc6_1
ankleAcc6_2
ankleAcc6_3
ankleGyro1
ankleGyro2
ankleGyro3
ankleMagne1
ankleMagne2
ankleMagne3
subject_id
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6

**Null Hypothesis: **

- h0 : The mean heart rate of the cumbersome activities has no mass difference from the mean of all activities

**Non Null Hypothesis: **

- h1 : The mean heart rate of the cumbersome activities has mass difference from the mean of all activities

```
running_data = train_df.loc[(train_df["activityID"] == 5)]
ropejumping_data = train_df.loc[(train_df["activityID"] == 24)]
cumbersome_data = running_data + ropejumping_data
```

```
import scipy.stats

p = train_df['heartrate'].mean() / (running_data['heartrate'].std() / math.sqrt( running_data['heartrate'].count() ))
pValue = 1 - scipy.stats.norm.cdf(p)

if pValue > 0.1:
    print("The p_value is ", pValue, " and h1 is rejected. There is no mass difference between the means of cumbersome activities and all activities.")
else:
    print("The p_value is ", pValue, " and h0 is rejected. There is mass difference between the means of cumbersome activities and all activities.")
```

```
The p_value is  0.0  and h0 is rejected. There is mass difference between the means of cumbersome activities and all activities.
```

Algorithms selection

- **Logistic Regression**

```
# Create and train the Logistic Regression model
log_reg = LogisticRegression()
log_reg.fit(X_train, y_train)
```

- **Random Forest Classification**

```
from sklearn.ensemble import RandomForestClassifier
rfc = RandomForestClassifier(n_jobs =4)
rfc.fit(X_train,y_train)
```

Algorithms selection

- **KNN**

```
from sklearn.neighbors import KNeighborsClassifier
neightbor_scenario = [1, 2, 3, 5, 10, 50, 100, 500, 1000]
for neighbor in neightbor_scenario:
    knc = KNeighborsClassifier(n_neighbors=neighbor)
    knc.fit(X_train, y_train)
    y_pred_kn = knc.predict(X_test)
    print("Neighbor: " + str(neighbor))
    get_metrics(y_test, y_pred_kn)
```

- **Naive Bayes**

```
gnbc = GaussianNB()
gnbc.fit(X_train, y_train)
y_pred_gnb = gnbc.predict(X_test)
print("Gaussian Naive Bayes:")
get_metrics(y_test, y_pred_gnb)
```

Algorithms selection

- **Adaboost**

```
#default estimator is decision tree
aboostc = AdaBoostClassifier(n_estimators=100)
aboostc.fit(X_train, y_train)
y_pred_ada = aboostc.predict(X_test)
print("AdaBoost:")
get_metrics(y_test, y_pred_ada)
```

- **Bagging**

```
#default estimator is decision tree
bagc = BaggingClassifier(n_estimators=100)
bagc.fit(X_train, y_train)
y_pred_bag = bagc.predict(X_test)
print("Bagging:")
get_metrics(y_test, y_pred_bag)
```

Model evaluation

- **Using accuracy and f-score to compare models' performance**

```
def get_metrics (y_true,y_pred):
    acc = accuracy_score(y_true, y_pred)
    err = 1-acc
    p = precision_score(y_true, y_pred,average=None).mean()
    r = recall_score(y_true, y_pred, average=None).mean()
    f1 = f1_score(y_true, y_pred, average=None).mean()

    print("Accuracy: ",acc)
    print("Error: ",err)
    print("Precision", p)
    print("Recall", r)
    print("F1", f1)
```

Model Performance

Before removing ouliers

```
Accuracy:  0.17077519864856441
Error:  0.8292248013514358
Precision 0.15265408332808045
Recall 0.16968907491682397
F1 0.15852497628438153
```

Logistic Regression

```
Accuracy:  0.3434691025881666
Error:  0.6565308974118333
Precision 0.40472849443345066
Recall 0.34253447675415866
F1 0.3470570099230174
```

Random Forest

Removing ouliers

```
# Drop any columns that are not relevant for the model, e.g., 'activityID'
X = filtered_data.drop(['activityID'], axis=1)
y = filtered_data['activityID']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Standardize the features using StandardScaler
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Identify and remove outliers based on z-scores
z_scores = zscore(X_train)
abs_z_scores = np.abs(z_scores)
filtered_entries = (abs_z_scores < 3).all(axis=1)
X_train = X_train[filtered_entries]
y_train = y_train[filtered_entries]
```

Model Performance

After removing ouliers

```
Accuracy:  0.8878887479978644
Error:  0.1121112520021356
Precision 0.8872340738686941
Recall 0.8877075080101632
F1 0.8873513427669546
```

Logistic Regression

```
Accuracy:  0.9844500800854245
Error:  0.01554991991457555
Precision 0.9859331514729814
Recall 0.9843418337059963
F1 0.9844500432869703
```

Random Forest

Model Performance

```
Neighbor: 1
Accuracy:  0.9656049786438868
Error:  0.034395021356113165
Precision 0.9665117669265197
Recall 0.965389516957873
F1 0.9652469892138615
```

KNN (neightbor = 1)

```
Gaussian Naive Bayes:
Accuracy:  0.8443005872931126
Error:  0.15569941270688736
Precision 0.85517015201471
Recall 0.8442515831196534
F1 0.8459131119877021
```

Gaussian Naive Bayes

Model Performance

```
AdaBoost Logistic Regression:
Accuracy:  0.542553723972237
Error:  0.457446276027763
Precision 0.5426036050623587
Recall 0.5418570465359541
F1 0.5050658834621337
```

Logistic Regression AdaBoost

```
AdaBoost:
Accuracy:  0.29548184730379073
Error:  0.7045181526962092
Precision 0.2170094107653123
Recall 0.2935248816324761
F1 0.2163457600405143
```

Decision Tree AdaBoost

Model Performance

```
Bagging:
Accuracy:  0.9796866657768286
Error:  0.020313334223171386
Precision 0.9810165887545729
Recall 0.9795230206803334
F1 0.9795738981979842
```

Decision Tree
Bagging

Hyperparameter Tuning

Tuning Models:

- Logistic Regression
- KNN
- Random Forest

Tuning Methods:

- GridSearchCV
- RandomSearchCV

Logistic Regression Tuning

Tuned Parameters

```
solvers = ['newton-cg', 'lbfgs', 'liblinear']
penalty = ['l2']
c_values = [100, 10, 1.0, 0.1, 0.01]
```

Implementation

```
grid = dict(solver=solvers,penalty=penalty,C=c_values)
cv = RepeatedStratifiedKFold(n_splits=5, n_repeats=2, random_state=0)
grid_search = GridSearchCV(estimator=model, param_grid=grid, n_jobs=-1, cv=cv, scoring='accuracy',error_score=0)
grid_result = grid_search.fit(X_train, y_train)
```

LR Tuning results

```
Best: 0.949094 using {'C': 100, 'penalty': 'l2', 'solver': 'newton-cg'}
0.949094 (0.000575) with: {'C': 100, 'penalty': 'l2', 'solver': 'newton-cg'}
0.939033 (0.000823) with: {'C': 100, 'penalty': 'l2', 'solver': 'lbfgs'}
0.861485 (0.000944) with: {'C': 100, 'penalty': 'l2', 'solver': 'liblinear'}
0.948137 (0.000556) with: {'C': 10, 'penalty': 'l2', 'solver': 'newton-cg'}
0.939424 (0.000589) with: {'C': 10, 'penalty': 'l2', 'solver': 'lbfgs'}
0.861044 (0.000934) with: {'C': 10, 'penalty': 'l2', 'solver': 'liblinear'}
0.942882 (0.000655) with: {'C': 1.0, 'penalty': 'l2', 'solver': 'newton-cg'}
0.938710 (0.000904) with: {'C': 1.0, 'penalty': 'l2', 'solver': 'lbfgs'}
0.860115 (0.000916) with: {'C': 1.0, 'penalty': 'l2', 'solver': 'liblinear'}
0.924212 (0.000812) with: {'C': 0.1, 'penalty': 'l2', 'solver': 'newton-cg'}
0.924397 (0.000806) with: {'C': 0.1, 'penalty': 'l2', 'solver': 'lbfgs'}
0.855685 (0.000952) with: {'C': 0.1, 'penalty': 'l2', 'solver': 'liblinear'}
0.889217 (0.000685) with: {'C': 0.01, 'penalty': 'l2', 'solver': 'newton-cg'}
0.889219 (0.000680) with: {'C': 0.01, 'penalty': 'l2', 'solver': 'lbfgs'}
0.843215 (0.001085) with: {'C': 0.01, 'penalty': 'l2', 'solver': 'liblinear'}
```

LR before & after Tuning

```
Accuracy:  0.8850023358248799
Error:  0.11499766417512014
Precision 0.8853531954603019
Recall 0.884499236387562
F1 0.882162449402084
```

Before Tuning

```
Accuracy:  0.9057411238654565
Error:  0.09425887613454353
Precision 0.9070117087196653
Recall 0.9053292926711437
F1 0.903831720532872
```

After Tuning

KNN Tuning

Tuned Parameters

```
n_neighbors = range(1, 5, 1)
weights = ['uniform', 'distance']
metric = ['euclidean', 'manhattan', 'minkowski']
```

Implementation

```
grid = dict(n_neighbors=n_neighbors,weights=weights,metric=metric)
grid_search = RandomizedSearchCV(estimator=model, param_distributions=grid,
                                 n_jobs=-1, cv=5, scoring='accuracy', n_iter=24)
grid_result = grid_search.fit(X_train, y_train)
```

KNN Tuning results

```
Best: 0.999078 using {'weights': 'uniform', 'n_neighbors': 1, 'metric': 'manhattan'}
0.997673 (0.000135) with: {'weights': 'uniform', 'n_neighbors': 1, 'metric': 'euclidean'}
0.997673 (0.000135) with: {'weights': 'distance', 'n_neighbors': 1, 'metric': 'euclidean'}
0.995720 (0.000172) with: {'weights': 'uniform', 'n_neighbors': 2, 'metric': 'euclidean'}
0.997673 (0.000135) with: {'weights': 'distance', 'n_neighbors': 2, 'metric': 'euclidean'}
0.996045 (0.000200) with: {'weights': 'uniform', 'n_neighbors': 3, 'metric': 'euclidean'}
0.996637 (0.000171) with: {'weights': 'distance', 'n_neighbors': 3, 'metric': 'euclidean'}
0.994758 (0.000194) with: {'weights': 'uniform', 'n_neighbors': 4, 'metric': 'euclidean'}
0.996743 (0.000138) with: {'weights': 'distance', 'n_neighbors': 4, 'metric': 'euclidean'}
0.999078 (0.000081) with: {'weights': 'uniform', 'n_neighbors': 1, 'metric': 'manhattan'}
0.999078 (0.000081) with: {'weights': 'distance', 'n_neighbors': 1, 'metric': 'manhattan'}
0.998332 (0.000089) with: {'weights': 'uniform', 'n_neighbors': 2, 'metric': 'manhattan'}
0.999078 (0.000081) with: {'weights': 'distance', 'n_neighbors': 2, 'metric': 'manhattan'}
0.998529 (0.000099) with: {'weights': 'uniform', 'n_neighbors': 3, 'metric': 'manhattan'}
0.998742 (0.000087) with: {'weights': 'distance', 'n_neighbors': 3, 'metric': 'manhattan'}
0.997996 (0.000162) with: {'weights': 'uniform', 'n_neighbors': 4, 'metric': 'manhattan'}
0.998833 (0.000075) with: {'weights': 'distance', 'n_neighbors': 4, 'metric': 'manhattan'}
0.997673 (0.000135) with: {'weights': 'uniform', 'n_neighbors': 1, 'metric': 'minkowski'}
0.997673 (0.000135) with: {'weights': 'distance', 'n_neighbors': 1, 'metric': 'minkowski'}
0.995720 (0.000172) with: {'weights': 'uniform', 'n_neighbors': 2, 'metric': 'minkowski'}
0.997673 (0.000135) with: {'weights': 'distance', 'n_neighbors': 2, 'metric': 'minkowski'}
0.996045 (0.000200) with: {'weights': 'uniform', 'n_neighbors': 3, 'metric': 'minkowski'}
0.996637 (0.000171) with: {'weights': 'distance', 'n_neighbors': 3, 'metric': 'minkowski'}
0.994758 (0.000194) with: {'weights': 'uniform', 'n_neighbors': 4, 'metric': 'minkowski'}
0.996743 (0.000138) with: {'weights': 'distance', 'n_neighbors': 4, 'metric': 'minkowski'}
```

KNN before & after Tuning

```
Neighbor: 1
Accuracy:  0.9656049786438868
Error:  0.0343950213561131 65
Precision 0.9665117669265197
Recall 0.965389516957873
F1 0.9652469892138615
```

Before Tuning (neightbor = 1)

```
Accuracy:  0.9793613187399893
Error:  0.0206386812600107
Precision 0.9798089593507512
Recall 0.9791925482810931
F1 0.9791787788699856
```

After Tuning (neightbor = 1)

Random Forest Tuning

Tuned Parameters

```
n_estimators = [10, 100, 1000]
max_features = ['sqrt', 'log2']
```

Implementation

```
grid = dict(n_estimators=n_estimators,max_features=max_features)
grid_search = RandomizedSearchCV(estimator=model, param_distributions=grid,
                                 n_jobs=-1, cv=4, scoring='accuracy', n_iter=1)
grid_result = grid_search.fit(X_train, y_train)
```

RF Tuning results

```
Best: 0.999632 using {'n_estimators': 100, 'max_features': 'log2'}
0.999632 (0.000051) with: {'n_estimators': 100, 'max_features': 'log2'}
```

Model Performance

```python
classifiers = [LogisticRegression(penalty='l2', C=100, solver='newton-cg', n_jobs=-1),
               KNeighborsClassifier(n_neighbors=1, metric='manhattan', weights='uniform', n_jobs=-1),
               RandomForestClassifier(n_estimators=20, n_jobs=-1)]

score_lst = []
for cls in classifiers:
    y_pred = cross_val_predict(cls, X_train, y_train, cv=5)
    accs = accuracy_score(y_train, y_pred)
    mse_scores = cross_val_score(cls, X_train, y_train, scoring="neg_mean_squared_error", cv=2)
    mse = np.sqrt(-mse_scores)
    f1 = cross_val_score(cls, X_train, y_train, scoring="f1_macro", cv=2)

    score_lst.append([cls.__class__.__name__, accs, mse.mean(), f1.mean()])
```

Cross-validation for Logistic regression, KNN & Random forest

Model Performance

	Classifier	Accuracy	MSE	F1
0	LogisticRegression	0.949109	0.998275	0.942381
1	KNeighborsClassifier	0.999078	0.278427	0.998286
2	RandomForestClassifier	0.999494	0.163299	0.998981

Result

Conclusion

1. **Drop unimportant data (NaN values, ActivityID = 0, timestamp)**
2. **Select equal amount of data from each Activity ID**
3. **Model building serveral algorithms**
4. **Tunned high performed models**

THANK YOU