

A business venture by  
Spotify, using the

## **WISDM Smartphone and Smartwatch Activity and Biometrics Dataset**

Alejandro Hohmann

Garrett Michael

Leonardo Clo



# Problem Definition

# Problem Definition – Classification of User Activity

- We (Data Scientists at Spotify) would like to dynamically target advertising to non-premium members based on their physical activity while using Spotify services. For example, while a listener is enjoying a podcast and folding their laundry, they would receive an ad for laundry detergent.



Algorithm detects user is doing laundry



An ad for detergent is triggered

# Problem Definition

- In addition, Spotify also wishes to cater to our premium members by enhancing music recommendation/auto-play options based on a member's physical activity. For example, while a user is exercising play up-tempo music, and while a user is eating pasta play Italian classics.

# Data Description

# Data Description

- Accelerometer and Gyroscope data was collected from 51 volunteer subjects. Each subject was asked to perform 18 tasks for 3 minutes each.

**51 Subjects**

**18 Activities**

**3 Minutes**

**4 Sensors**

**2 Devices**

# Data Description

- The 18 tasks were a mix of physical activities that could be distinctly identified, such as walking, eating, laundry, etc. We (Spotify) tried to collect data for activities that our members might be doing while using our services. The tasks are listed in the table.

THE 18 ACTIVITIES REPRESENTED IN DATA SET

Activity	Code
Walking	A
Jogging	B
Stairs	C
Sitting	D
Standing	E
Typing	F
Brushing Teeth	G
Eating Soup	H
Eating Chips	I
Eating Pasta	J
Drinking from Cup	K
Eating Sandwich	L
Kicking (Soccer Ball)	M
Playing Catch w/Tennis Ball	O
Dribbling (Basketball)	P
Writing	Q
Clapping	R
Folding Clothes	S

## Data Description Cont.

- Each subject had a smartwatch placed on his/her dominant hand and a smartphone in their pocket.
- The smartphone and smartwatch both had an accelerometer and gyroscope, yielding four total sensors (Phone - Accelerometer, Phone - Gyroscope, Watch – Accelerometer, Watch - Gyroscope).





DEFINITION OF ELEMENTS IN RAW DATA MEASUREMENTS

Field name	Description
Subject-id	Type: Symbolic numeric identifier. Uniquely identifies the subject. Range: 1600-1650.
Activity code	Type: Symbolic single letter. Identifies a specific activity as listed in Table 2. Range: A-S (no "N" value)
Timestamp	Type: Integer. Linux time
x	Type Numeric: real. Sensor value for x axis. May be positive or negative.
y	Same as x but for y axis
z	Same as x but for z axis

## Data Description Cont.

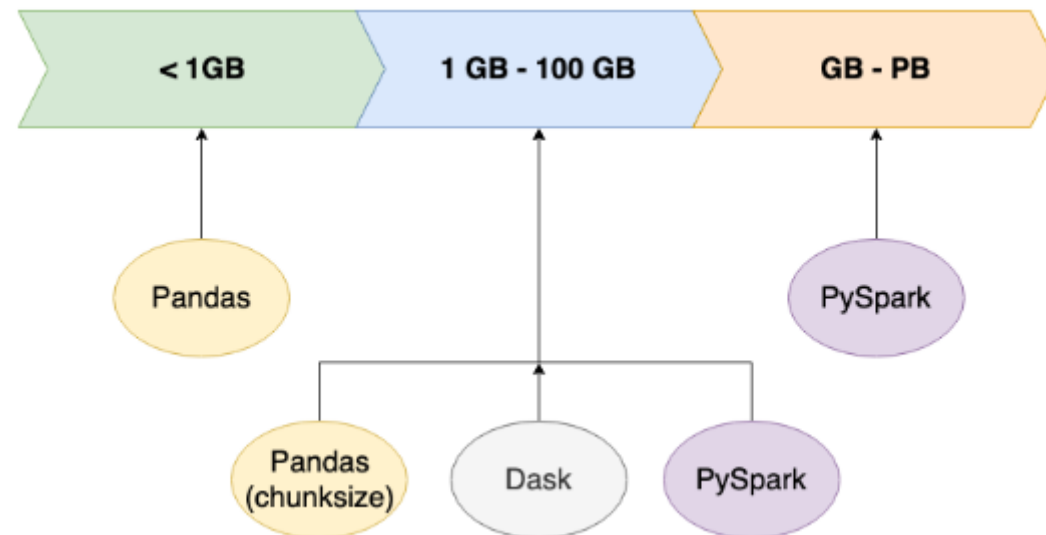
- The results for each subject are stored in a comma delimited text file. Since there are 51 subjects and 4 different sensors, there are a total of 204 text files. Each text file has the same six attributes:
  - Subject-id,
  - Activity Code,
  - Timestamp,
  - x, y, z

# Data Preparation

Dask or Spark?

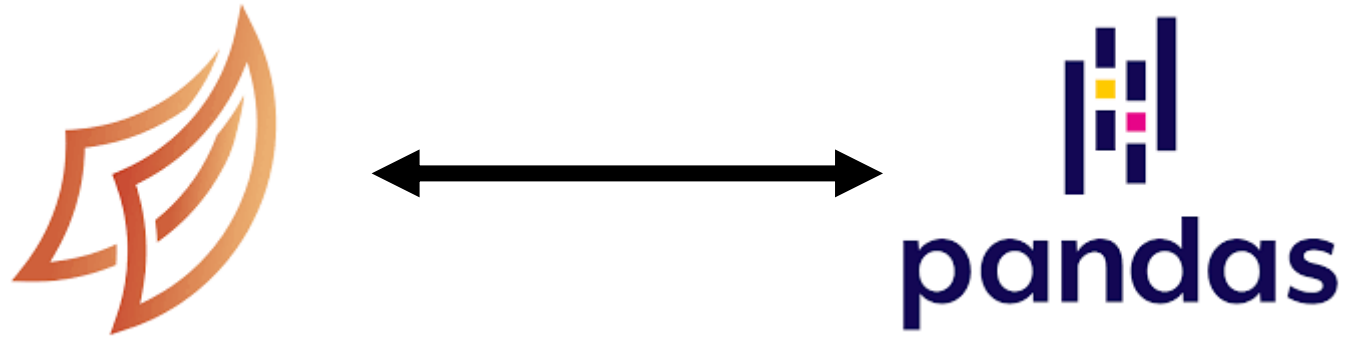
# Data Preparation – Dask or Spark?

- To clean, prepare and train our data, we decided to go with Dask. Our reasoning was that, while our data was large (approx. 15 million records), it was not large enough to warrant the use of Spark. The image below summarizes the point.



# Data Preparation – Dask or Spark? Cont

- In addition to file size, we also wanted to use Dask because of its natural integration with python and pandas



# Data Preparation – Dask or Spark? Cont

- Our strategy was to align with Dask's best practices by using Dask to reduce the data, then switching to pandas when suitable

## Reduce, and then use Pandas

Similar to above, even if you have a large dataset there may be a point in your computation where you've reduced things to a more manageable level. You may want to switch to Pandas at this point.

```
df = dd.read_parquet('my-giant-file.parquet')
df = df[df.name == 'Alice']           # Select a subsection
result = df.groupby('id').value.mean() # Reduce to a smaller size
result = result.compute()             # Convert to Pandas dataframe
result...                             # Continue working with Pandas
```


<https://docs.dask.org/en/stable/dataframe-best-practices.html>

# Data Preparation – Dask or Spark? Cont

- Throughout this process, we leveraged the Dask dashboard to track memory usage

```
#get link to client dashboard  
client.dashboard_link
```

```
'http://127.0.0.1:8787/status'
```

Memory Use (%)													
													
name	address	nthreads	cpu	memory	limit	memory %	managed	unmanage	unmanage	spilled	# fds	read	write
Total (4)		8	2 %	3.1 GiB	7.8 GiB	40.1 %	0.0	3.1 GiB	0.0	0.0	120	27 KiB	39 KiB
0	tcp://127.0.0.1:8787	2	2 %	901.5 MiB	1.9 GiB	45.3 %	0.0	901.5 MiB	0.0	0.0	30	6 KiB	9 KiB
1	tcp://127.0.0.1:8787	2	2 %	730.8 MiB	1.9 GiB	38.7 %	0.0	730.8 MiB	0.0	0.0	30	6 KiB	9 KiB
2	tcp://127.0.0.1:8787	2	2 %	841.1 MiB	1.9 GiB	42.3 %	0.0	841.1 MiB	0.0	0.0	30	8 KiB	11 KiB
3	tcp://127.0.0.1:8787	2	2 %	715.0 MiB	1.9 GiB	35.9 %	0.0	715.0 MiB	0.0	0.0	30	7 KiB	10 KiB

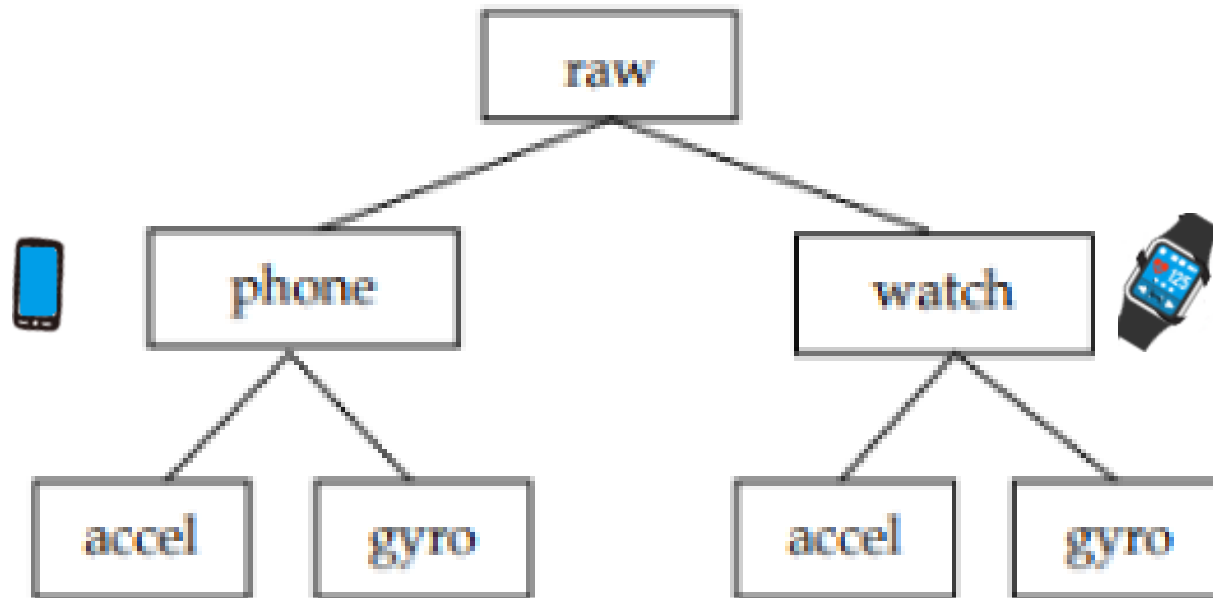
Team 001 - Sensor Data

# Data Preparation

Merging and Parquet

# Data Preparation – Merging

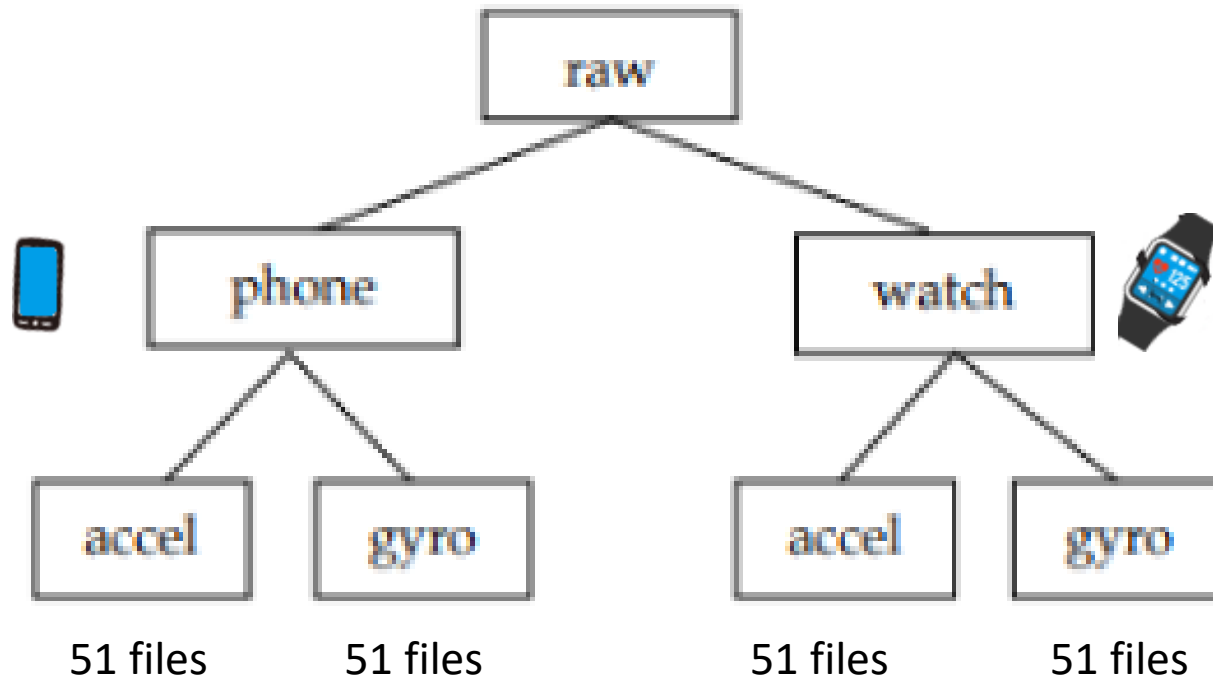
- The raw data is split up into 4 subdirectories, one for each device and sensor.





# Data Preparation – Merging

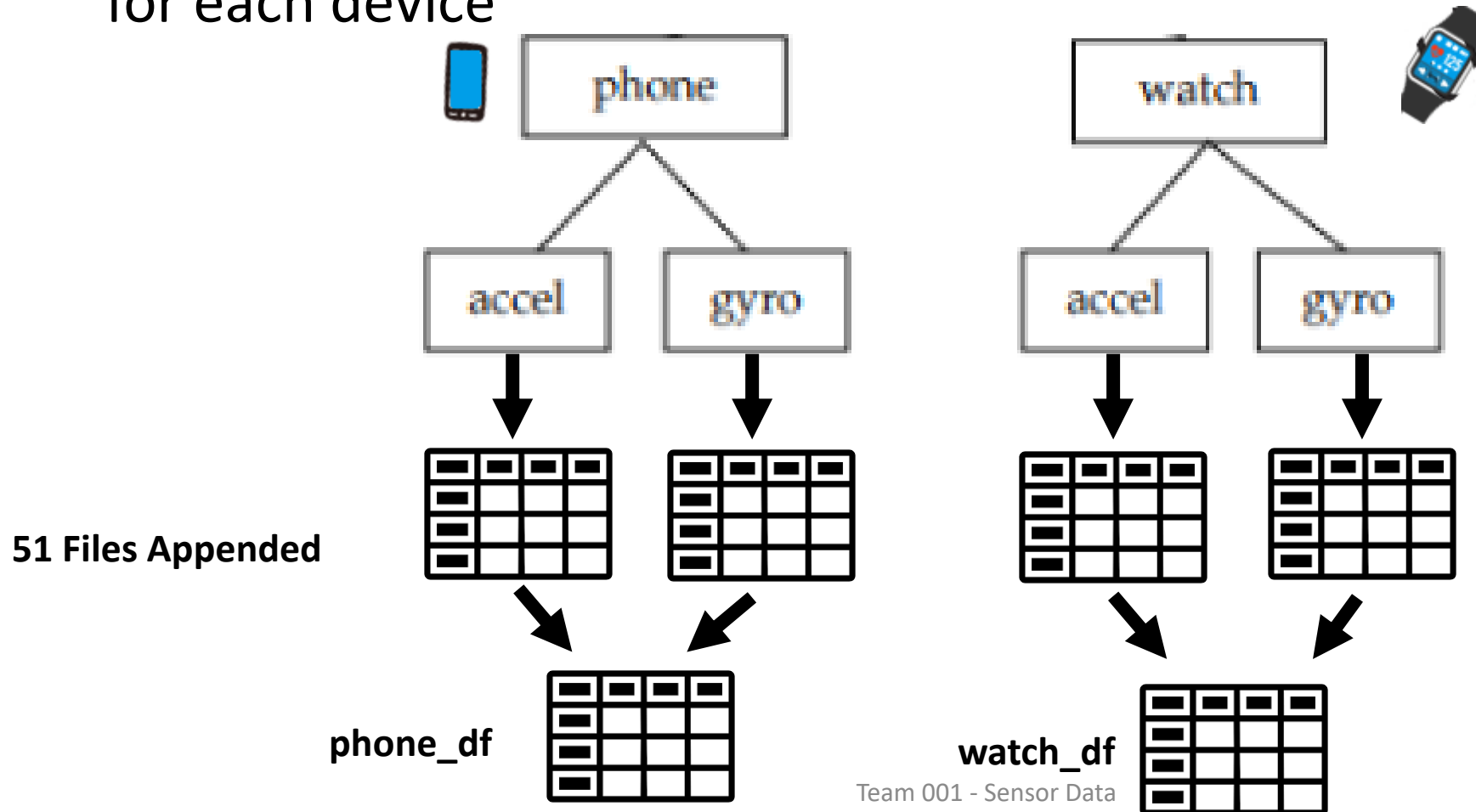
- Each directory contains the sensor results for the 51 subject's performance of the 18 activities.



**One file per  
subject, per folder**

# Data Preparation – Merging

- We decided to append all the files from each folder, and then merge for each device



# Data Preparation – Merging Problems

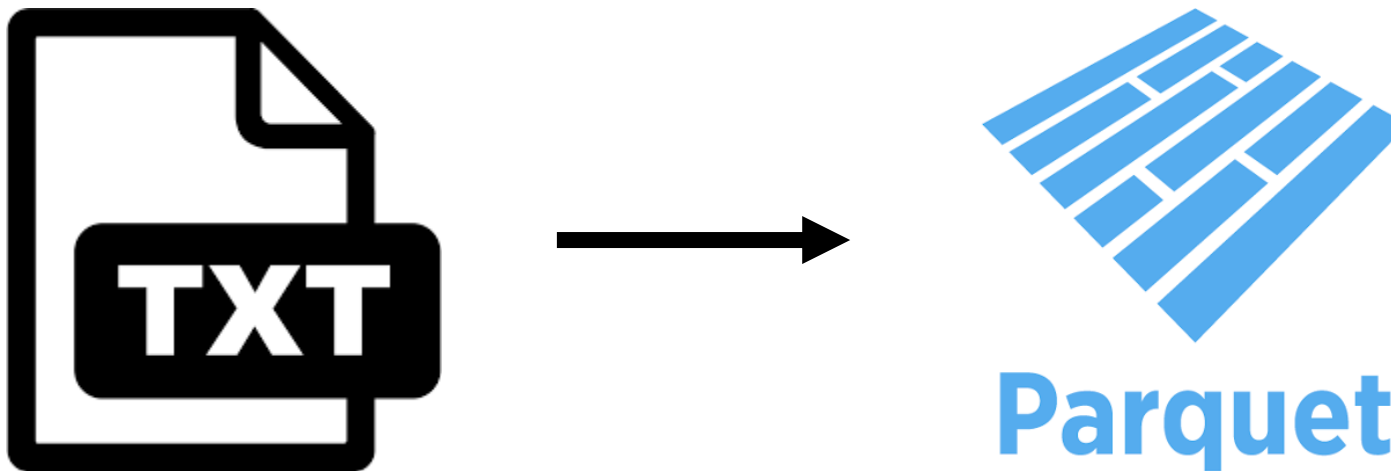
- While our logic was sound, our approach needed improvement. The reasons are as follows:
  1. Reading in the text files and appending was inefficient and slow
  2. We were wasting memory where we didn't need to
  3. Merging was a problem, because Dask is only efficient when data frames are merged on an index

# Data Preparation – Merging **Solution: Parquet**

- While our logic was sound, our approach needed improvement. The reasons are as follows:
  1. Reading in the text files and appending was inefficient and slow
    - **Solution:** Convert text files to parquet
  2. We were wasting memory where we didn't need to
    - **Solution:** Define a schema in the parquet files to use float32 and int16 columns instead of float64 and int64
  3. Merging was a problem, because Dask is only efficient when data frames are merged on an index
    - **Solution:** Make a custom index in parquet

# Data Preparation – Convert to Parquet

- Per Dask's best practices, we decided to convert our text files into parquet files. It was a little inconvenient to convert the original files to parquet, but it helped enormously in the long run



# Data Preparation – Convert to Parquet

- When we converted the text files to parquet, we
  1. Improved import time (~200 files in 4 seconds)
  2. Saved ~400mb.

raw data	858MB	<-- 2DFs
parquet	468MB	<--2DFs

Part of the memory saving was just a property of parquet, but we also explicitly defined the schema to use float32/int16. We did this by reading each text file with pandas, and using `pd. to_parquet`



# Data Preparation – Custom Indexes w/ Parquet

- The last thing we did in Parquet was define a custom index. This is because, per Dask's best practices, the best way to merge Dask dataframes is on the index.
- Originally, we were merging on three columns in the raw data which made our performance terrible

# Data Preparation – Custom Indexes w/ Parquet

- The last thing we did in Parquet was define a custom index. This is because, per Dask's best practices, the best way to merge files is on the index.
- We were merging on three columns in the raw data
- To leverage this, we made a custom column in the raw data and defined it



# Data Preparation – Custom Indexes w/ Parquet

- To leverage this, we made a custom field in the raw data, and set it as an index in Dask when we imported each Parquet file

```
df['index'] = df['subject_id'].astype('str') +  
df['code'] + df['timestamp'].astype('str')
```



```
dd.read_parquet(file, index = 'index')
```

Field name	Description
Subject-id	Type: Symbolic numeric identifier. Uniquely identifies the subject. Range: 1600-1650.
Activity code	Type: Symbolic single letter. Identifies a specific activity as listed in Table 2. Range: A-S (no "N" value)
Timestamp	Type: Integer. Linux time
x	Type Numeric: real. Sensor value for x axis. May be positive or negative.
y	Same as x but for y axis
z	Same as x but for z axis

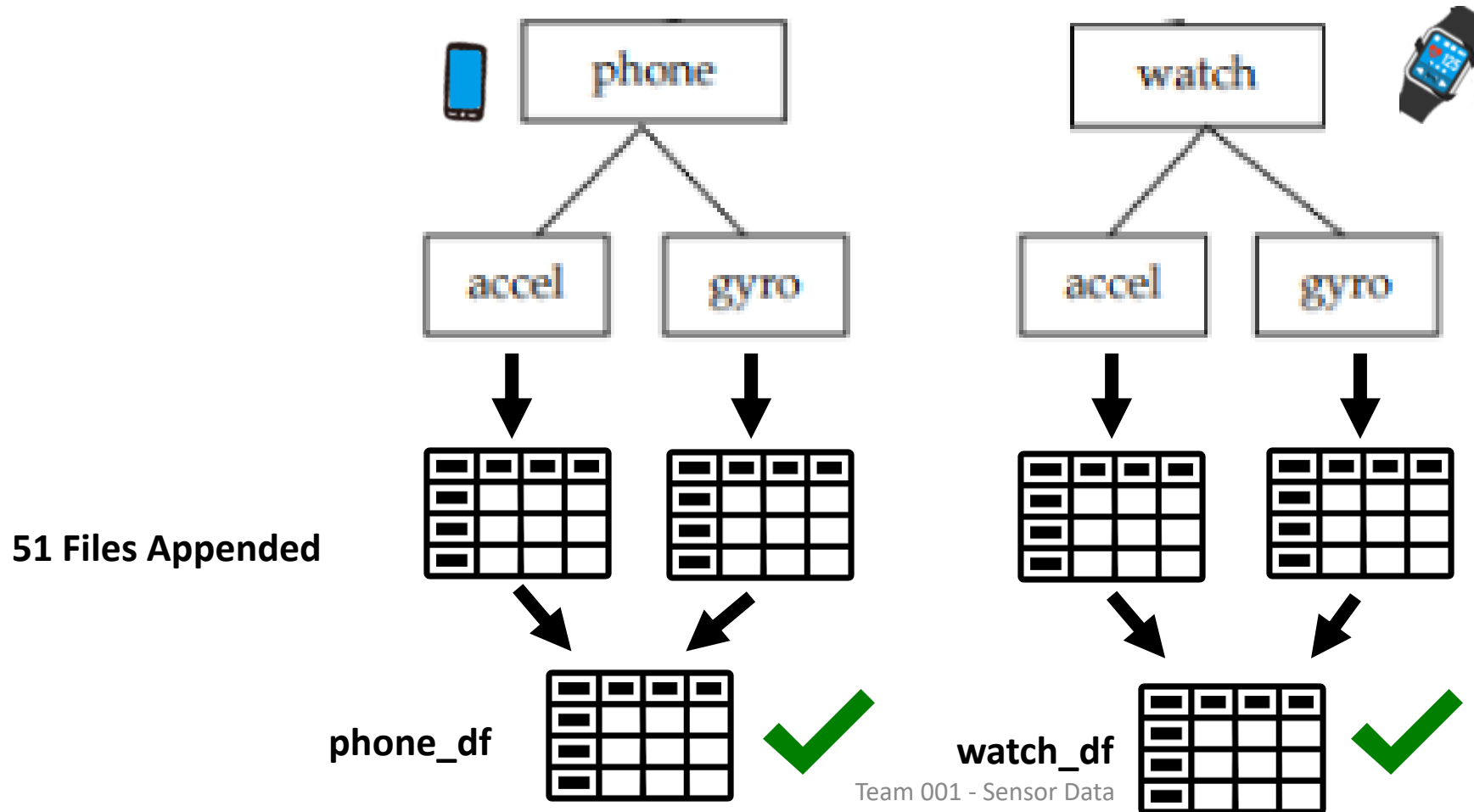
# Data Preparation – Merging w/ Parquet Files

- After all this work we reached our goals

```
def merge_dfs(df1, df2, suffixes):  
    df1partitions = df1.npartitions  
    df2partitions = df2.npartitions  
    partitions = min(df1partitions, df2partitions)  
    merged = dd.merge(  
        df1, df2[feat_cols], how='inner', left_index=True, right_index=True, suffixes=suffixes  
    ).reset_index(drop = True)  
    return dd.from_pandas(merged.compute(), npartitions = partitions)
```

# Data Preparation – Merging

- After all this work we reached our goals



# Data Preparation

Grouping and Cosine Similarity

# Data Preparation – Grouping

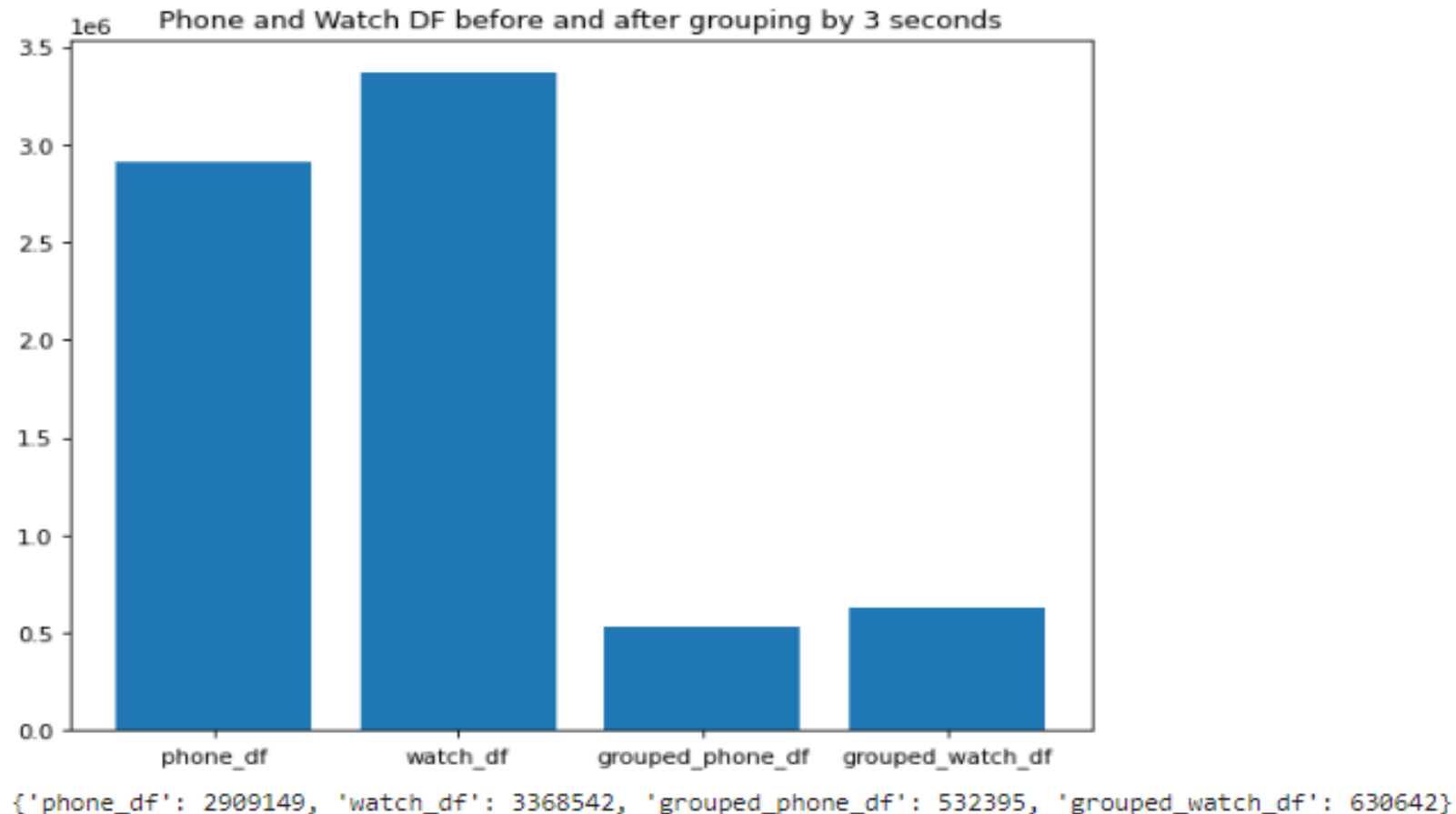
- For each sensor, data was recorded once every 50 ms (1/1000 second)
- For data reduction, we grouped all results into 3 second intervals, grouping by the subject and activity

# Data Preparation – Cosine Similarity

- Since data is recorded in x, y, z axis, we thought the cosine similarity might provide more insight
- Due to memory limitations/group by, we couldn't use Dask's built in functions
- We constructed the cosine similarity by hand. The steps were as follows:
  - Create new columns xy, yz, xz,  $x^2$ ,  $y^2$ ,  $z^2$
  - After group by, use the sum of each column to compute:
    - Cos Similarity for x and y =  $xy / \sqrt{x^2 * y^2}$

# Data Preparation – Grouping

- Grouping by 3 seconds allowed us to drastically reduce our data size



# From Dask to Pandas



# From Dask to Pandas

- When we started, the raw data was too large to fit in to memory
- We successfully leveraged Dask parallelization and reduced millions of rows to an aggregation useful for our classification models
- At this point, the overhead of Dask actually hurt runtime and our data was now at an appropriate size to use Pandas for ML models

# From Dask to Pandas

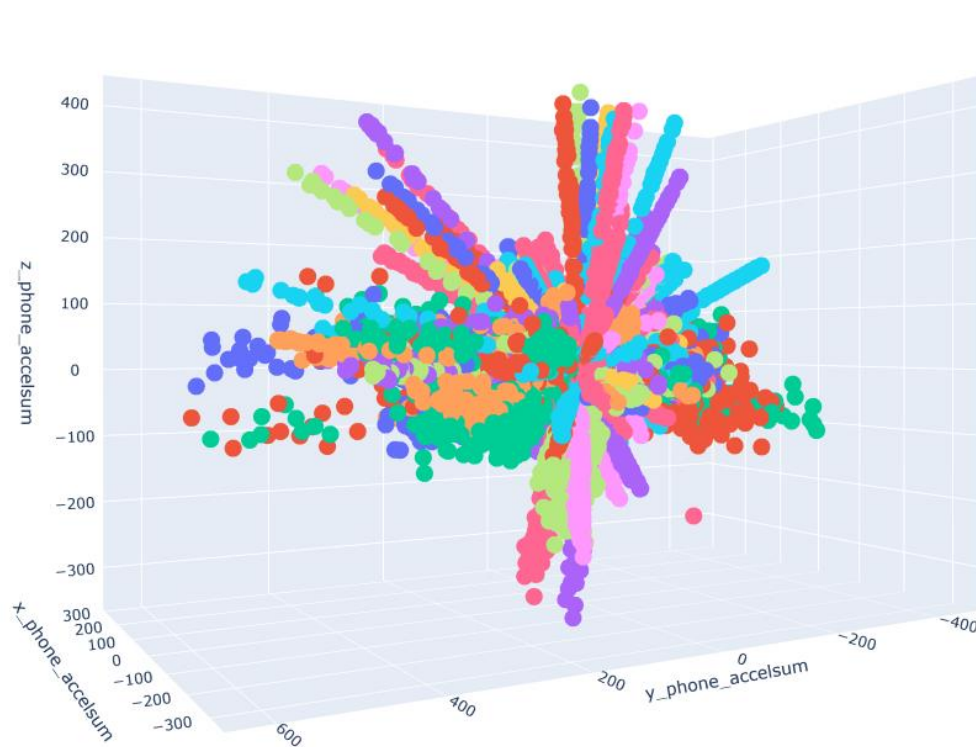
- Upper limit per pandas dataframe 100mb
  - Grouped phone 69mb
  - Grouped watch 82mb
- Adjusting pandas dtypes

raw data	858MB	<-- 2DFs
parquet	468MB	<--2DFs
final (grouped) data	151MB	<--2DFs

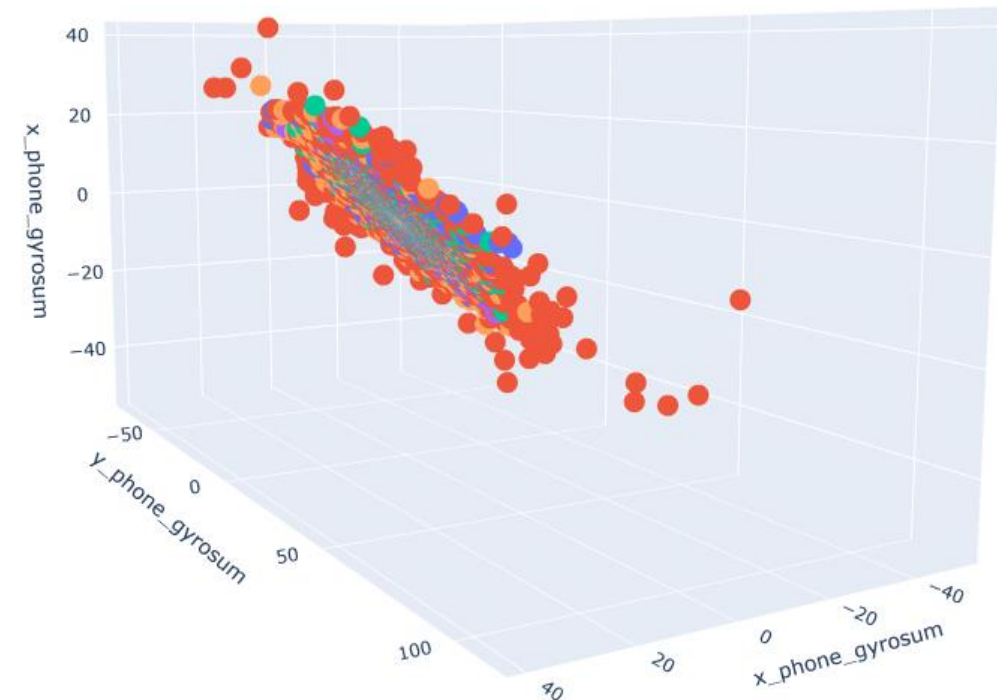
# EDA on Grouped Data

# Feature Visualization - Phone

## Phone Accelerometer

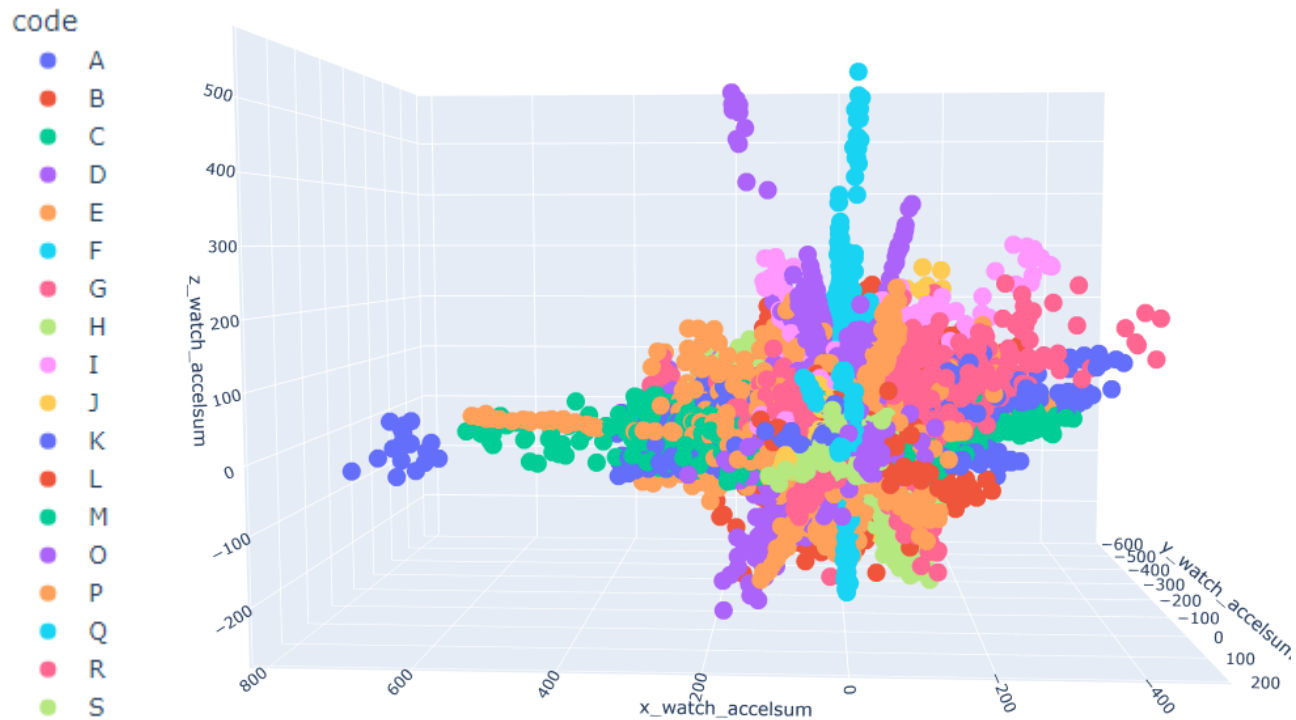


## Phone Gyroscope

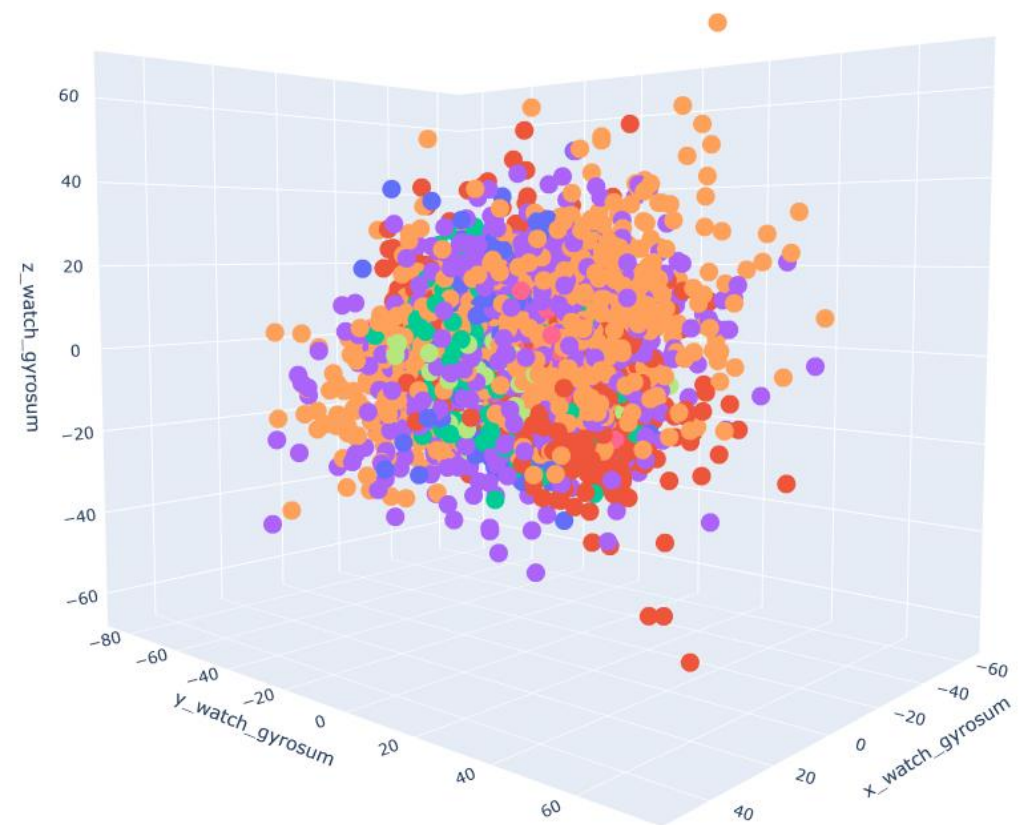


# Feature Visualization - Watch

## Watch Accelerometer

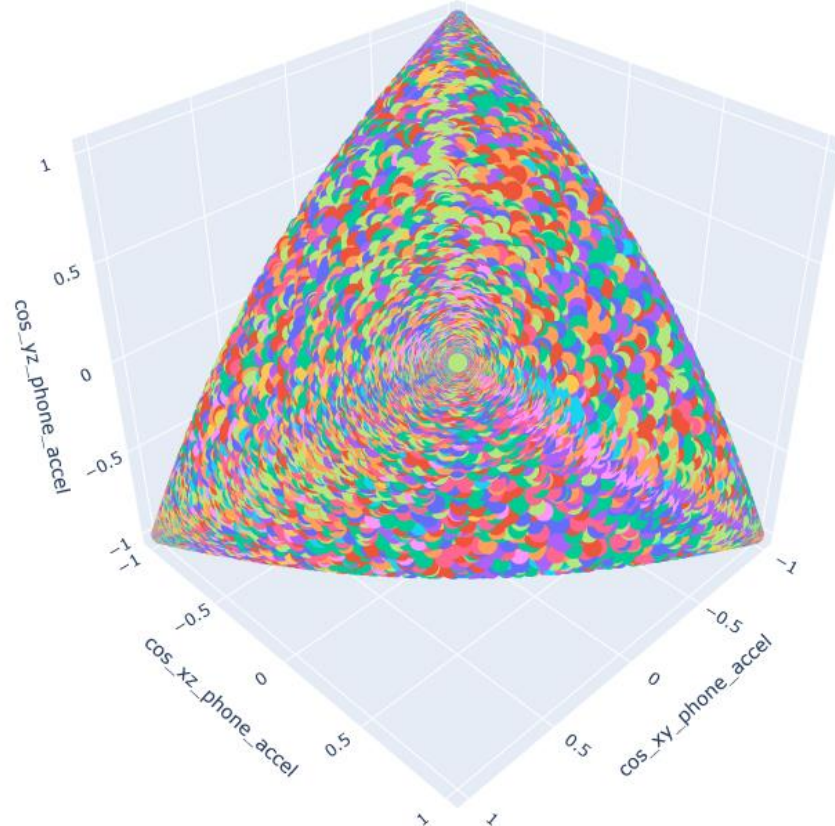


## Watch Gyroscope

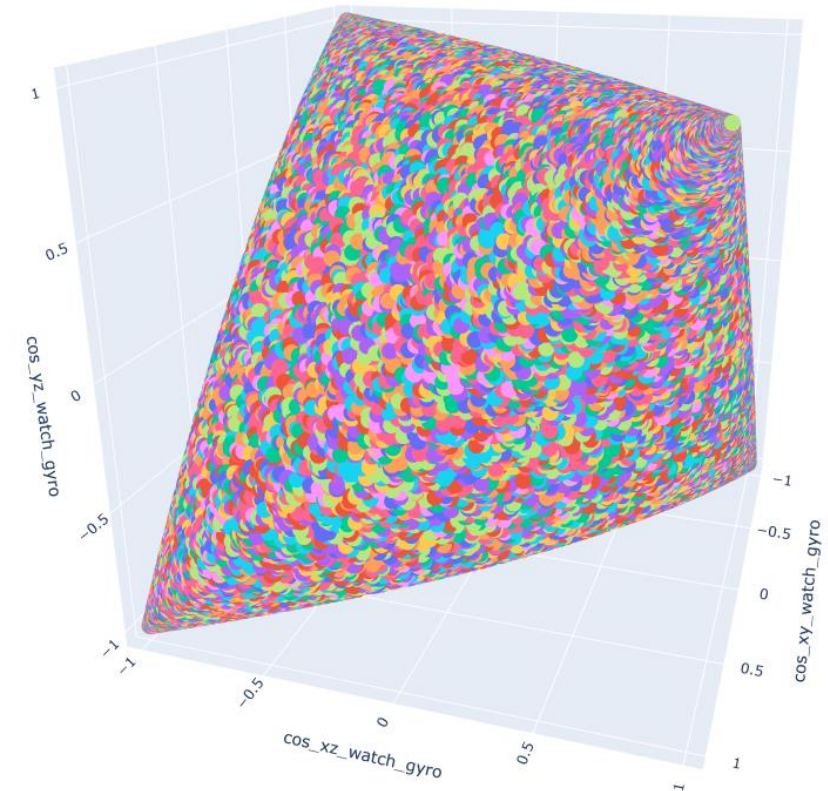


# Feature Visualization – Phone and Watch

Phone Accelerometer  
Cosine Similarity



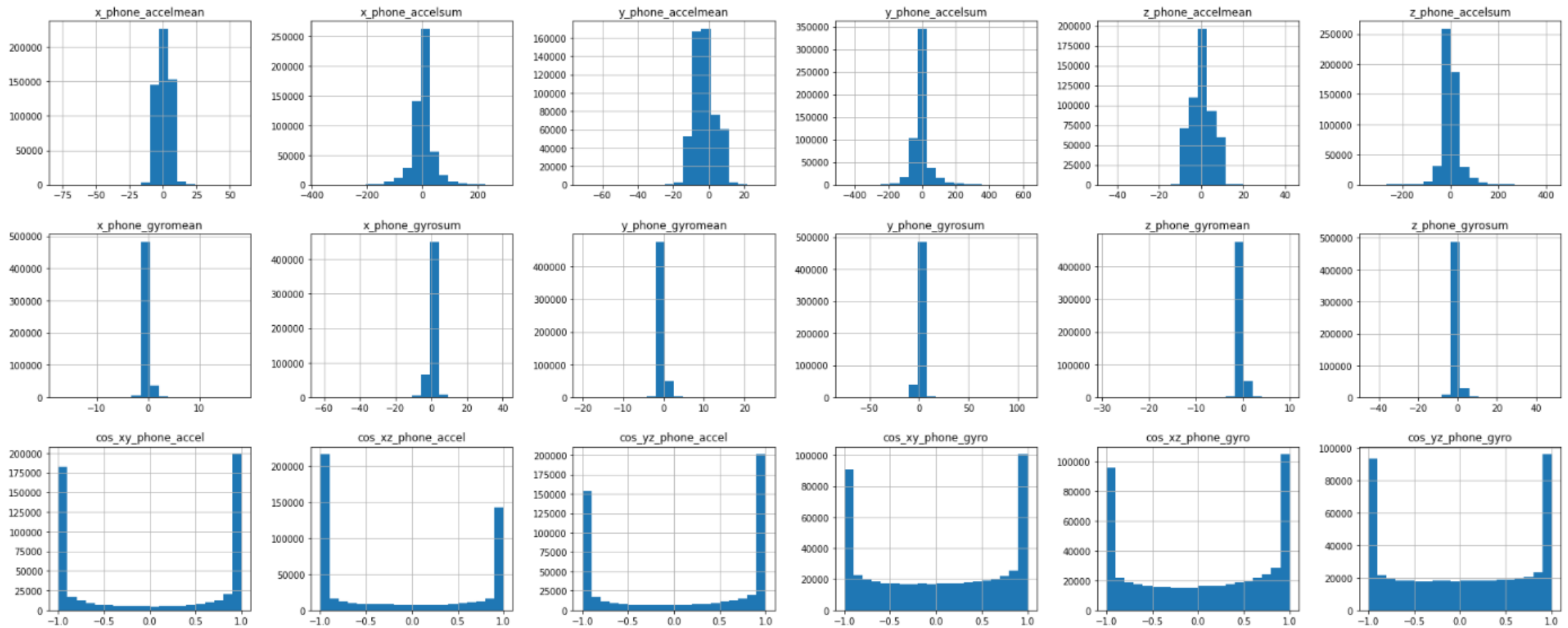
Watch Gyroscope  
Cosine Similarity



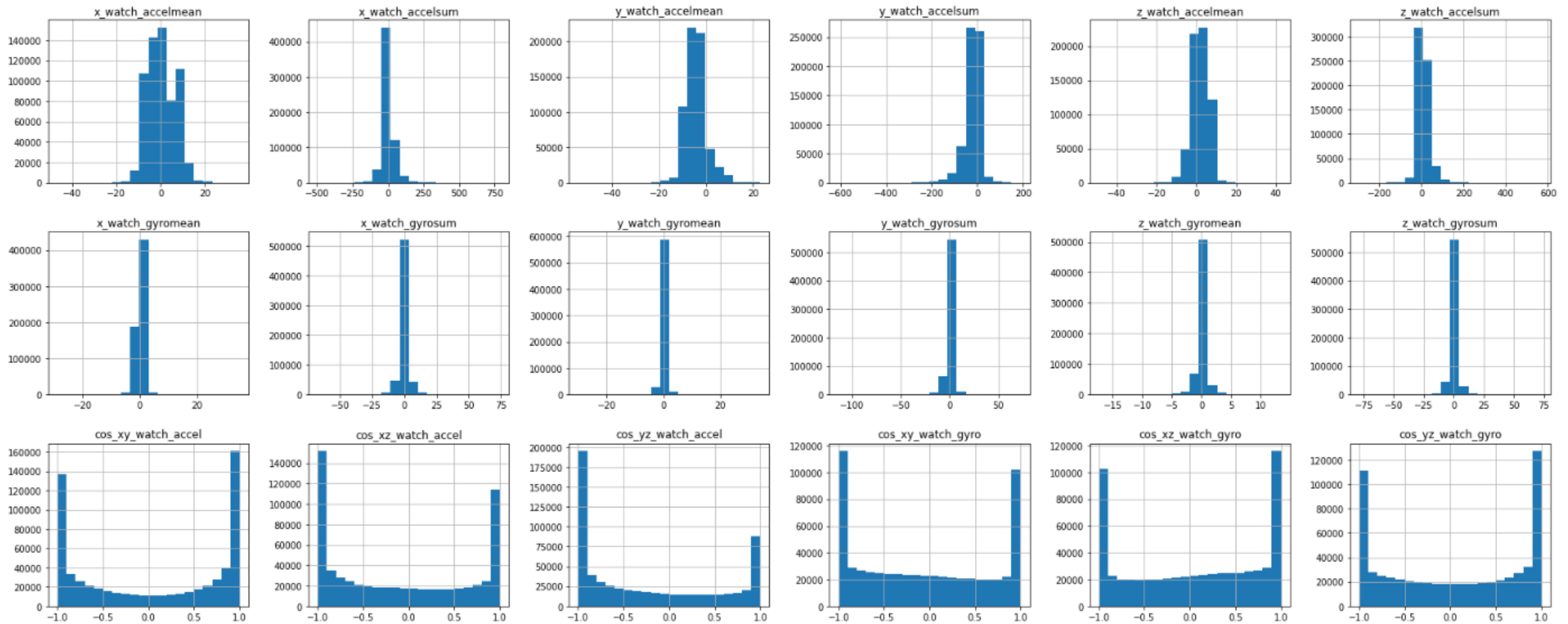
code

- A
- B
- C
- D
- E
- F
- G
- H
- I
- J
- K
- L
- M
- O
- P
- Q
- R
- S

# Distribution of Features - Phone



# Distribution of Features - Watch





# Analysis Approaches

Classification

# Analysis Approaches – Classification

- The dataset supports classification, with 18 possible outcomes
- We will predict classification on the phone and watch separately, since a user will most likely only have one device

THE 18 ACTIVITIES REPRESENTED IN DATA SET

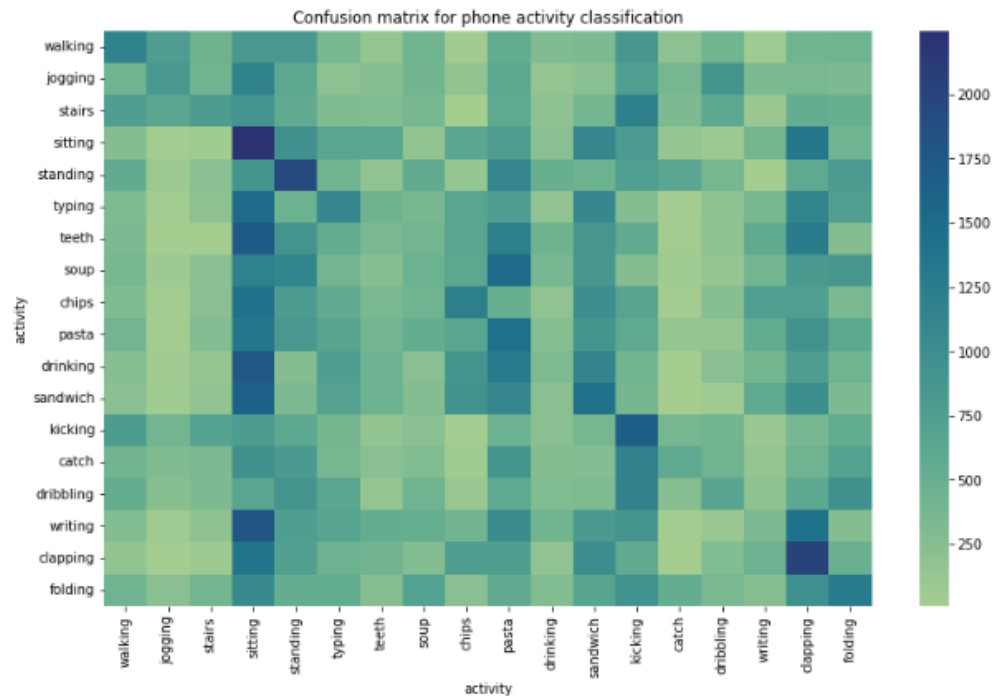
Activity	Code
Walking	A
Jogging	B
Stairs	C
Sitting	D
Standing	E
Typing	F
Brushing Teeth	G
Eating Soup	H
Eating Chips	I
Eating Pasta	J
Drinking from Cup	K
Eating Sandwich	L
Kicking (Soccer Ball)	M
Playing Catch w/Tennis Ball	O
Dribbling (Basketball)	P
Writing	Q
Clapping	R
Folding Clothes	S

# Analysis Approaches – Classification

- We tried an array of different classification methods
  - AdaBoost
  - Logistic Regression
  - Gaussian Naïve Bayes
  - KNN
  - Decision Trees
  - Random Forest

# Analysis Approaches – Logistic Regression

## Phone



```
%%time  
clf = LogisticRegression(random_state=seed)  
clf.fit(phone_x_train, phone_y_train)
```

CPU times: user 14.6 s, sys: 11.4 s, total: 26 s  
Wall time: 6.22 s

```
LogisticRegression(random_state=23)
```

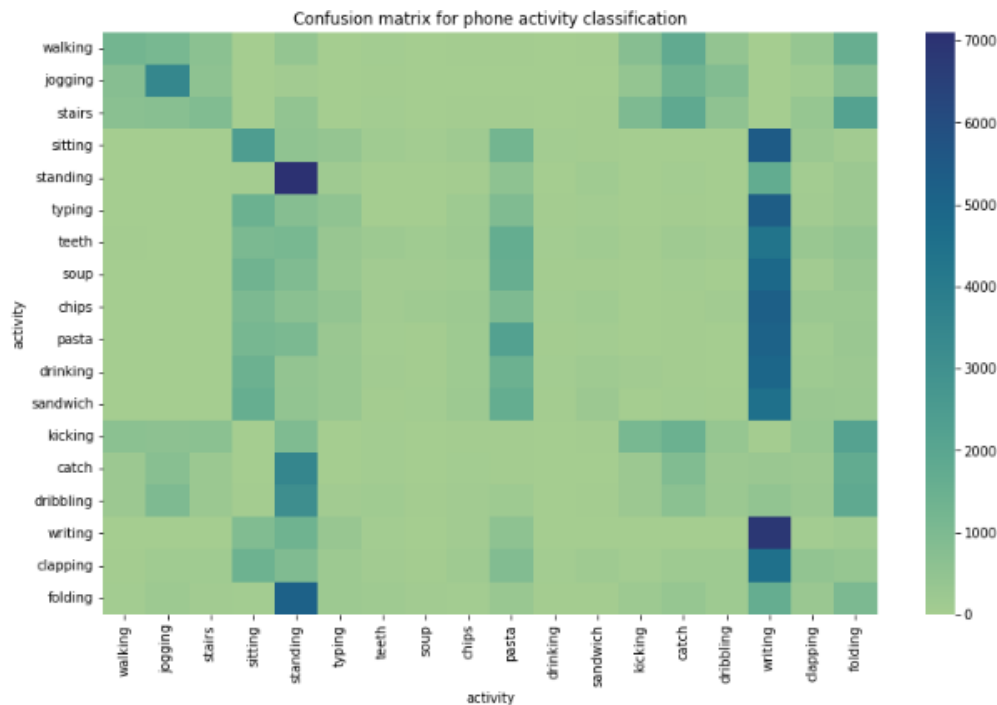
```
%%time  
clf.score(phone_x_test, phone_y_test)
```

CPU times: user 435 ms, sys: 418 ms, total: 853 ms  
Wall time: 174 ms

```
0.11350609877569141
```

# Analysis Approaches – Gaussian Naïve Bayes

## Phone



```
%%time  
gnb = GaussianNB()  
gnb.fit(phone_x_train, phone_y_train)
```

```
CPU times: user 434 ms, sys: 13.6 ms, total: 447 ms  
Wall time: 420 ms
```

```
GaussianNB()
```

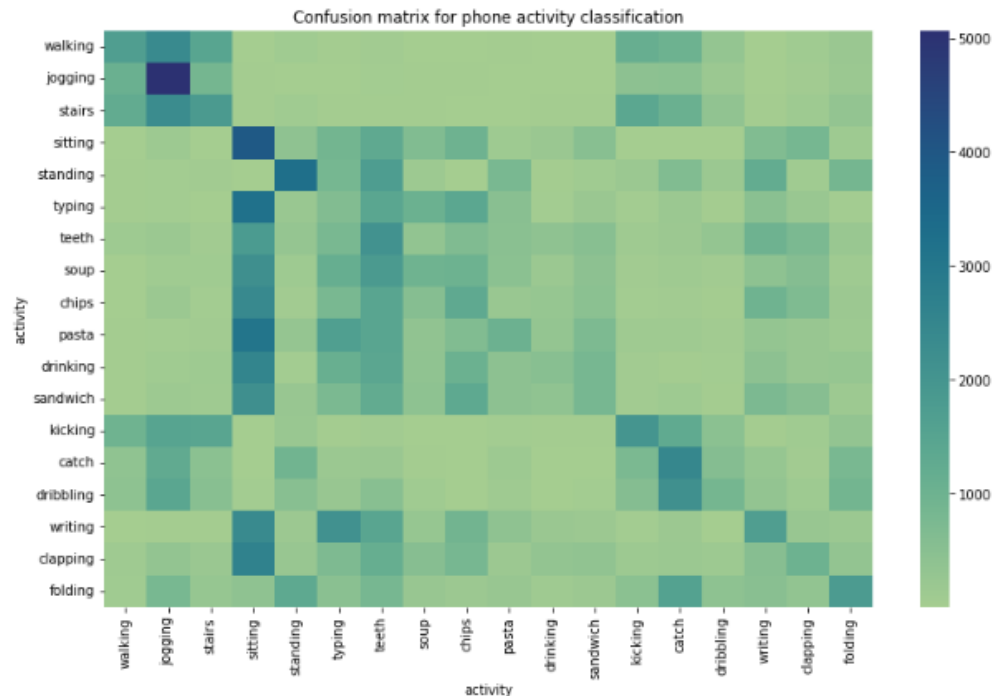
```
%%time  
gnb.score(phone_x_test, phone_y_test)
```

```
CPU times: user 274 ms, sys: 5.26 ms, total: 280 ms  
Wall time: 264 ms
```

```
0.16778890210653932
```

## Analysis Approaches – AdaBoost

### Phone



```
%%time  
clf = AdaBoostClassifier(n_estimators=100, random_state=seed)  
clf.fit(phone_x_train, phone_y_train)
```

```
CPU times: user 1min 55s, sys: 3.33 s, total: 1min 59s  
Wall time: 1min 53s
```

```
AdaBoostClassifier(n_estimators=100, random_state=23)
```

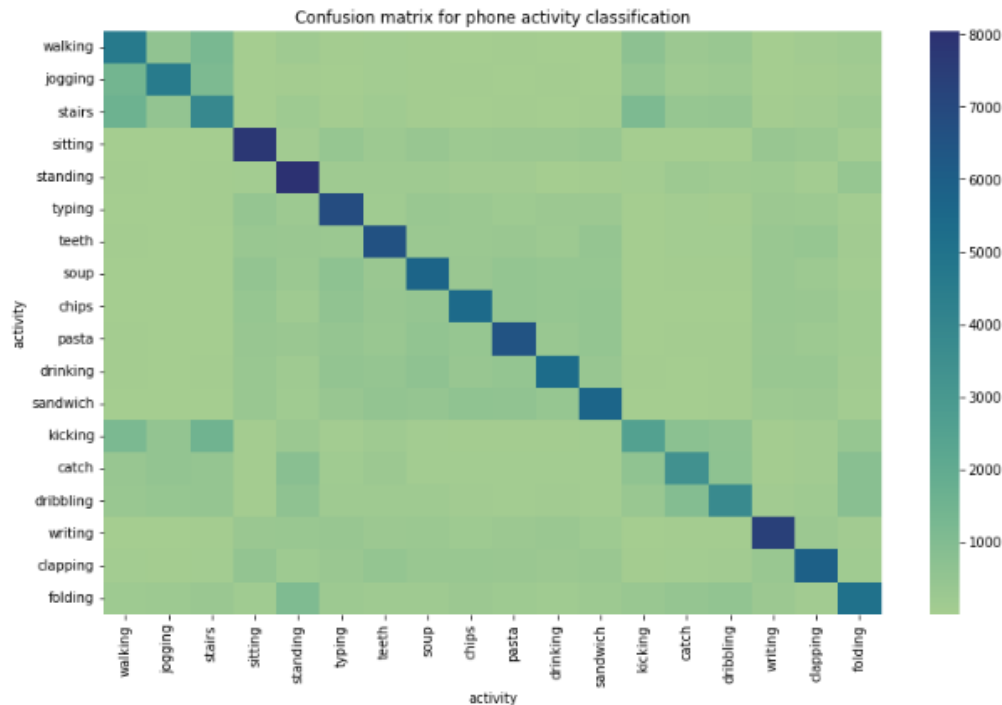
```
%%time  
clf.score(phone_x_test, phone_y_test)
```

```
CPU times: user 4.34 s, sys: 47.1 ms, total: 4.39 s  
Wall time: 4.16 s
```

```
0.18822819609427915
```

## Analysis Approaches – KNN

### Phone



```
%%time  
knn = KNeighborsClassifier(n_neighbors = 18, n_jobs = -1)  
knn.fit(phone_x_train, phone_y_train)
```

```
CPU times: user 284 ms, sys: 1.2 ms, total: 285 ms  
Wall time: 273 ms
```

```
KNeighborsClassifier(n_jobs=-1, n_neighbors=18)
```

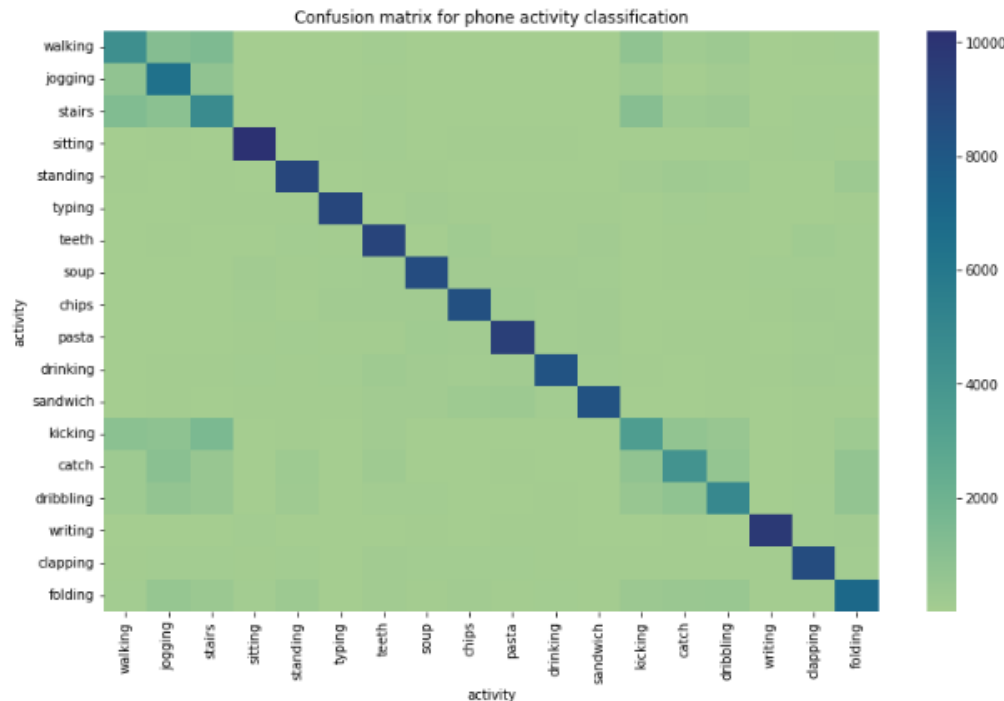
```
%%time  
knn.score(phone_x_test, phone_y_test)
```

```
CPU times: user 34min 8s, sys: 23min 44s, total: 57min 53s  
Wall time: 19min 53s
```

```
0.5639958791286975
```

# Analysis Approaches – Random Forest

## Phone



```
%%time  
clf = RandomForestClassifier(min_samples_leaf=20, random_state=seed)  
clf.fit(phone_x_train, phone_y_train)
```

CPU times: user 2min 15s, sys: 950 ms, total: 2min 16s

Wall time: 2min 9s

```
RandomForestClassifier(min_samples_leaf=20, random_state=23)
```

```
%%time  
clf.score(phone_x_test, phone_y_test)
```

CPU times: user 3.71 s, sys: 41.8 ms, total: 3.75 s

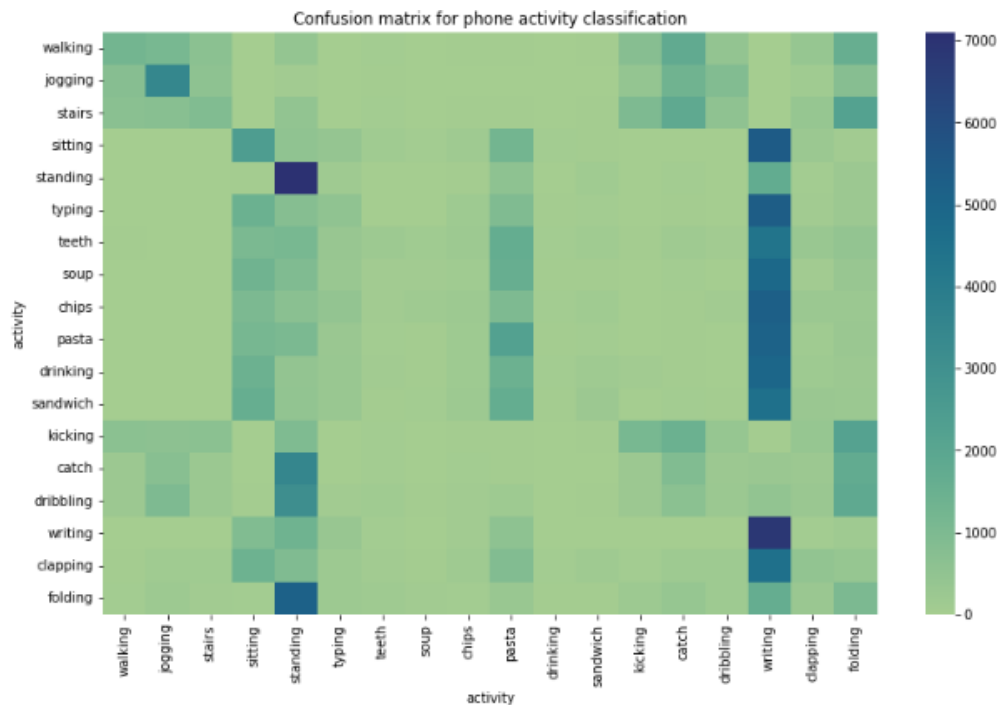
Wall time: 3.56 s

0.7644671610953322



# Analysis Approaches – Gaussian Naïve Bayes

## Watch



```
%%time  
gnb = GaussianNB()  
gnb.fit(phone_x_train, phone_y_train)
```

```
CPU times: user 434 ms, sys: 13.6 ms, total: 447 ms  
Wall time: 420 ms
```

```
GaussianNB()
```

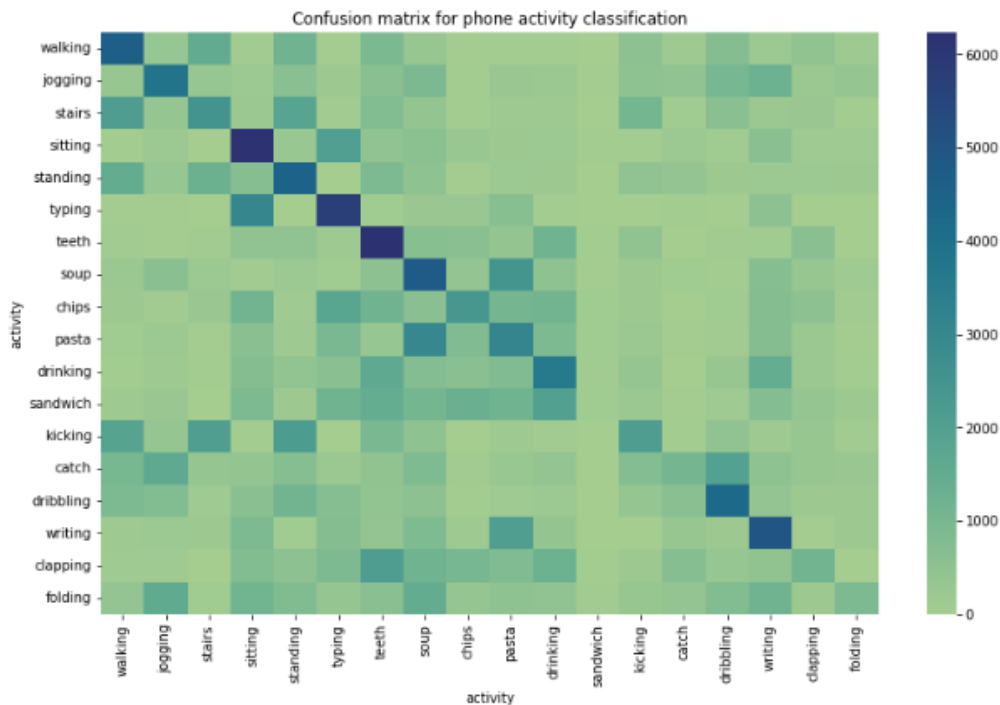
```
%%time  
gnb.score(phone_x_test, phone_y_test)
```

```
CPU times: user 274 ms, sys: 5.26 ms, total: 280 ms  
Wall time: 264 ms
```

```
0.16778890210653932
```

## Analysis Approaches – Logistic Regression

### Watch



```
%%time  
clf = LogisticRegression(random_state=seed)  
clf.fit(watch_x_train, watch_y_train)
```

```
CPU times: user 29.5 s, sys: 22 s, total: 51.5 s  
Wall time: 12.3 s
```

```
LogisticRegression(random_state=23)
```

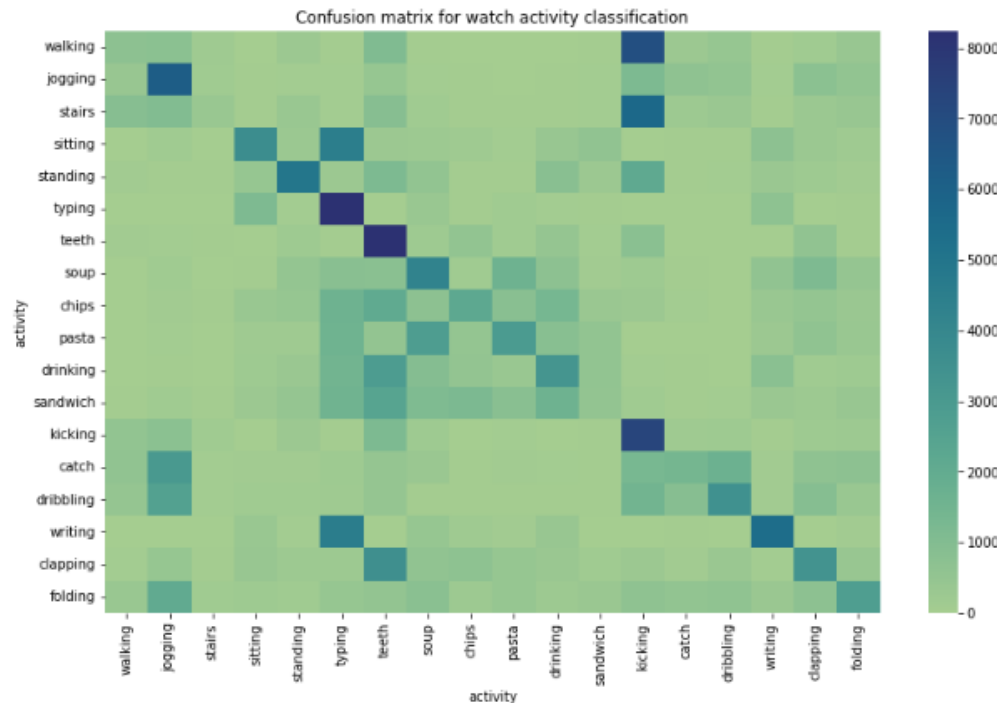
```
%%time  
clf.score(watch_x_test, watch_y_test)
```

```
CPU times: user 480 ms, sys: 420 ms, total: 901 ms  
Wall time: 201 ms
```

```
0.29865168755285615
```

## Analysis Approaches – AdaBoost

### Watch



```
%%time  
clf = AdaBoostClassifier(n_estimators=100, random_state=seed)  
clf.fit(watch_x_train, watch_y_train)
```

CPU times: user 2min 23s, sys: 3.1 s, total: 2min 26s  
Wall time: 2min 21s

```
AdaBoostClassifier(n_estimators=100, random_state=23)
```

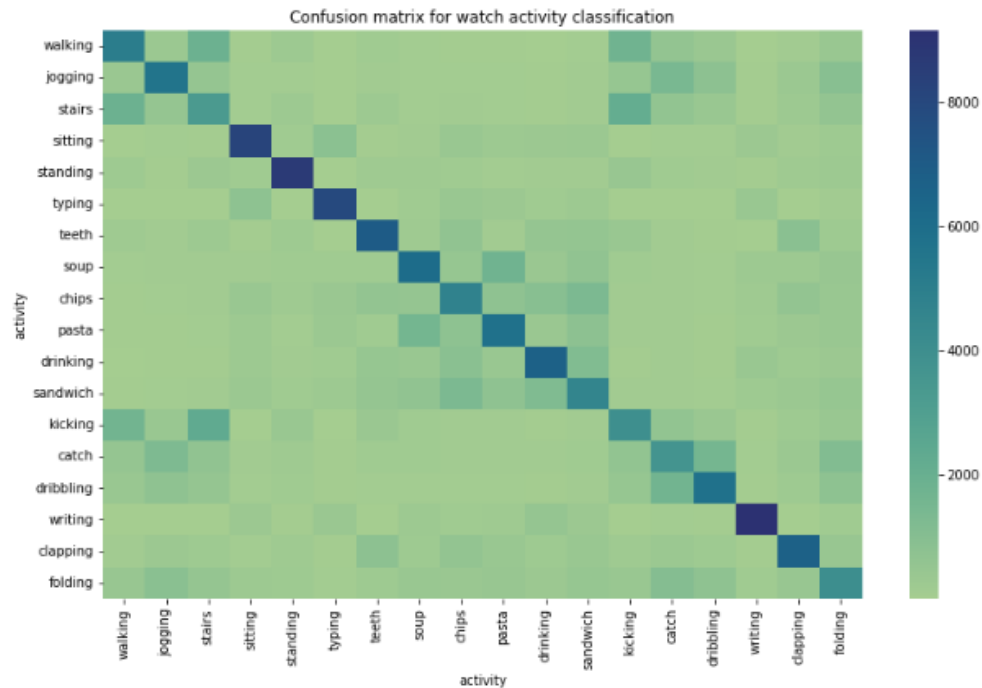
```
%%time  
clf.score(watch_x_test, watch_y_test)
```

CPU times: user 5.03 s, sys: 40.3 ms, total: 5.07 s  
Wall time: 4.89 s

```
0.3204764742061967
```

# Analysis Approaches – Decision Tree

## Watch



```
%%time  
clf = DecisionTreeClassifier(random_state=seed)  
clf.fit(watch_x_train, watch_y_train)
```

```
CPU times: user 14.8 s, sys: 111 ms, total: 15 s  
Wall time: 14.2 s
```

```
DecisionTreeClassifier(random_state=23)
```

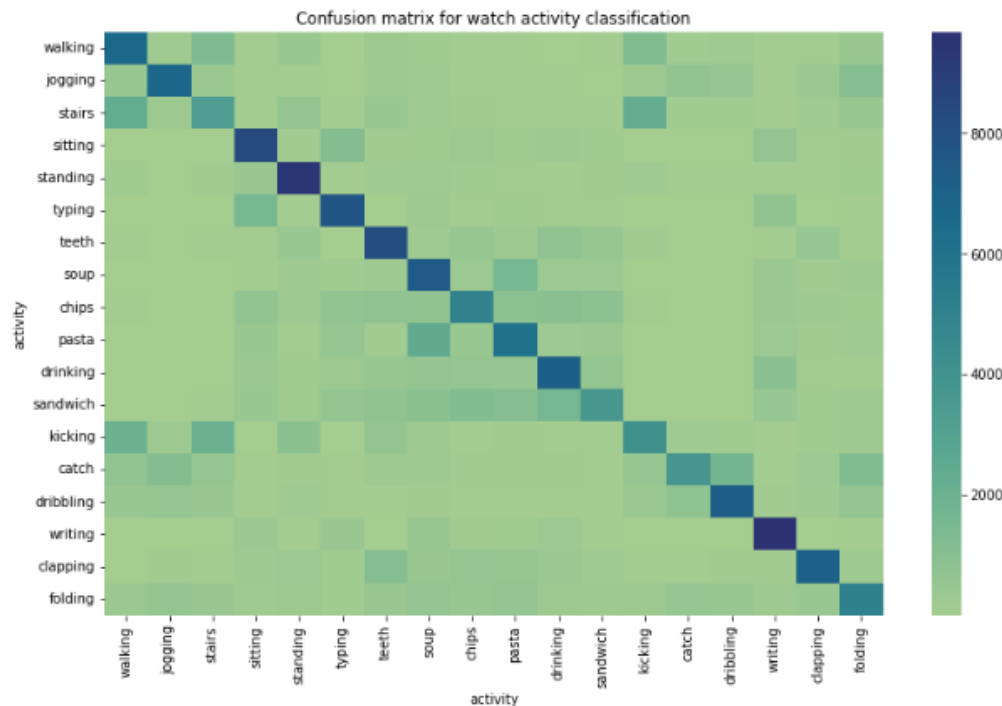
```
%%time  
clf.score(watch_x_test, watch_y_test)
```

```
CPU times: user 242 ms, sys: 6.21 ms, total: 249 ms  
Wall time: 236 ms
```

```
0.5153715307142308
```

## Analysis Approaches – KNN

### Watch



```
%%time  
knn = KNeighborsClassifier(n_neighbors = 18, n_jobs = -1)  
knn.fit(watch_x_train, watch_y_train)
```

```
CPU times: user 325 ms, sys: 0 ns, total: 325 ms  
Wall time: 318 ms
```

```
KNeighborsClassifier(n_jobs=-1, n_neighbors=18)
```

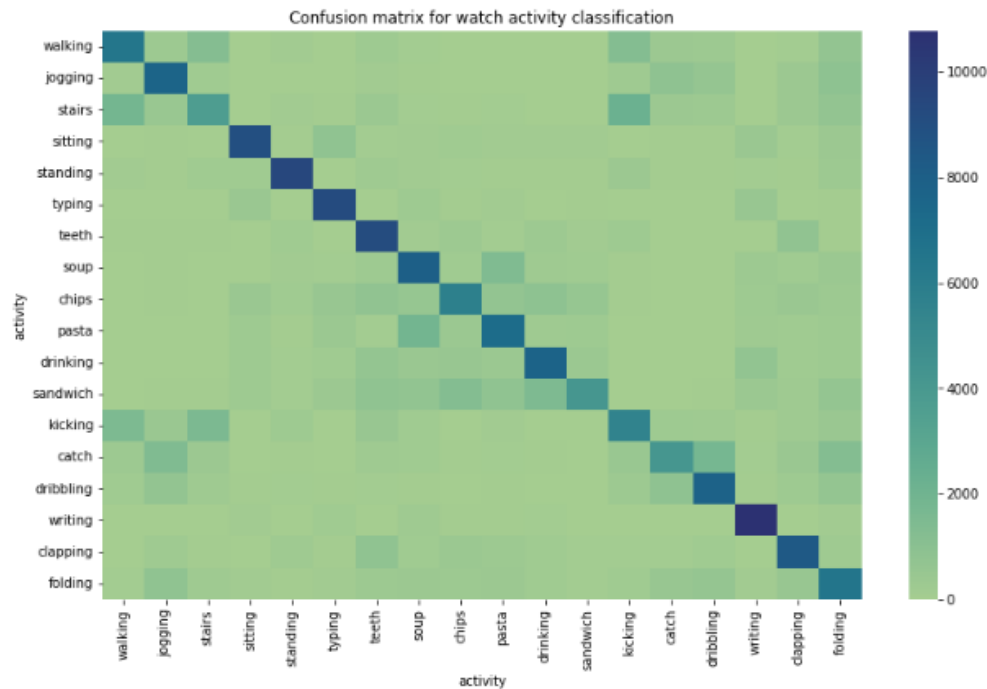
```
%%time  
knn.score(watch_x_test, watch_y_test)
```

```
CPU times: user 47min 59s, sys: 33min 30s, total: 1h 21min 29s  
Wall time: 28min 58s
```

```
0.5613515799185054
```

# Analysis Approaches – Random Forest

## Watch



```
%%time  
clf = RandomForestClassifier(min_samples_leaf=20, random_state=seed)  
clf.fit(watch_x_train, watch_y_train)
```

CPU times: user 2min 46s, sys: 1.17 s, total: 2min 47s

Wall time: 2min 38s

```
RandomForestClassifier(min_samples_leaf=20, random_state=23)
```

```
%%time  
clf.score(watch_x_test, watch_y_test)
```

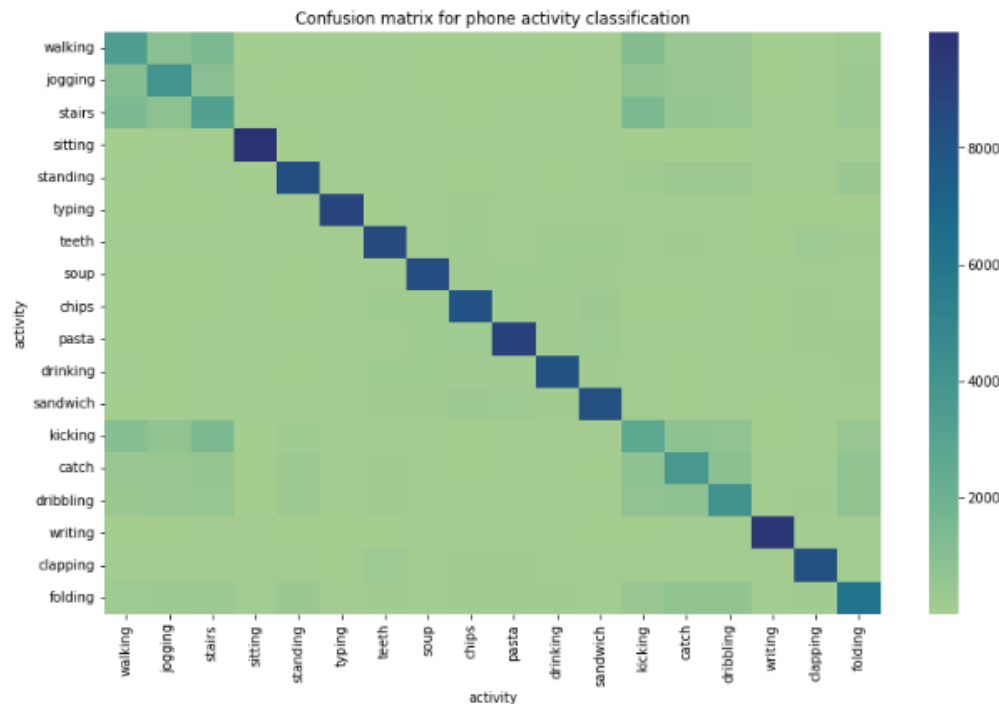
CPU times: user 4.58 s, sys: 45.7 ms, total: 4.63 s

Wall time: 4.4 s

0.6285317521334666

# Analysis Approaches – Decision Tree

## Phone



```
%%time  
clf = DecisionTreeClassifier(random_state=seed)  
clf.fit(phone_x_train, phone_y_train)
```

```
CPU times: user 11.2 s, sys: 87.7 ms, total: 11.3 s  
Wall time: 10.7 s
```

```
DecisionTreeClassifier(random_state=23)
```

```
%%time  
clf.score(phone_x_test, phone_y_test)
```

```
CPU times: user 185 ms, sys: 4.83 ms, total: 189 ms  
Wall time: 180 ms
```

```
0.7035306304819257
```

## Analysis Approaches – Decision Tree



- The most accurate model we fit used the Phone Accelerometer and Phone Gyroscope using a set with all of the features, including the x-y-z cosine similarities
- Surprisingly, the watch did not return as accurate of a model

### Decision Tree Accuracy Table

device	DT_score	features
phone	0.745393	all
phone	0.742124	mean
phone	0.685264	sum
watch	0.585466	all
watch	0.579744	mean
watch	0.544810	sum
phone	0.212609	cos
watch	0.204621	cos

### Features

RangeIndex: 532395 entries, 0 to 532394				
Data columns (total 43 columns):				
#	Column	Non-Null Count	Dtype	
0	code	532395 non-null	category	
1	x_phone_accelmean	532395 non-null	float64	
2	x_phone_accelsum	532395 non-null	float64	
3	y_phone_accelmean	532395 non-null	float64	
4	y_phone_accelsum	532395 non-null	float64	
5	z_phone_accelmean	532395 non-null	float64	
6	z_phone_accelsum	532395 non-null	float64	
7	xy_phone_accelmean	532395 non-null	float64	
8	xy_phone_accelsum	532395 non-null	float64	
9	yz_phone_accelmean	532395 non-null	float64	
10	yz_phone_accelsum	532395 non-null	float64	
11	xz_phone_accelmean	532395 non-null	float64	
12	xz_phone_accelsum	532395 non-null	float64	
13	x2_phone_accelmean	532395 non-null	float64	
14	x2_phone_accelsum	532395 non-null	float64	
15	y2_phone_accelmean	532395 non-null	float64	
16	y2_phone_accelsum	532395 non-null	float64	
17	z2_phone_accelmean	532395 non-null	float64	
18	z2_phone_accelsum	532395 non-null	float64	
19	x_phone_gyromean	532395 non-null	float64	
20	x_phone_gyrosum	532395 non-null	float64	
21	y_phone_gyromean	532395 non-null	float64	
22	y_phone_gyrosum	532395 non-null	float64	
23	z_phone_gyromean	532395 non-null	float64	
24	z_phone_gyrosum	532395 non-null	float64	
25	xy_phone_gyromean	532395 non-null	float64	
26	xy_phone_gyrosum	532395 non-null	float64	
27	yz_phone_gyromean	532395 non-null	float64	
28	yz_phone_gyrosum	532395 non-null	float64	
29	xz_phone_gyromean	532395 non-null	float64	
30	xz_phone_gyrosum	532395 non-null	float64	
31	x2_phone_gyromean	532395 non-null	float64	
32	x2_phone_gyrosum	532395 non-null	float64	
33	y2_phone_gyromean	532395 non-null	float64	
34	y2_phone_gyrosum	532395 non-null	float64	
35	z2_phone_gyromean	532395 non-null	float64	
36	z2_phone_gyrosum	532395 non-null	float64	
37	cos_xy_phone_accel	532395 non-null	float64	
38	cos_xz_phone_accel	532395 non-null	float64	
39	cos_yz_phone_accel	532395 non-null	float64	
40	cos_xy_phone_gyro	532395 non-null	float64	
41	cos_xz_phone_gyro	532395 non-null	float64	
42	cos_yz_phone_gyro	532395 non-null	float64	



# Challenges and Approaches

# Challenges and Approaches

- Constant memory spillage in Dask
  - Solution: Used the Dask dashboard, used Parquet, reduced datatypes, canceled Dask objects when they were not needed, reduced data to leverage Pandas
- Grouping by 3 seconds might not be the best aggregation
  - Group by 1, 3, 5, 10 second intervals
- Leveraging the results from the phone vs watch predictions
  - The phone seemed to react better to the models, so we will have to give that a higher weight when predicting if two devices are involved

# Insights Gained

# Insights Gained

- Dask can be a very handy tool when used correctly. When used incorrectly, it can be much more difficult than pandas. Some lessons learned
  - Keep an eye on memory usage
  - Don't compute or persist until you need to
  - Take advantage of parquet files and schema design (data type selection)
  - Remove objects from memory when they're not used
  - Work with a sample of the data while writing initial code

# Future Work

# Future Work

- Dimensionality Reduction
- Model Tuning
- Aggregation Tuning