Tutorial

Start with raw data | Start with balanced dataset

Before start...

All the pieces of code you will find in this guide are documented in our Github guide where you can also download our code. Below is the link to our repository:

Transportation Mode Detection with Unconstrained Smartphones Sensors (https://github.com/vlomonaco/US-TransportationMode)

Start with raw data

Before using this guide, you have to download raw data (static/dataset /raw_data/raw_data.tar.gz) or you can build your own dataset or expand our using this app (static/apk/SensorCollector.apk) (for bugs,send an email to developers. Contacts here (aboutus.html)).

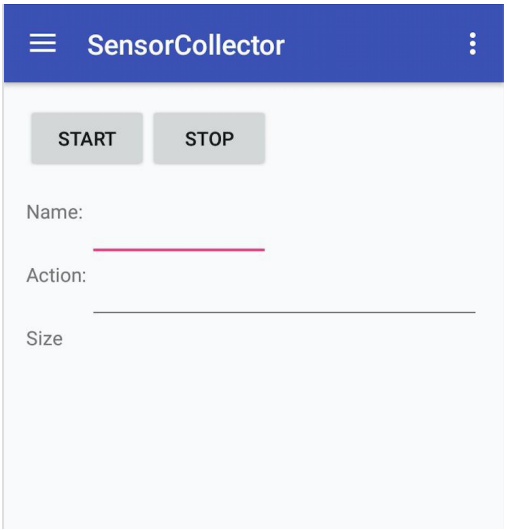

Fig 1: Initial screen of the application used to collect data (Sensor collector)

Data collection was controlled by an Android application running on the user's phone as they performed activities. This application, through a simple graphical user interface, shown in Figure 1, permitted volunteers to record

their name, start and stop the data collection, and label the activity being performed. We asked users to use the application during specific activities such as walking, being on a car or on a train or on a bus or standing still. Above we refer to activities with this abbreviations:
T M = {bus, car, train, still, walking}
The application registers each sensor event with a maximum frequency of 20 Hz. Events occurs every time a sensor detects a change in the parameters it is measuring, providing four pieces of information:

- the name of the sensor that triggered the event;
- the timestamp for the event;
- the accuracy of the event;
- the raw sensor data that triggered the event.

The length and content of the raw sensor data depend on the type of sensor. Each raw data registered has the following structure:
$< timestamp, sensor_i , sensorOutput_i >$
The application saves each sample on a csv file on the device. The maximum frequency of sampling (20Hz) ensures small file size suitable to be stored in a smartphone. The file's data was collected directly from files stored on the phones via a USB connection.
Result files contained **raw data** from twenty-three sensors. Twenty-three is the number of sensors of which we collect data (we will see that not all will be used for the model though). Instead of deciding before hands which sensors to use based on the task being performed, we choose to collect data from all sensors available on the device and then we decide which sensor is good to keep.
In our study we exclude sensors that are not supported by enough devices (support is too low), and sensor that collect data that can me too misleading for the actual classification of the activity.

After collecting our data or after downloading the dataset, we are ready to start with the creation of the balanced dataset.

```
if __name__ == "__main__":
        dataset = TMDataset()
        dataset.create_balanced_dataset()
```

We analyze dataset composition in term of class and user contribution and we fill balance time with minimum number of **window** for transportation mode. In order to facilitate the processing of data, the samples need to be cut, namely windowed. The size of the time window depends on the types of actions to be recognized.
To do this, within our **create_balanced_dataset** function, we call the **create_dataset** function: this function is intended to generate a single dataset starting by joining all the raw data.

```
def __create_dataset(self):
        [...]
        if self.sintetic:
        # create files with time window if not exsist
        if not os.path.exists(dir_src):
            self.__create_time_files()

        if not os.path.exists(dir_dst):
            os.makedirs(dir_dst)
        else:
            shutil.rmtree(dir_dst)
            os.makedirs(dir_dst)

        filenames = listdir(dir_src)

        result_file_path = os.path.join(dir_dst, file_dst)
        with open(result_file_path, 'w') as result_file:
            j = 0
            for file in filenames:
                if file.endswith(".csv"):
                    current_file_path = os.path.join(dir_src, file)
                    with open(current_file_path) as current_file:  # tm file
                        i = 0
                        for line in current_file:
                            # if the current line is not the first, the header
                            if i != 0:
                                result_file.write(line)
                            else:
                                if j == 0:
                                    result_file.write(line)
                                i += 1
                        j += 1
```

Within the previous function shown, we notice the presence of the function **__create_time_files** which allows us to divide our files into time windows and to go and calculate the features. We generated statistical features based on the multiple raw sensor readings. For each sensor we generate 4 different features:

- maximum
- minimum
- mean
- standard deviation

```
def __create_time_files(self):
    [...]
        # define time range
        end_current = start_current + window_dim
        if end_time <= end_current:
            range_current = list(range(start_current, end_time, 1))
            start_current = end_time
        else:
            range_current = list(range(start_current, end_current, 1))
            start_current = end_current
        # df of the current time window
        df_current = df_file.loc[df_file['time'].isin(range_current)]
        nfeature = 0
        if self.sintetic:
            if df_current.loc[:, "target"].size > 0:
                df_current_tm = df_current.loc[:, "target"]
                current_user = df_current.loc[:, "user"].iloc[0]
                equal = True
                for tm in range(0,df_current_tm.size-1,1):
                    if not df_current_tm.iloc[tm] == df_current_tm.iloc[tm+1]:
                        equal = False
                        break

                if equal == False:
                    continue
                else:
                    current_tm = df_current_tm.iloc[0]

        currentLine = ""
        for feature in featureNames:
            currentFeatureSerie = df_current[feature]
            currentMean = currentFeatureSerie.mean(skipna=True)
            currentMin = currentFeatureSerie.min(skipna=True)
            currentMax = currentFeatureSerie.max(skipna=True)
            currentStd = currentFeatureSerie.std(skipna=True)
            if i == 0:
                previous_mean.append(str(currentMean))
                current_mean.append(str(currentMean))
                previous_min.append(str(currentMin))
                current_min.append(str(currentMin))
                previous_max.append(str(currentMax))
                current_max.append(str(currentMax))
                previous_std.append(str(currentStd))
                current_std.append(str(currentStd))
            else:
                if str(currentMean) == 'nan':
                    current_mean.append(str(previous_mean[nfeature]))
                else:
                    current_mean.append(str(currentMean))
                if str(currentMin) == 'nan':
                    current_min.append(str(previous_min[nfeature]))
                else:
                    current_min.append(str(currentMin))
                if str(currentMax) == 'nan':
                    current_max.append(str(previous_max[nfeature]))
                else:
                    current_max.append(str(currentMax))
                if str(currentStd) == 'nan':
                    current_std.append(str(previous_std[nfeature]))
```

```
                else:
                    current_std.append(str(currentStd))
                currentLine = currentLine + str(current_mean[nfeature]) + ","
                currentLine = currentLine + str(current_min[nfeature]) + ","
                currentLine = currentLine + str(current_max[nfeature]) + ","
                currentLine = currentLine + str(current_std[nfeature]) + ","
                nfeature += 1
        if df_current.shape[0] > 0:
            # select 'activityrecognition#0' and 'activityrecognition#1' from  df_current
            df_current_google = df_current[['activityrecognition#0', 'activityrecognition
#1']]
            df_current_google = df_current_google[df_current_google['activityrecognition#
1'] >= 0]
            current_values = []
            if df_current_google.shape[0] == 0:
                current_values.append(previous_activityRec)
                current_values.append(previous_activityRecProba)
            else:
                if df_current_google.shape[0] == 1:
                    df_row = df_current_google
                    current_values.append(df_row['activityrecognition#0'].item())
                    current_values.append(df_row['activityrecognition#1'].item())
                    previous_activityRec = ""
                    previous_activityRecProba = ""
                else:
                    # pick prediction with max probability to be correct
                    activity0 = df_current_google.loc[df_current_google['activityrecognit
ion#1'].idxmax()][
                        'activityrecognition#0']
                    activity1 = df_current_google.loc[df_current_google['activityrecognit
ion#1'].idxmax()][
                        'activityrecognition#1']
                    current_values.append(activity0)
                    current_values.append(activity1)
                    previous_activityRec = activity0
                    previous_activityRecProba = activity1

        previous_mean = list(current_mean)
        previous_min = list(current_min)
        previous_max = list(current_max)
        previous_std = list(current_std)
        [...]
```

The computation of time window and feature is done into a *clean* dataset. With *clean*, we mean a set of operation that we do before generate our balanced dataset.

One of the most important operation is the the trasformation of the data come from sensors. Each sensor returns an array of values, with different lengths and content. Before use them is important to consider if they are influenced by the orientation of the device during experiments.

Some sensors, like ambiental (sound, light and pressure) and proximity, returns a single data value as the result of sense, this can be directly used in

dataset. Instead, all the other return more than one values that are related to the coordinate system used, so their values are strongly related to **orientation**.

For almost all we can use an orientation-independent metric, **magnitude**. Magnitude is an appropriate transformation for all the remaining sensor except for rotation vector and game rotation vector, that capture the rotation of the device, rotation can be described by an angle θ.

```
def transform_raw_data(self):
      [...]
      for file in filenames:
        if file.endswith(".csv"):
            with open(os.path.join(dir_src, file)) as current_file:
                with open(os.path.join(dir_dst, file), "w") as file_result:
                    for line in current_file:
                        line_data = line.split(",")
                        endLine = ",".join(line_data[2:])
                        current_time = line_data[0]
                        sensor = line_data[1]
                        user = "," + line_data[(len(line_data) - 2)] if self.sintetic els
e ""
                        target = "," + line_data[(len(line_data) - 1)] if self.sintetic e
lse ""
                        target = target.replace("\n","")
                        # check sensors
                        if line_data[1] not in const.SENSORS_TO_EXCLUDE_FROM_DATASET:  #
the sensor is not to exlude
                            if line_data[1] not in const.SENSOR_TO_TRANSFORM_MAGNITUDE:
# not to transofrom
                                if line_data[1] not in const.SENSOR_TO_TRANSFROM_4ROTATIO
N:  # not to trasform (4 rotation)
                                    if line_data[1] not in const.SENSOR_TO_TAKE_FIRST:  #
not to take only first data
                                        # report the line as it is
                                        current_sensor = line_data[1]
                                        line_result = current_time + "," + current_sensor
+ "," + endLine
                                    else:
                                        current_sensor = line_data[1]
                                        vector_data = line_data[2:] if not self.sintetic
else line_data[2:(len(line_data) - 2)]
                                        vector_data = [float(i) for i in vector_data]
                                        line_result = current_time + "," + current_sensor
+ "," + str(vector_data[0]) + user + target + "\n"
                                else:  # the sensor is to transform 4 rotation
                                    current_sensor = line_data[1]
                                    vector_data = line_data[2:] if not self.sintetic else
line_data[2:(len(line_data) - 2)]
                                    vector_data = [float(i) for i in vector_data]
                                    magnitude = math.sin(math.acos(vector_data[3]))
                                    line_result = current_time + "," + current_sensor + "
," + str(magnitude) + user + target + "\n"
                            else:  # the sensor is to transform
                                current_sensor = line_data[1]
                                vector_data = line_data[2:] if not self.sintetic else lin
e_data[2:(len(line_data)-2)]
                                vector_data = [float(i) for i in vector_data]
                                magnitude = math.sqrt(sum(((math.pow(vector_data[0], 2)),
                                                           (math.pow(vector_data[1], 2)),
                                                           (math.pow(vector_data[2], 2)))
))
                                line_result = current_time + "," + current_sensor + "," +
str(magnitude) + user + target + "\n"
                                file_result.write(line_result)
        elif file.endswith(".json"):
            shutil.copyfile(os.path.join(dir_src,file),os.path.join(dir_dst,file))
      [...]
```

Other operations to clean up the dataset can be summarized as follows and managed within the **clean_files** function:

- we delete measure from the sensors to exclude from our computation;
- we make the values of the sound and speed sensors positive
- if time windows have incorrect values ("/", ">", "<", "-", "_"...), we delete file
- if a file is empty, we delete it

```
def clean_files(self):
       [...]
        filenames = listdir(const.DIR_RAW_DATA_ORIGINAL)
        # iterate on files in raw data directory - delete files with incorrect rows
        nFiles = 0
        deletedFiles = 0
        for file in filenames:

            if file.endswith(".csv"):
                nFiles += 1
                # to_delete be 1 if the file have to be excluded from the dataset
                to_delete = 0
                with open(os.path.join(const.DIR_RAW_DATA_ORIGINAL, file)) as current_fil
e:
                    res_file_path = os.path.join(const.DIR_RAW_DATA_CORRECT, file)
                    with open(res_file_path, "w") as file_result:
                        for line in current_file:
                            line_data = line.split(",")

                            first_line = True
                            if first_line:
                                first_line = False
                                if line_data[1] == "activityrecognition":
                                    line_data[0] = "0"

                            endLine = ",".join(line_data[2:])
                            # check if time data is correct, if is negative, make modulo
                            if re.match(patternNegative, line_data[0]):
                                current_time = line_data[0][1:]
                            else:
                                # if is not a number the file must be deleted
                                if re.match(patternNumber, line_data[0]) is None:
                                    to_delete = 1
                                current_time = line_data[0]
                            # check sensor, if is in sensors_to_exclude don't consider
                            if line_data[1] not in const.SENSORS_TO_EXCLUDE_FROM_FILES:
                                current_sensor = line_data[1]
                                line_result = current_time + "," + current_sensor + "," +
endLine
                                file_result.write(line_result)

                # remove files with incorrect values for time
                if to_delete == 1:
                    logging.info("  Delete: " + file + " --- Time with incorrect values")
                    deletedFiles += 1
                    os.remove(res_file_path)

        # delete empty files
        file_empty = []
        filenames = listdir(const.DIR_RAW_DATA_CORRECT)
        for file in filenames:
            full_path = os.path.join(const.DIR_RAW_DATA_CORRECT, file)
            # check if file is empty
            if (os.path.getsize(full_path)) == 0:
                deletedFiles += 1
                file_empty.append(file)
                logging.info("  Delete: " + file + " --- is Empty")
                os.remove(full_path)
```

```
        pattern = re.compile("^[0-9]+,[a-z,A-Z._]+,[-,0-9a-zA-Z.]+$", re.VERBOSE)
        # pattern = re.compile("^[0-9]+,[a-z,A-Z,\.,_]+,[-,0-9,a-z,A-Z,\.]+$", re.VERBOSE
)
        filenames = listdir(const.DIR_RAW_DATA_CORRECT)
        for file in filenames:
            n_error = 0
            full_path = os.path.join(const.DIR_RAW_DATA_CORRECT, file)
            # check if all row respect regular expression
            with open(full_path) as f:
                for line in f:
                    match = re.match(pattern, line)
                    if match is None:
                        n_error += 1
            if n_error > 0:
                deletedFiles += 1
                os.remove(full_path)
        [...]
```

Start with balanced dataset

Before you start reading this section, you must have the balanced dataset we discussed above. The balanced dataset can be downloaded in our download section (download.html) or built through the steps described above.
Before applying our machine learning techniques, you must split our balanced dataset into a training set and test set. This is automatically generated through the **split_dataset** function. This feature is applied after building the balanced dataset. If you already have the dataset, it is only partitioned.

```
def __split_dataset(self, df):
    [...]
    training, cv, test = util.split_data(df, train_perc=const.TRAINING_PERC, cv_perc=cons
t.CV_PERC,
                                         test_perc=const.TEST_PERC)
    training.to_csv(dir_src + '/' + file_training_dst, index=False)
    test.to_csv(dir_src + '/' + file_test_dst, index=False)
    cv.to_csv(dir_src + '/' + file_cv_dst, index=False)
```

Now, we are ready to apply machine learning techniques to build a **model**.

```
if __name__ == "__main__":
        detection = TMDetection()
```

However, current methods have two key limitations, first on the dataset, on the way it is collected as on the dimension in user's, and second on sensors choice.
We try to overcome the first limitation collecting data from thirteen users without any restriction on the device's position and on the sensor device characteristics. Now we investigate how transportation mode recognition can benefit from the use of sensor data from sensors already available on mobile phones.
In literature, the use of GSM, GPS, accelerometer and gyroscope data has already been explored reaching good accuracy results. Unfortunately, due to the lack of recognized dataset by the scientific community, compare these results in not possible. To the best of our knowledge, no study has been conducted on the possibility to take advantage from other device's sensors.
Before expanding the base sensors on which the classification model is built, is important to consider the current mobile context. Mobile and handheld devices are generally constrained due to resource limitations primarily caused by limited battery life, limited size of memory or limited power of the processor.
Limitations on resource motivated us to carefully consider if, how, and when to use sensors data. We also have to consider that many sensors are always active and already providing services to the system, then collection and use will not result in a greater battery consumption. However, we still have to consider memory and CPU consumption needed to process the data because that can impact other applications and the overall user experience.
Appears clear the need of a trade-off between the consumption of resources and the accuracy of the model. Obviously, this equilibrium point changes under different scenarios. Applications have different accuracy requirements or different use cases or even different devices on which they are used. All of these variations can make more or less relevant resource consumption.
For these reasons it seems hard to find one model right for all situations but it

is clear the importance to understand which and how much mobile phone sensors data can help in distinguishing between different user activity.
We have selected **fifteen sensors** among those that have been collected by our volunteers, based on activity and user support. Among these fifteen sensors, there are some whose data can give their contribution to the transportation mode recognition, others which introduce only noise.
Based on meaning reasons, we exclude from data usedcfor training model the following sensors:

- light
- pression
- magnetic field
- magnetic field uncalibrated
- gravity
- proximity

From remaining nine possibly relevant sensors we have created **three different sensors set**.
First set contain only three sensors whose are base sensors and have almost complete or complete users support (accelerometer, gyroscope, and sound). The **second** set contains sensors from first set plus all the other sensor excluded speed, that is a GPS-based sensor, for its high consumption of the battery. Lastly, the **third** set contains all sensor that we define as relevant for the task.

Sensors of first class of classification	Sensors of second class of classification	Sensors of third class of classification
Accelerometer	Accelerometer	Accelerometer
Sound	Sound	Sound
	Orientation	Orientation
	Linear acceleration	Linear acceleration
		Speed

Sensors of first class of classification	Sensors of second class of classification	Sensors of third class of classification
Gyroscope	Gyroscope	Gyroscope
	Rotation vector	Rotation vector
	Game rotation vector	Game rotation vector
	Gyroscope uncalibrated	Gyroscope uncalibrated

Our main purpose is to create three different models, based on these three different set and see which of them perform better on the test set and try to understand why.
For each set, we build four model with four different **classification algorithms**:

- **Decision Trees** (DT)
- **Random Forest** (RF)
- **Support Vector Machines**(SVM)
- **Neural Network** (NN)

Choosing every time the model that is the most accurate for transportation mode inference on the test set. NN and SVM, require accurate selection of thresholds and parameters.

Single sensors dataset

Random forest is widely used and seems to perform pretty well on different datasets. Therefore, for a first investigation of how the sensors can be discriminating with the defined classes, we choose random forest algorithm.
We first restricted the dataset to the features related to a single sensor, trained the model on this new dataset, and then tested on the test set.

```
def single_sensor_accuracy(self):
        sensor = []
        accuracy = []
        std = []
        for s in self.dataset.get_sensors:
            if s != "activityrecognition":
                print(s)
                features = self.dataset.get_sensor_features(s)
                train = self.dataset.get_train.copy()
                test = self.dataset.get_test.copy()

                train_features, train_classes, test_features, test_classes = self.__get_s
ets_for_classification(train,

test,

features)
                singleAcc = []
                for i in range(const.REPEAT):
                    # build classifier
                    classifier_forest = RandomForestClassifier(n_estimators=const.PAR_RF_
ESTIMATOR)
                    classifier_forest.fit(train_features, train_classes)
                    test_prediction_forest = classifier_forest.predict(test_features)
                    acc_forest = accuracy_score(test_classes, test_prediction_forest)
                    singleAcc.append(acc_forest)
                accM = util.average(singleAcc)
                variance = list(map(lambda x: (x - accM) ** 2, singleAcc))
                standard_deviation = math.sqrt(util.average(variance))
                print(s, accM, standard_deviation)

                accuracy.append(accM)
                std.append(standard_deviation)
                sensor.append(s)
        df_single_sensor_acc = pd.DataFrame({'sensor': sensor, 'accuracy': accuracy, 'dev
_standard': std})
        df_single_sensor_acc = df_single_sensor_acc.sort_values(by='accuracy', ascending=
False)
        [...]
```

This preliminary result shows in Figure 2 gives us the idea of how great can be the impact of different sensors.

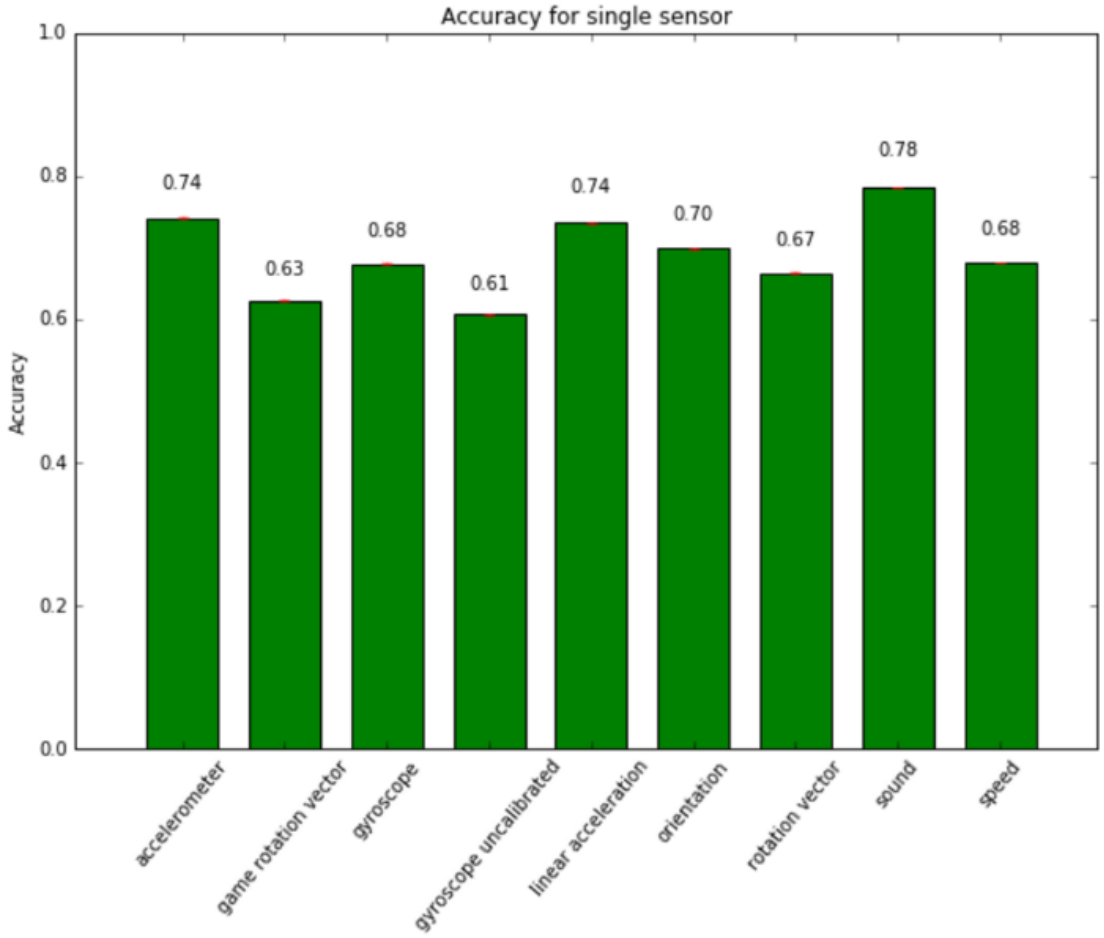

Fig 2: The accuracy of the model trained on a dataset composed of features of a single sensor with random forest algorithm

Accuracy's values with only one sensor are between 0.57 and 0.75, so all considered sensors have the capability to capture some pattern related to the activity the user is performing during the measurements.
According to what we could expect, **accelerometer** and **linear acceleration** data are two of most discriminating of the considered sensors. Even if no studies have investigated the use of the linear acceleration data, its relevance is the direct consequence of accelerometer, because its data are based only on accelerometer sensors. **Gyroscope** reaches good accuracy, but this result was to be expected from previous studies. The most interesting result is the high relevance of **sound's data**. At the best of our knowledge no one has investigated the use of microphone data, although it is an essential sensor for mobile phone use, so is present in all devices.

Three sensors set dataset

For reasons that we already explained previously, we start from dataset formed by a smaller set of sensors. The 1, 2, 3 parameters assigned to the functions refer to the three different set of sensors shown in the previous table.

```
if __name__ == "__main__":
        detection = TMDetection()

#       detection.decision_tree(1)
#       detection.decision_tree(2)
#       detection.decision_tree(3)

#       detection.random_forest(1)
#       detection.random_forest(2)
#       detection.random_forest(3)

#       detection.neural_network(1)
#       detection.neural_network(2)
#       detection.neural_network(3)

#       detection.support_vector_machine(1)
#       detection.support_vector_machine(2)
#       detection.support_vector_machine(3)

[...]
```

Sensors included in the first set (parameter 1) are accelerometer, sound, and gyroscope. These three sensors have the highest values of accuracy taken individually.

First dataset D_{first} is formed by twelve features, four for each sensor. We perform classification with the four classification algorithms mentioned before. The overall accuracy for algorithms is between 82% and 88%. Even if random forest produce the highest accuracy values (88%), all algorithms perform substantially well.

By expanding the dataset adding all other relevant sensors except speed, for battery saving purposes, we reach better results in term of accuracy. With **second set dataset D_{second}** , formed by eight sensor and thirty-two features, accuracy increases up to values between 86% and 93%.

Lastly we train a model on the **third dataset D_{third}** formed by all nine relevant sensors and thirty-six features, differ from previous D_{second} only for speed

derived features. Result show how considering speed, further increasing the ability of the model to infer which transportation mode the user is currently using. In this last case, the accuracy reached a range level between 91% and 96%

Algorithm	Accuracy of first dataset	Accuracy of second dataset	Accuracy of third dataset
Decision Tree (DT)	82%	86%	91%
Random Forest (RF)	88%	93%	96%
Support Vector Machine (SVM)	85%	93%	95%
Neural Network (NN)	85%	92%	95%

However, typically context-aware applications do not need to recognize all the classes as we did, since they generally need only a subset of them. Therefore, the next analysis we perform is devoted to the **class-to-class classification**, in which we perform a similar analysis but considering only two classes to discriminate between.

Two classes classification

Obviously, reducing the amount of classes to be classified raises without any exception the accuracy of any classification algorithm. However, we report only the results from the RF, since it has shown the best results also for this analysis.

```
def classes_combination(self, sensors_set):
        features = list(self.dataset.get_sensors_set_features(sensors_set))
        class_combination = list(itertools.combinations(self.classes, 2))
        train = self.dataset.get_train.copy()
        test = self.dataset.get_test.copy()
        if not os.path.exists(const.DIR_RESULTS):
            os.makedirs(const.DIR_RESULTS)
        with open(const.DIR_RESULTS + "/" + str(sensors_set) + const.FILE_TWO_CLASSES_COM
BINATION, 'w') as f:
            f.write("combination, algorithm, accuracy")
            for combination in class_combination:
                cc_train = train.loc[(train['target'] == combination[0]) | (train['target
'] == combination[1])]
                cc_test = test.loc[(test['target'] == combination[0]) | (test['target'] =
= combination[1])]
                train_features, train_classes, test_features, test_classes = self.__get_s
ets_for_classification(
                    cc_train, cc_test, features)

                # buil all classifier
                classifier_tree = tree.DecisionTreeClassifier()
                classifier_forest = RandomForestClassifier(n_estimators=const.PAR_RF_ESTI
MATOR)
                classifier_nn = MLPClassifier(hidden_layer_sizes=(const.PAR_NN_NEURONS[se
nsors_set],),
                                              alpha=const.PAR_NN_ALPHA[sensors_set], max_
iter=const.PAR_NN_MAX_ITER,
                                              tol=const.PAR_NN_TOL)
                classifier_svm = SVC(C=const.PAR_SVM_C[sensors_set], gamma=const.PAR_SVM_
GAMMA[sensors_set],
                                     verbose=False)

                # train all classifier
                classifier_tree.fit(train_features, train_classes)
                classifier_forest.fit(train_features, train_classes)
                classifier_nn.fit(train_features, train_classes)
                classifier_svm.fit(train_features, train_classes)

                # use classifier on test set
                test_prediction_tree = classifier_tree.predict(test_features)
                test_prediction_forest = classifier_forest.predict(test_features)
                test_prediction_nn = classifier_nn.predict(test_features)
                test_prediction_svm = classifier_svm.predict(test_features)

                # evaluate classifier
                acc_tree = accuracy_score(test_classes, test_prediction_tree)
                acc_forest = accuracy_score(test_classes, test_prediction_forest)
                acc_nn = accuracy_score(test_classes, test_prediction_nn)
                acc_svm = accuracy_score(test_classes, test_prediction_svm)
    [...]
```

Figure 3 reports the accuracy when classifying all the possible couples of classes. At first, it is straightforward to note how some couples are more challenging than others, especially for D_{first} and D_{second}. For instance, **{Bus,Car}**, **{Bus,Train}** and **{Car,Train}** highlight a considerable increase in

accuracy when switching from D_{first} to D_{second} and eventually to D_{third}, hence confirming the importance of the speed feature for motorized classes. On the other hand, recognizing couples of activities in which one is **{Walking}** is easier even for D_{first}. Hence, for those tasks the importance of having the features obtained from the speed is lower.

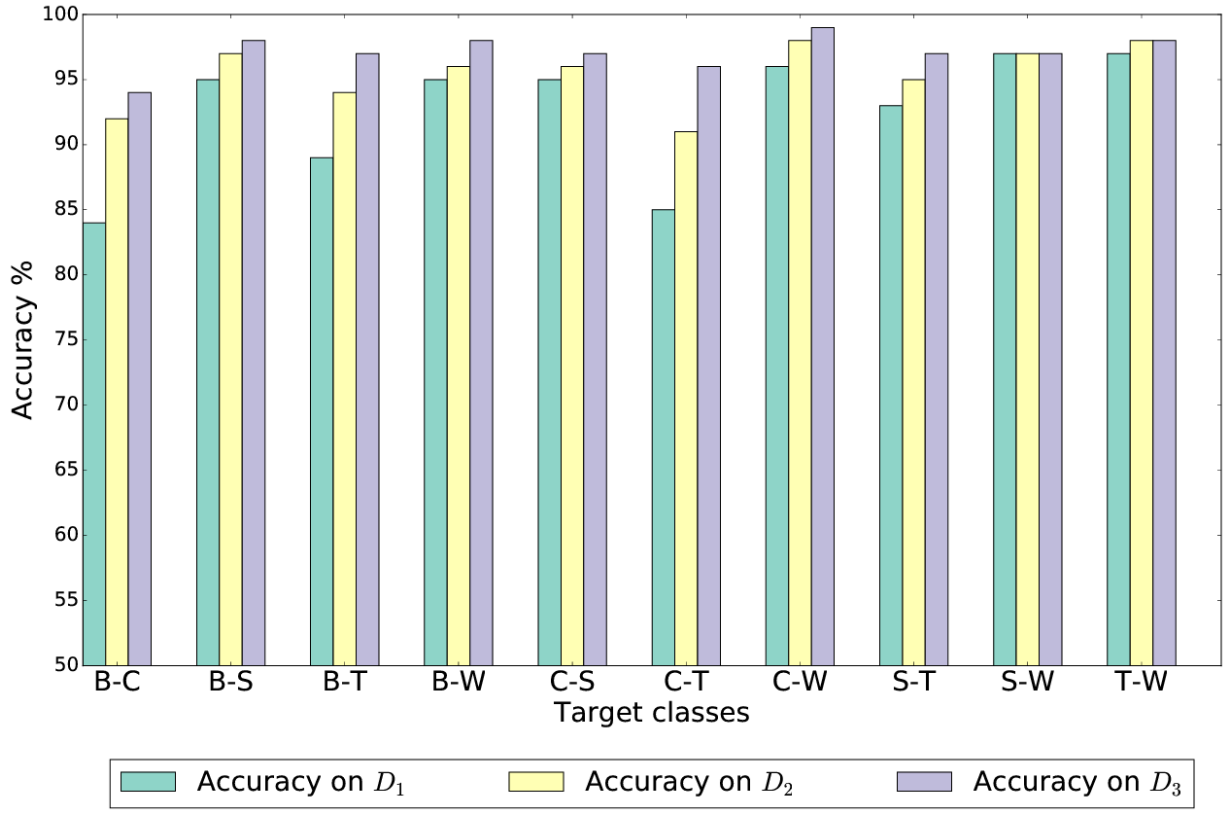

Fig 3: Class-vs-class accuracy on the test set with respect to the three different sensors sets D_{first}, D_{second} and D_{third}. Letters B, C, S, W, T stand for **B**us, **C**ar, **S**till, **W**alking and **T**rain respectively.