Multiclass classification problem for Human Activity Recognition

Technical report of the obtained results via scikit-learn (Python)

Statistical Learning and Data Mining course

Automation Engineering
University of Naples Federico II

April 2021

Student:
Alessandro Melone
(matr. M58000263)

Supervisor:
Prof. Roberta Siciliano

Abstract

In this work are shown the results of the application of various learning techniques to a multi-class prediction problem. The considered classification problem belongs to the Human Activity Recognition (HAR) field, in particular the dataset is focused on the classification of Activities of Daily Living (ADL).

The dataset is fitted with different estimators in order to find the estimator that has the highest accuracy score.

The used learning methods are implemented in the `scikit-learn` python package [3].

Contents

List of Figures

Acronyms

ADL Activities of Daily Living.

HAR Human Activity Recognition.

KNN K-Nearest Neighbor Classifier.

PCA Principal component analysis.

SVC Support Vector Classifier.

Chapter 1

Dataset and problem specification

The dataset is obtained from the measurements of inertial sensors embedded in a mass-marketed smartphone[1], the dataset is also freely downloadable [2].

For the experiment is considered a group of 30 volunteers with ages ranging from 19 to 48 years, each volunteer was instructed to follow a protocol of activities while wearing a waist-mounted Samsung Galaxy S II.

Six ADL are considered, i.e. standing, sitting, laying down, walking, walking downstairs and upstairs.

In Fig.1.1 is shown a frame of the video [5] that illustrates the setup of the experiment and the experiment protocol of activities.

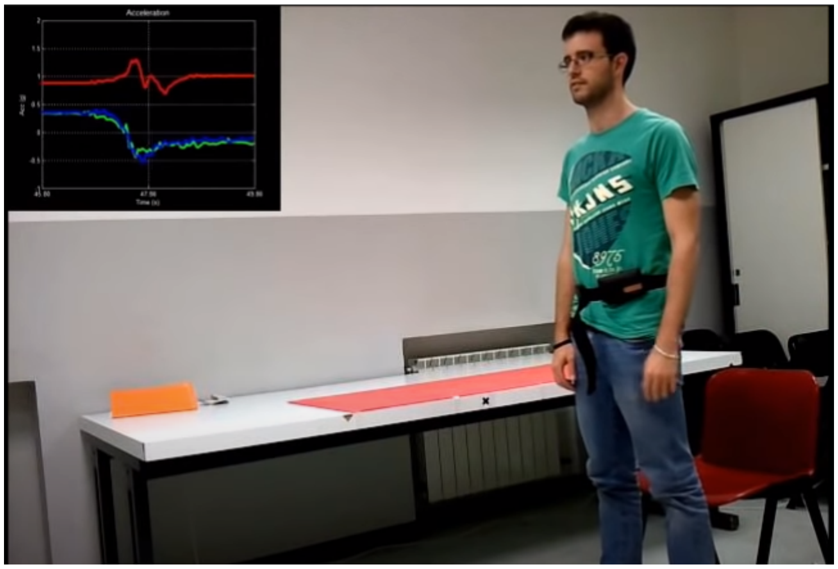


Figure 1.1: Setup of the experiment in [1]

1.1 Signal acquisition

To extract the features various steps have been done, firstly the triaxial linear acceleration and angular velocity signals (obtained respectively using the phone accelerometer and gyroscope) are sampled with rate of 50Hz.

The noise reduction is performed with a median filter and a 3rd order low-pass Butterworth filter with a 20 Hz cutoff frequency. The choice of the cutoff frequency is taken considering that 99 of the human body motion energy is contained below 15Hz.

In the acceleration signal the gravitational force need to be separated from the body motion components. Making the assumption that gravitational force have only low frequency components a Butterworth low-pass filter with cutoff frequency of 0.3 Hz is used.

Are also calculated other time signals: euclidean magnitude and time derivatives (jerk da/dt and angular acceleration dw/dt).

These time variant signals are sampled in fixed-width sliding windows of 2.56 sec and 50% overlap between them.

Finally, the signals are also mapped in the frequency domain through a Fast Fourier Transform (FFT), except for the gravity related signals (are approximately constants). Thus the number of the all signals (i.e. both time and frequency domain) is 17.

1.1.1 Feature Mapping

For each sampled window is extracted a vector of 561 features. Thus 561 is the number of all the measurements done considering all the 17 signal.

The standard measures previously used in HAR literature are: mean, correlation, signal magnitude area (SMA) and autoregression coefficients. The dataset uses the above measures as well as a new set of features in order to improve the learning performance: including energy of different frequency bands, frequency skewness, and angle between vectors.

The names of all the features are present in the `dataset/dataset_documentation/features.txt`.

1.2 Exploratory Data Analysis

The results reported in this paragraph are obtained from `dataset_analysis.py`.

In Fig.1.2 is possible to see that the dataset is not unbalanced. A classification dataset is unbalanced if the classes are not represented equally, i.e. the target variable has more observations in one specific class than the others.

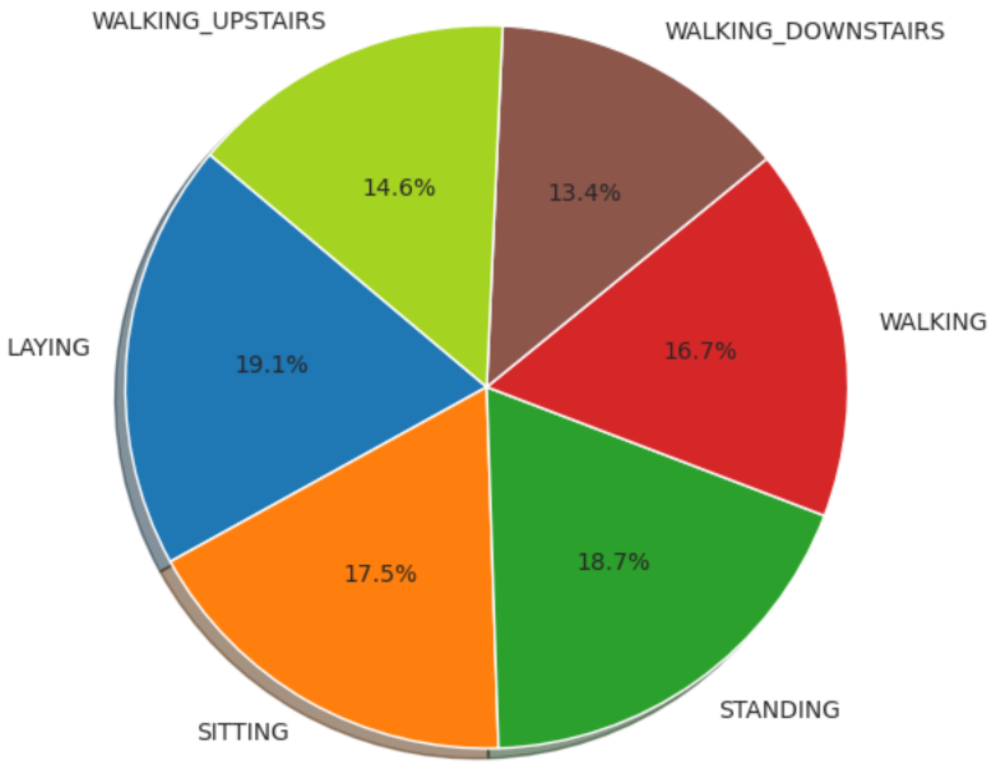


Figure 1.2: Dataset pie label occurrence

Also the amount of collected data per user is not unbalanced, thus the estimator trained on the dataset will not be affected by the particular kind of walking gait of the most represented user.

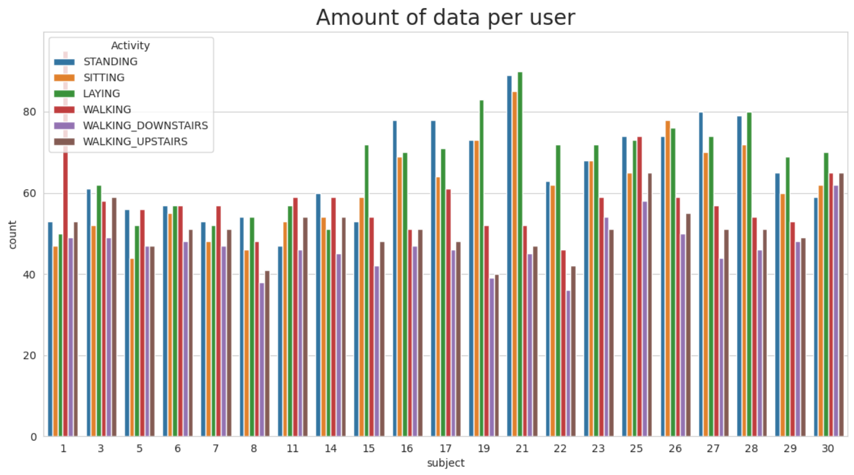


Figure 1.3: Amount of data collected for each of the 30 volunteers

Chapter 2

Hardware and Software Platforms

In this chapter are described the hardware and the software environment used to perform the fitting of the dataset with the considered methods.

2.1 Hardware

The computer hardware specification are:

- notebook: LENOVO B50-80
- CPU: Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz
- RAM: dual channel, each slot has Kingston SODIMM DDR3 Synchronous 1600 MHz, 4GB
- HD: Samsung SSD 860, 500GB

To speed up the searching of some computational demanding hyper-parameter tuning has been used the `aws ec2 t2.xlarge`, that offers 4 virtual CPUs.

2.2 Software environment

For what concern the software specification:

- OS: Ubuntu 18.04
- programming language: Python 3.8.5
- IDE: Spyder
- Environment manager: Anaconda

Chapter 3

Chosen methods and results

In this chapter are discussed the estimators considered to achieve the best prediction accuracy, in particular are used estimators implemented in the `scikit-learn` python library [3], the used functions are well documented in the online user guide[4].

For each of the proposed estimators a tuning of the hyper-parameters is performed, so in this chapter is introduced first the way this tuning is performed and then the estimators are discussed.

3.1 Standardization of the dataset

For what concern the preprocessing the range of the features is already contained between $[-1; 1]$, so the only preprocessing operation is done using the `StandardScaler`.

This function using the method `fit` computes and stores for each feature the mean and the variance of that feature among the training set. In formulas, considering the $j-th$ feature, with $j \in \{1, ..., p\}$ ($p = 561$ in our case of study) and indicating the i-th observation in the training set with $\mathbf{x}_i = [x_{i1}, ...x_{ip}] \in X$, with X the training set and $i \in \{1, ..., n\}$ ($n = 7352$ in our case of study) then the algorithm computes:

$$\mu_j = \frac{1}{n}\sum_{i=1}^{n} x_{ij} \qquad \sigma_j^2 = \frac{1}{n}\sum_{i=1}^{n}(x_{ij} - \mu_j)^2$$

After the model is fitted is possible to transform a generic observation using the function `tansform`, i.e. given the observation $\mathbf{x} = [x_1, ..., x_p]$ the transformed observation will be $\bar{\mathbf{x}}$ with:

$$\bar{x}_j = \frac{x_j - \mu_j}{\sigma_j} \quad \text{with} \quad j \in \{1, ..., p\}.$$

The standardization is a common requirement for many estimator, like the Support Vector Machines and the l1 or l2 regularizers of linear models, these estimators assumes that all features are centered around 0 and have variance in the same order. If a feature have variance of order of magnitude bigger, it might dominate the objective function.

This scaling is performed for all the considered methods (even if for the k-nearest neighbour is not necessary) using the class `Pipeline` that is useful to concatenate a fixed sequence of steps, in which a ”step” is a transformation and/or an estimator.

3.2 Hyper-parameters tuning methods

The hyper-parameters tuning has the goal to choose the best set of parameters that are not automatically learnt by the estimator through the fitting (i.e. the hyper-parameters).

To assess the goodness of a given set of hyper-parameters is used the *stratified 5-fold* cross validation on the training set, the used objective metric is the `scikit-learn` default score function for classification problems. The model with the highest accuracy obtained through cross-validation is then fitted on the whole training set.

3.2.1 Choice of the candidates parameters

The candidates hyper-parameters fed to the cross validation algorithm are chosen using one of the two following approaches.

Exhaustive Grid Search

This approach is implemented through the function `GridSearchCV`, in which the input variable `param_grid` is a python dictionary that contains a list for each of the hyper-parameters. The function will do the cross-validation *for each of the possible* parameters combination, thus the overall number of fitting is equal to the number of all the possible parameters combination multiplied by 5 (since is used a 5-fold cross validation).

Note that the `GridSearchCV` allow an easier representation of the results in a plot, for this reason in the following the cross-validation result will not be shown through a plot if the `RandomizedSearchCV` function is used.

Randomized Parameter Optimization

Since the `GridSearchCV` could led to high number of fitting, an alternative is the function `RandomizedSearchCV` that will choose a limited number of candidate parameters, the number of trials is defined by the input variable `n_iter`.

The candidate parameters are chosen among the set of all the possible parameters combination (specified again by the `param_grid` variable) in a random fashion. In the following is always given as input to the function the `random_state` $= 42$, this means that if the function is called various time the chosen parameters are always the same, this allows to have at the same time the effect of a random choice of the parameters and the reproducibility of the results.

How to read cross-validation results

In the following the cross-validation results will be shown using tables that for each combination of the hyper-parameters show two variables, namely the `mean_test_score` and `std_test_score`. These two variables belong to the dictionary `cv_results_` that is an attribute of the `GridSearch` variable, this variable is accessible after that the fitting is performed.

The `mean_test_score` is the average of the accuracy of the 5-fold, in formulas:

$$\texttt{mean\_test\_score} = \frac{1}{5}\sum_{i=1}^{5} acc_i$$

with acc_i is the accuracy on the $i-th$ fold.

Meanwhile `std_test_score` is the standard deviation computed as:

$$\texttt{std\_test\_score} = \sqrt{\frac{1}{5}\sum_{i=1}^{5}(acc_i - \texttt{mean\_test\_score})^2} \tag{3.1}$$

The parameters corresponding to the higher `mean_test_score` are selected and the estimator is trained on the whole training set, in this way is obtained the final model. Note that the variable `mean_test_score` is an estimate of the testing accuracy meanwhile the `std_test_score` indicates the goodness of this estimate, i.e. an higher value means a less reliable estimation.

3.2.2 Stratified k-fold

The Stratified k-fold is the default method used for classification problems by both `GridSearchCV` and `RandomizedSearchCV`. This method is a variation of the k-fold which gives for each fold an approximately same percentage of samples of each target class as the complete set.

This method avoids that a class is over represented in a given fold, in that case the accuracy gives too much weight to the over represented class. Furthermore if a class is not sufficiently represented in a given fold the estimator cannot be properly fitted.

3.3 Logistic regression

The function `LogisticRegression` implements the logistic regression estimator, using the input variable `multi_class = 'multinomial'` implies that the multinomial loss is minimized, i.e. the softmax function. If is desired to use the logistic function is necessary to set `multi_class = 'ovr'`, i.e. using the one-vs-rest approach.

The regularization method is chosen through the input variable `penalty`, the function support all the three regularization methods: `'l1'`, `'l2'`, `'elasticnet'`. The input variable `C` is a hyper-parameter that sets the regularization strength: if `C` is close to 0 implies a strong regularization, if `C` is close to 1 implies a light regularization.

The `'elasticnet'` penalty is a combination of the `l1` and `l2` penalties, the `l1_ratio` input variable is a real number that belong to $[0; 1]$. If `l1_ratio=0` the penalty is equivalent to the `l2` penalty, instead if `l1_ratio=1` the penalty is equivalent to the `l1` penalty.

All the images and tables shown in this section are obtained by running the `code/plot_log_regr.py` file.

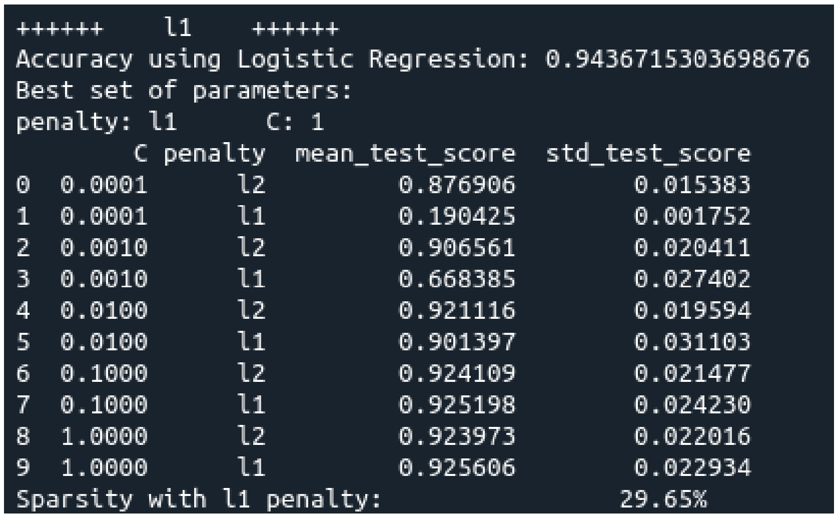
3.3.1 Hyper-parameters tuning

The hyper-parameters tuning is implemented in the file `code/log_regr.py`. Two separate cross-validation have been performed to allow a more clear representation of the results.

l1,l2 cross validation

First are considered the penalty `l1` and `l2`:

- estimator: `LogisticRegression`
- parameter space:
  `C = [0.0001, 0.001, 0.01, 0.1, 1]`
  `penalty = [l1,l2]`
- method to choose the candidates: `GridSearchCV`
- cross-validation scheme: stratified 5-fold
- score function: accuracy



```
++++++    l1    ++++++
Accuracy using Logistic Regression: 0.9436715303698676
Best set of parameters:
penalty: l1      C: 1
        C penalty  mean_test_score  std_test_score
0  0.0001      l2         0.876906        0.015383
1  0.0001      l1         0.190425        0.001752
2  0.0010      l2         0.906561        0.020411
3  0.0010      l1         0.668385        0.027402
4  0.0100      l2         0.921116        0.019594
5  0.0100      l1         0.901397        0.031103
6  0.1000      l2         0.924109        0.021477
7  0.1000      l1         0.925198        0.024230
8  1.0000      l2         0.923973        0.022016
9  1.0000      l1         0.925606        0.022934
Sparsity with l1 penalty:                  29.65%
```

Figure 3.1: l1,l2 cross-validation results table

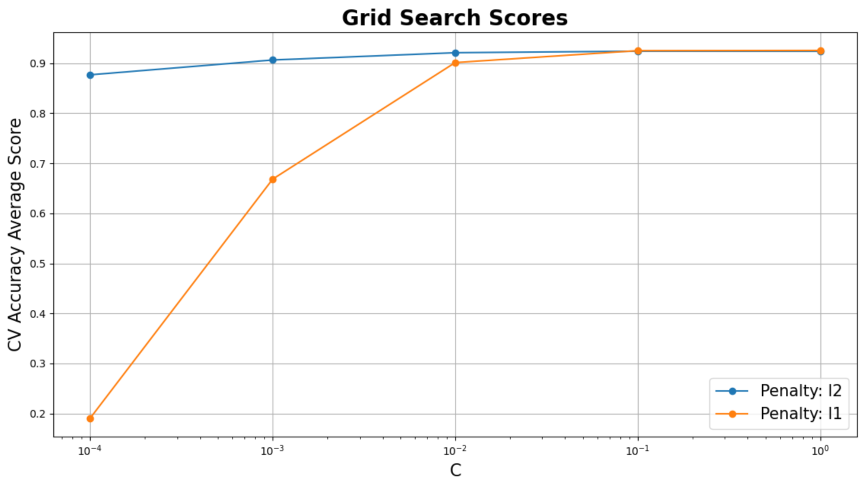


Figure 3.2: l1,l2 cross-validation results, C in log-scale

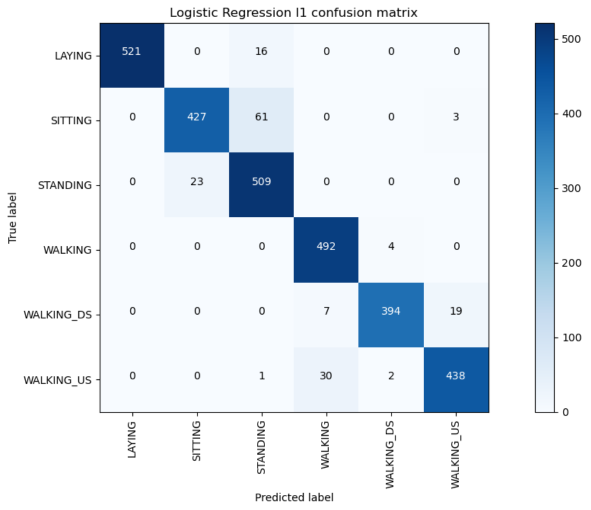


Figure 3.3: Logistic regression best model penalty l1 or l2 confusion matrix

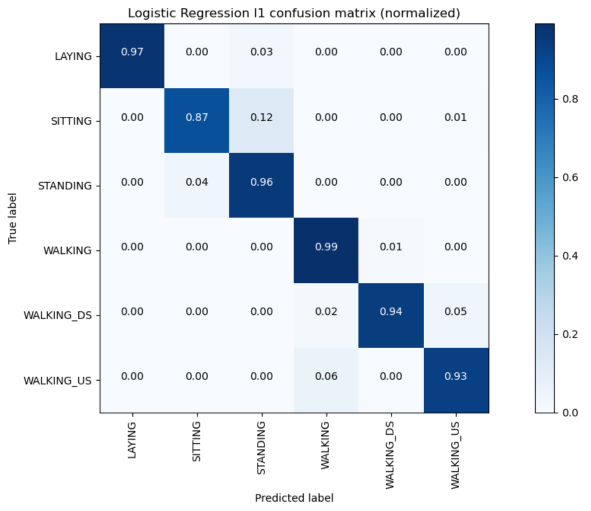


Figure 3.4: Same as Fig.3.3 but normalized

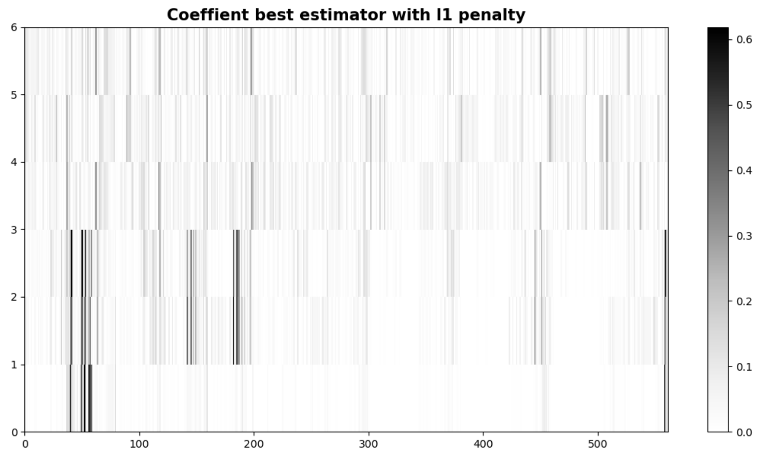
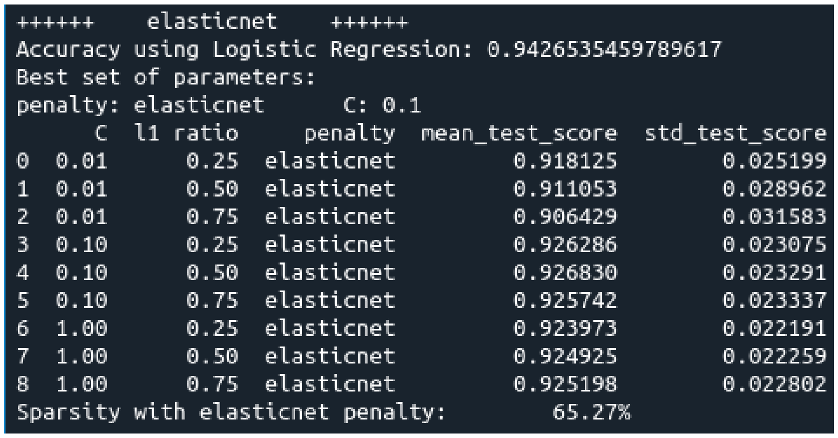


Figure 3.5: Plot of the coefficient of the Logistic Regression estimator with l1 penalty

Elasticnet cross validation

First are considered the penalty `l1` and `l2`:

- estimator: `LogisticRegression`
- parameter space: `C = [0.01, 0.1, 1]` and `l1_ratio = [0.25, 0.5, 0.75]`
- method to choose the candidates: `GridSearchCV`
- cross-validation scheme: stratified 5-fold
- score function: accuracy



```
++++++    elasticnet    ++++++
Accuracy using Logistic Regression: 0.9426535459789617
Best set of parameters:
penalty: elasticnet      C: 0.1
      C  l1 ratio     penalty  mean_test_score  std_test_score
0  0.01      0.25  elasticnet         0.918125        0.025199
1  0.01      0.50  elasticnet         0.911053        0.028962
2  0.01      0.75  elasticnet         0.906429        0.031583
3  0.10      0.25  elasticnet         0.926286        0.023075
4  0.10      0.50  elasticnet         0.926830        0.023291
5  0.10      0.75  elasticnet         0.925742        0.023337
6  1.00      0.25  elasticnet         0.923973        0.022191
7  1.00      0.50  elasticnet         0.924925        0.022259
8  1.00      0.75  elasticnet         0.925198        0.022802
Sparsity with elasticnet penalty:        65.27%
```

Figure 3.6: Elasticnet cross-validation results table

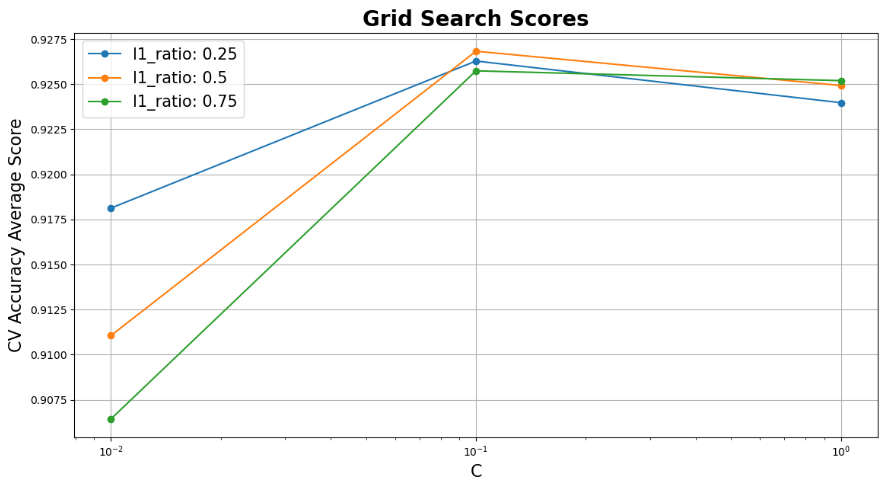


Figure 3.7: Elasticnet cross-validation results, C in log-scale

In Fig.3.6 can be noticed that the elasticnet penalty implies more sparsity respect to the l1 penalty in Fig.3.1, this fact can be explained considering that the parameter C is smaller for the elasticnet case, thus a stronger regularization is done. In fact, if the C parameter is the same the l1 norm gives more sparsity.

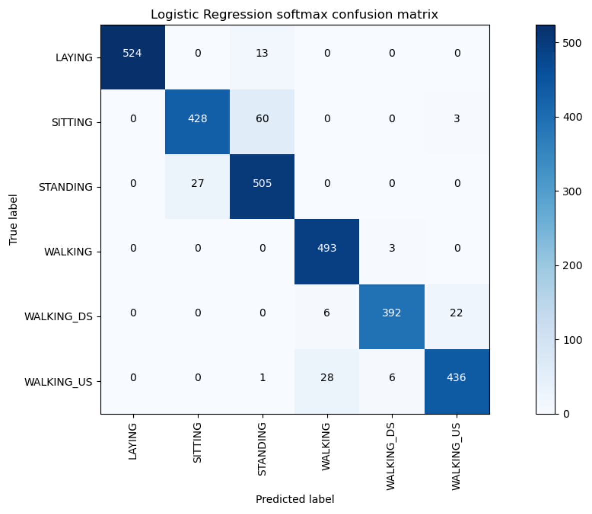


Figure 3.8: Logistic regression best model penalty elasticnet confusion matrix

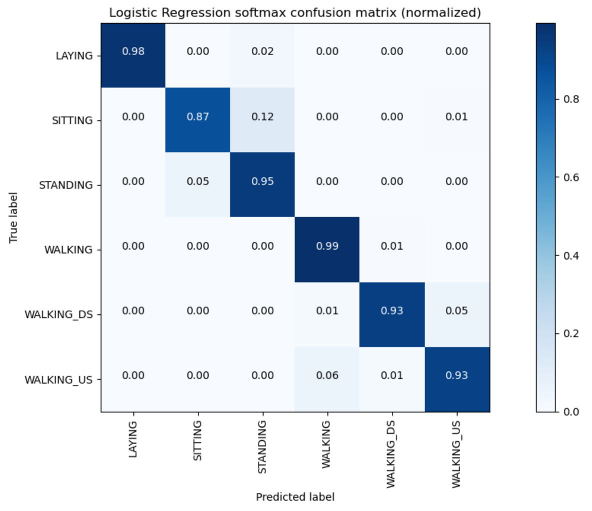


Figure 3.9: Same as Fig.3.8 but normalized

The accuracy of the final model is indicated in the figure with: `Accuracy using Logistic Regression: 0.9436715303698676`. Note that this accuracy is higher than the others in the cross-validation resume because is the accuracy of the model trained on the whole training set.

Same consideration can be done for the Fig.3.6

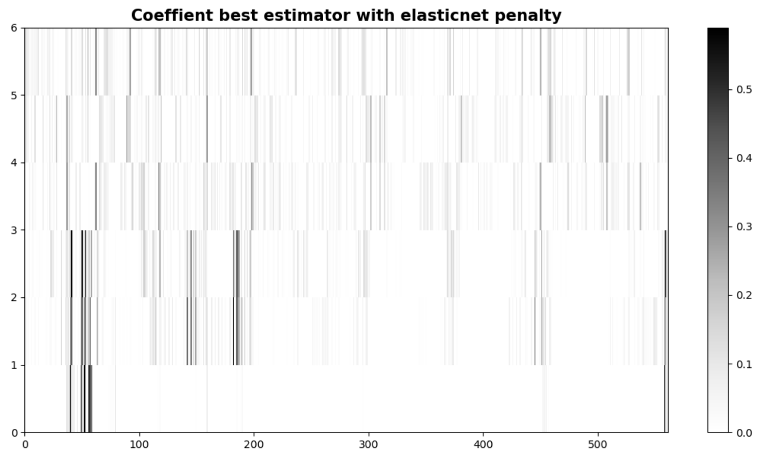


Figure 3.10: Plot of the coefficient of the Logistic Regression estimator with elasticnet penalty

3.4 Nearest Neighbors Classification

The K-Nearest Neighbor Classifier (KNN) is implemented through the function `KNeighborsClassifier`, the input variable `n_neighbors` sets the value of k. The input variable `weights = 'uniform'` means that all the points in each neighborhood are weighted equally despite the distance to the query point.

All the images and tables shown in this section are obtained by running the `code/plot_knn.py` file.

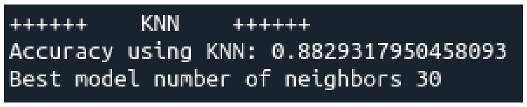
3.4.1 Hyper-parameters tuning

The hyper-parameters tuning is implemented in the file `code/knn.py`. Two separate cross-validation have been performed, one for the KNN alone and the other for the pipeline composed by a dimensionality reduction through a Principal component analysis (PCA) and the KNN.

KNN cross validation

- estimator: `KNeighborsClassifier`
- parameter space: `n_neighbors = np.arange(2,40,1)`
- method to choose the candidates: `GridSearchCV`
- cross-validation scheme: stratified 5-fold
- score function: accuracy

The command `n_neighbors = np.arange(2,40,1)` means that is searched the best `n_neighbors` in the set $\{1, 2, ..., 40\}$



```
++++++    KNN    ++++++
Accuracy using KNN: 0.8829317950458093
Best model number of neighbors 30
```

Figure 3.11: Nearest Neighbors Classifier best model parameters

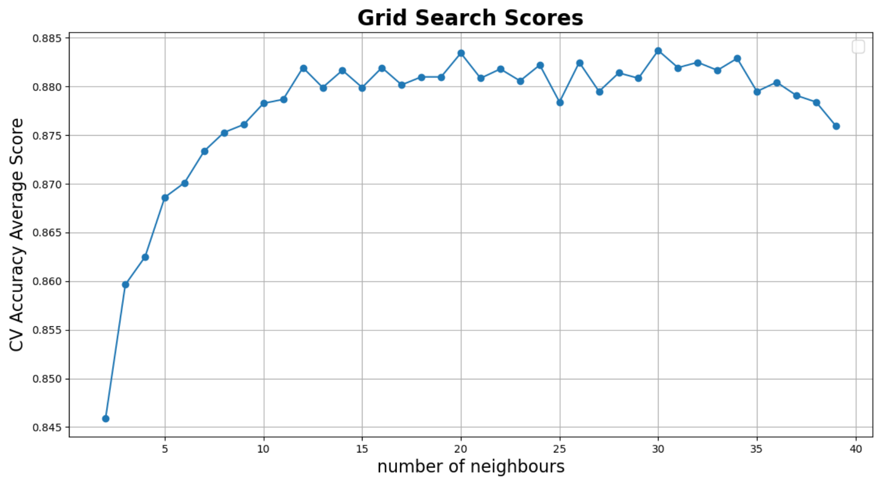


Figure 3.12: Nearest Neighbors Classifier cross-validation plot

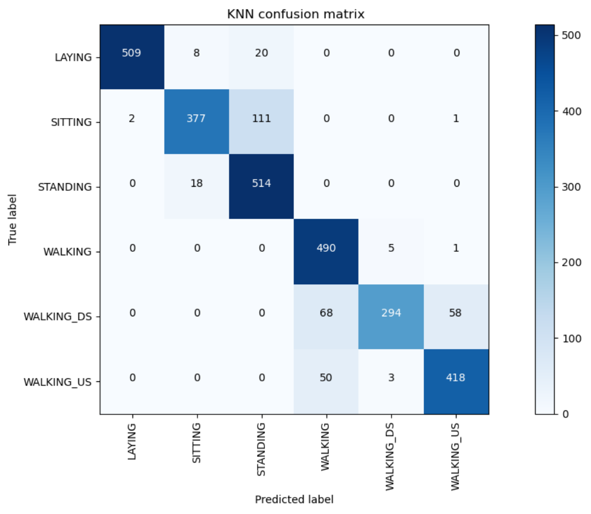


Figure 3.13: Nearest Neighbors Classifier best model confusion matrix

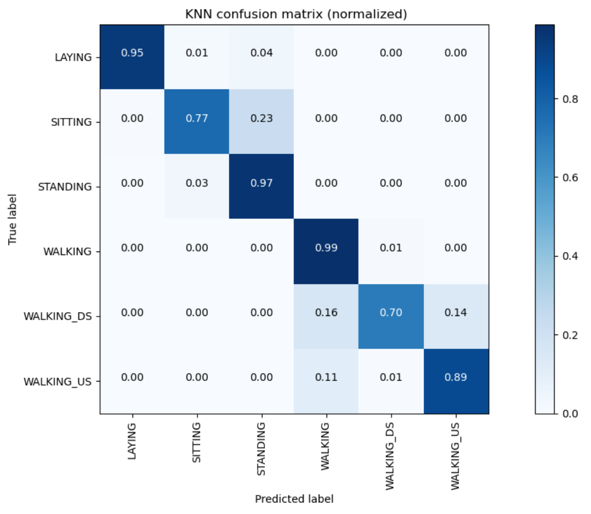


Figure 3.14: Same as Fig.3.13 but normalized

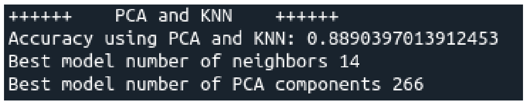
Pipeline PCA and KNN cross validation

- estimator: `KNeighborsClassifier`

- parameter space:
  `n_neighbors = np.arange(2,40,1)`
  `n_components = np.arange(20,400,1)`
- method to choose the candidates: `RandomizedSearchCV` with `n_iter=40`
- cross-validation scheme: stratified 5-fold
- score function: accuracy

The command `n_components = np.arange(20,400,1)` means that is searched the best number of components of the PCA named `n_components` in the set $\{20, 21, ..., 400\}$.

Note that the parameters space is bigger than the previous cases, for this reason is used the function `RandomizedSearchCV`. The parameter `n_iter=40` means that only 40 couple (picked in a random fashion) of `(n_components,n_neighbors)` are considered.



```
++++++    PCA and KNN    ++++++
Accuracy using PCA and KNN: 0.8890397013912453
Best model number of neighbors 14
Best model number of PCA components 266
```

Figure 3.15: PCA + KNN best model parameters

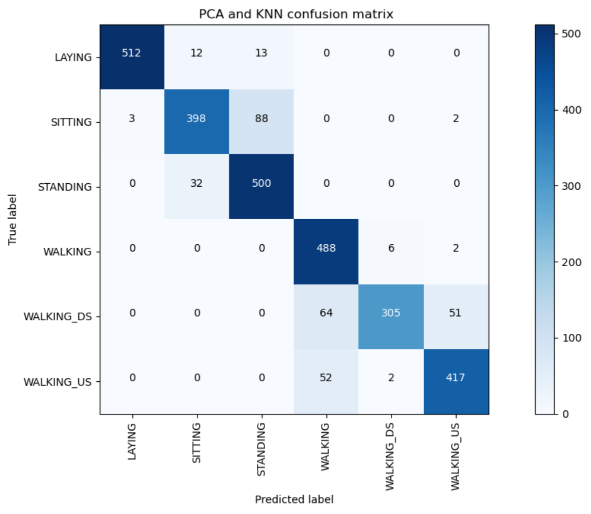


Figure 3.16: Nearest Neighbors Classifier with PCA feature selection, best model confusion matrix

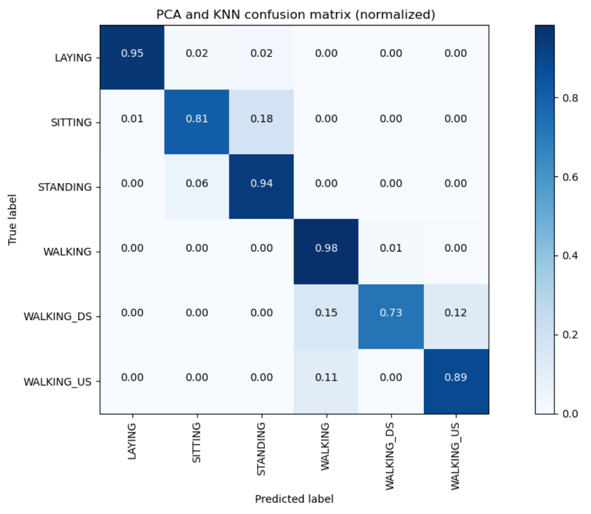


Figure 3.17: Same as Fig.3.16 but normalized

Can be seen that the use of the PCA implies a slightly improvement of the accuracy.

3.5 Linear Support Vector Classification

This estimator is implemented by the function `LinearSVC`, the one-vs-rest method is used to extend the estimator to the multi-class case with the input variable `multi_class = 'ovr'`.

The one-vs-rest method perform the fitting of an estimator for each class, to the considered class is assigned the value 1 while to all the other classes is assigned the value 0. This means that each fitted estimator solve a binary classification problem.

The default input variable `loss = 'squared_hinge'` is chosen to set the squared hinge loss.

The regularization is chosen through the input variable `penalty`. In this paragraph are compared the performance of both the `l1` and `l2` regularization.

Let be $x_i \in \mathbb{R}^p$ with $i = 1, \ldots, n$ the generic observation in the dataset associated with the response $y_i \in \{0, 1\}$. The fitting problem can be formulated as the search of the optimal $w \in \mathbb{R}^p$ and $b \in \mathbb{R}$ that minimize:

$$\min_{w,b} \|w\|_1 + C \sum_{i=1} max(0, y_i(w^T x_i + b))^2 \qquad \text{with l1 penalty}$$

$$\min_{w,b} \frac{1}{2} w^T w + C \sum_{i=1} max(0, y_i(w^T x_i + b))^2 \qquad \text{with l2 penalty}$$

The parameter C is in inverse relationship with the regularization, e.g. decreasing the C value means more regularization. Note that lower C values in general leads to more support vectors, thus an increasing of the prediction time.

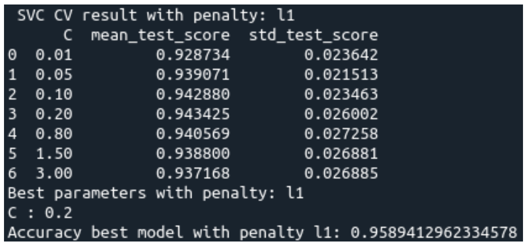
All the images and tables shown in this section are obtained by running the `code/plot_svc.py` file.

3.5.1 Hyper-parameters tuning

The hyper-parameters tuning is implemented in the file `code/svc.py`. Two separate cross-validations have been performed, one using the `l1` regularization and the other using the `l2`.

l1 cross validation

- estimator: `LinearSVC` with `penalty = 'l1'`
- parameter space: `C = [0.01, 0.05, 0.1, 0.2, 0.8, 1.5, 3]`
- method to choose the candidates: `GridSearchCV`
- cross-validation scheme: stratified 5-fold
- score function: accuracy



```
 SVC CV result with penalty: l1
      C  mean_test_score  std_test_score
0  0.01         0.928734        0.023642
1  0.05         0.939071        0.021513
2  0.10         0.942880        0.023463
3  0.20         0.943425        0.026002
4  0.80         0.940569        0.027258
5  1.50         0.938800        0.026881
6  3.00         0.937168        0.026885
Best parameters with penalty: l1
C : 0.2
Accuracy best model with penalty l1: 0.9589412962334578
```

Figure 3.18: Linear SVC with l1 penalty cross-validation results table

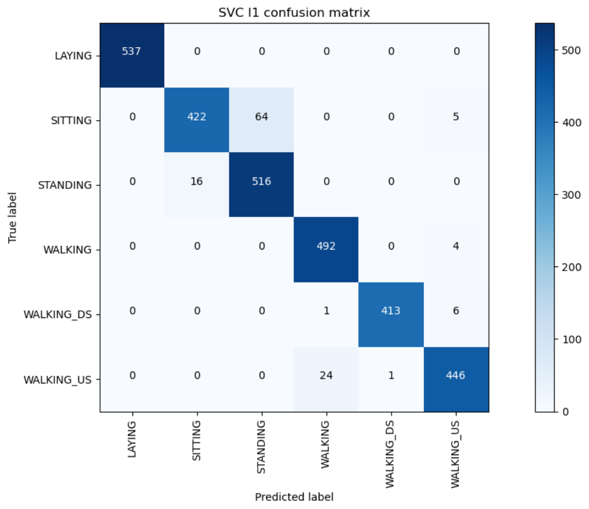


Figure 3.19: Linear SVC with l1 penalty best model confusion matrix

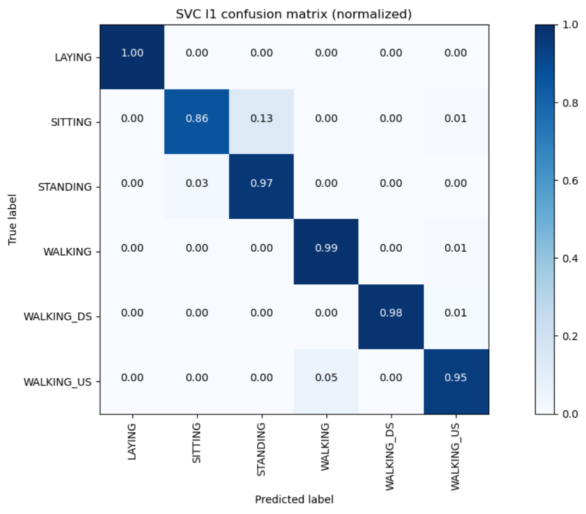
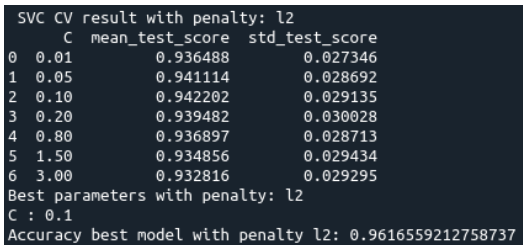


Figure 3.20: Same as Fig.3.19 but normalized

l2 cross validation

- estimator: `LinearSVC` with `penalty = 'l2'`
- parameter space: `C = [0.01, 0.05, 0.1, 0.2, 0.8, 1.5, 3]`
- method to choose the candidates: `GridSearchCV`
- cross-validation scheme: stratified 5-fold
- score function: accuracy



```
 SVC CV result with penalty: l2
      C  mean_test_score  std_test_score
0  0.01         0.936488        0.027346
1  0.05         0.941114        0.028692
2  0.10         0.942202        0.029135
3  0.20         0.939482        0.030028
4  0.80         0.936897        0.028713
5  1.50         0.934856        0.029434
6  3.00         0.932816        0.029295
Best parameters with penalty: l2
C : 0.1
Accuracy best model with penalty l2: 0.9616559212758737
```

Figure 3.21: Linear SVC with l2 penalty cross-validation results table

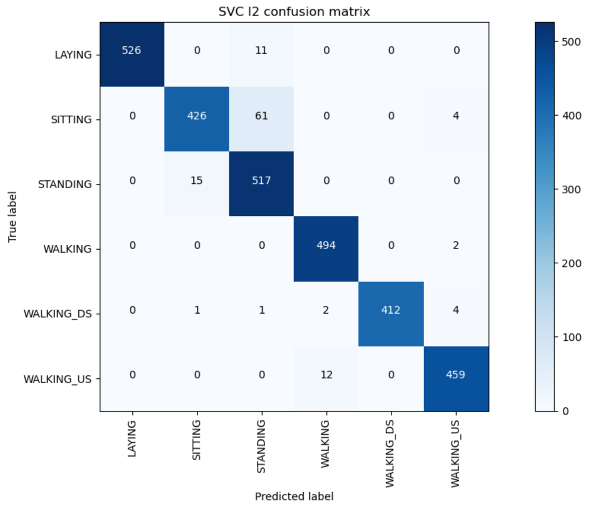


Figure 3.22: Linear SVC with l2 penalty cross-validation best model confusion matrix

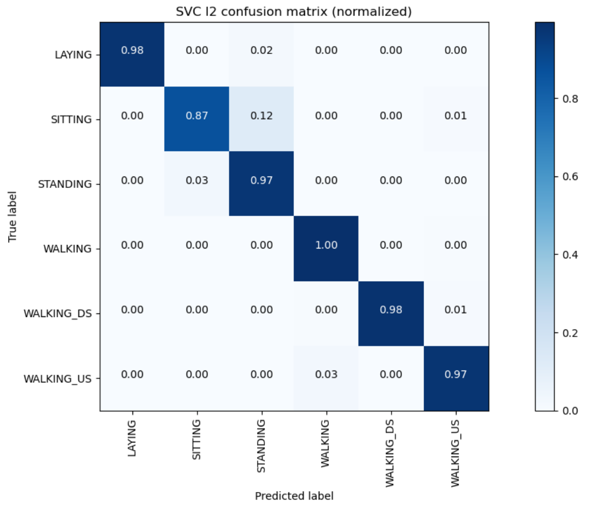


Figure 3.23: Same as Fig.3.22 but normalized

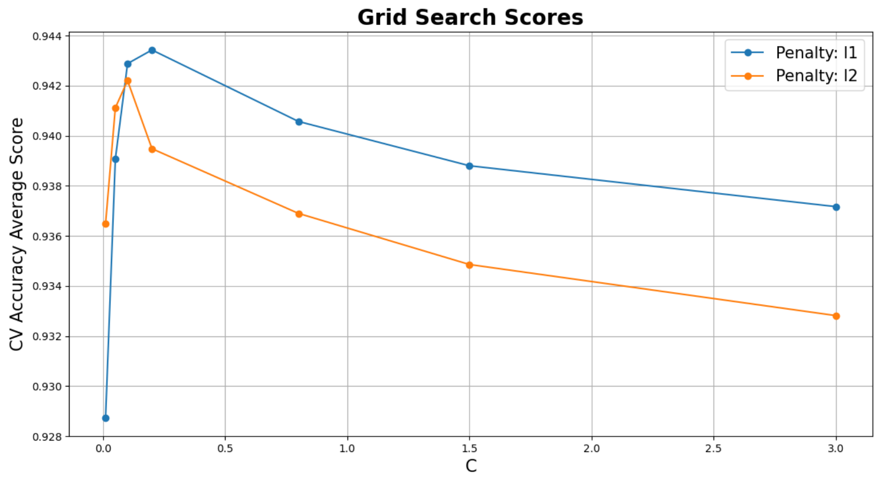


Figure 3.24: Linear SVC cross-validation plot comparison l1 and l2 penalties

3.6 Forests of randomized trees

The `sklearn.ensemble` module implements two algorithms based on randomized decision trees, namely the Random Forests algorithm and the Extra-Trees method, both of them are considered and in this chapter are shown the results.

All the images and tables shown in this section are obtained by running the `code/plot_tree_based.py` file.

3.6.1 Random Forests

The Random Forests is implemented through the function `RandomForestClassifier`, the input variable `criterion = 'gini'` sets the function used to measure the quality of a split, meanwhile the variable `max_features=sqrt(n_features)` imposes that at each split the best split is found in a random subset of size `sqrt(n_features)`.

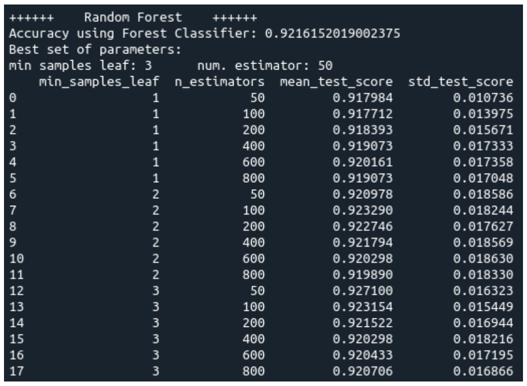
The variable `min_samples_leaf` sets the minimum number of samples that can be contained in a leaf, i.e. before a splitting is done is checked if both left and right branches that would be produced have a number of samples bigger or equal to `min_samples_leaf`. Increasing this variable has a smoothing effect, increase the bias and reduce the overfitting.

Meanwhile, the variable `n_estimators` sets the number of decision trees that composes the random forest estimator.

Hyper-parameters tuning

The hyper-parameters tuning is implemented in the file `code/tree_based.py`.

- estimator: `RandomForestClassifier`
- parameter space: `min_samples_leaf = [1,2,3]`, `n_estimators = [50,100,200,400,600,800]`
- method to choose the candidates: `GridSearchCV`
- cross-validation scheme: stratified 5-fold
- score function: accuracy



```
++++++    Random Forest    ++++++
Accuracy using Forest Classifier: 0.9216152019002375
Best set of parameters:
min samples leaf: 3      num. estimator: 50
    min_samples_leaf  n_estimators  mean_test_score  std_test_score
0                  1            50         0.917984        0.010736
1                  1           100         0.917712        0.013975
2                  1           200         0.918393        0.015671
3                  1           400         0.919073        0.017333
4                  1           600         0.920161        0.017358
5                  1           800         0.919073        0.017048
6                  2            50         0.920978        0.018586
7                  2           100         0.923290        0.018244
8                  2           200         0.922746        0.017627
9                  2           400         0.921794        0.018569
10                 2           600         0.920298        0.018630
11                 2           800         0.919890        0.018330
12                 3            50         0.927100        0.016323
13                 3           100         0.923154        0.015449
14                 3           200         0.921522        0.016944
15                 3           400         0.920298        0.018216
16                 3           600         0.920433        0.017195
17                 3           800         0.920706        0.016866
```

Figure 3.25: Random Forests Classifier cross-validation results

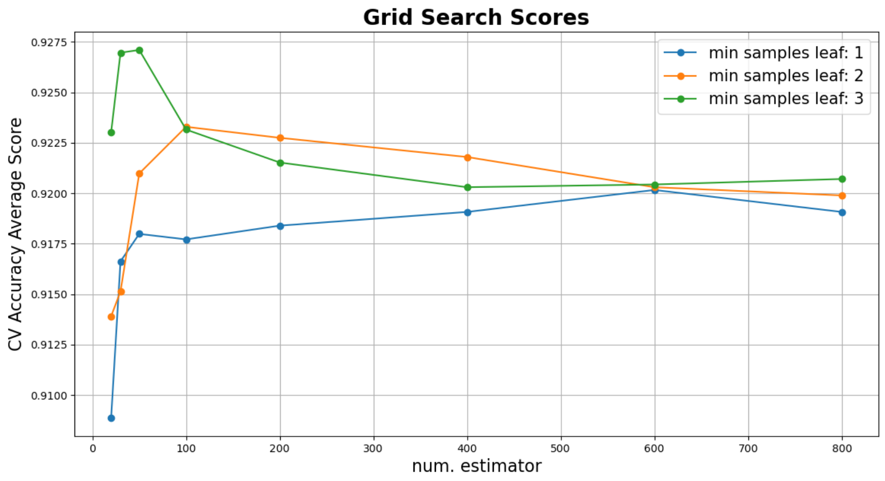


Figure 3.26: Random Forests Classifier cross-validation plot

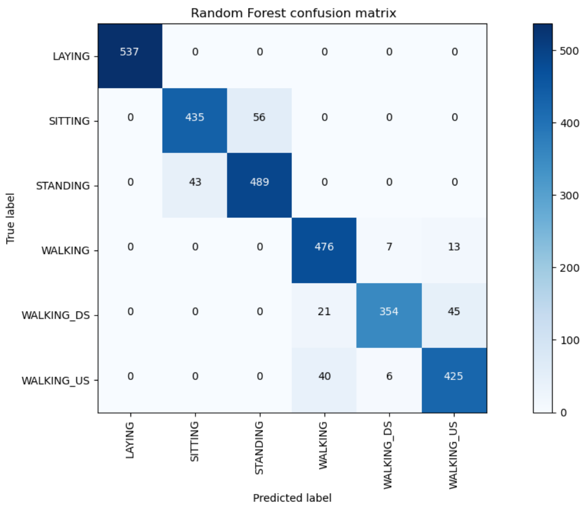


Figure 3.27: Random Forests Classifier best model confusion matrix

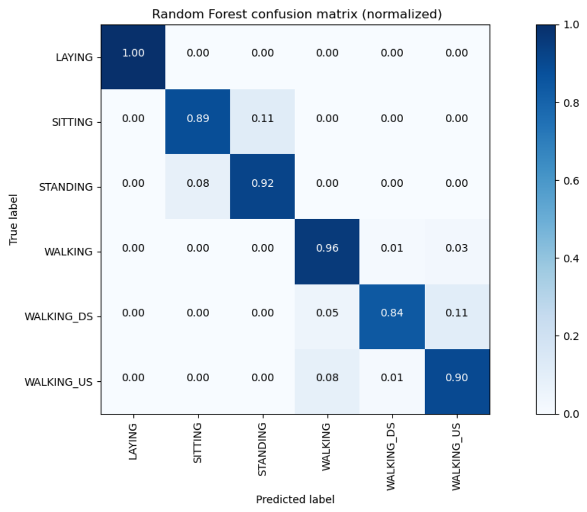


Figure 3.28: Same as Fig.3.27 but normalized

3.6.2 Extremely Randomized Trees

The Extremely Randomized Trees is implemented through the function `ExtraTreesClassifier`, the input variable `criterion = 'gini'` sets the gini index as splitting criterion.

Both the variables `min_samples_leaf` and `n_estimators` as well as `max_features` have the same meaning of the Random Forests case in 3.6.1.

The main difference between the Random Forests algorithm and the Extremely Randomized Trees is that the latter to find the splitting rule consider for each candidate feature[1] a random threshold, in this way for each candidate feature is found a splitting rule, then the purity increasing for each splitting is computed and the best splitting is used.

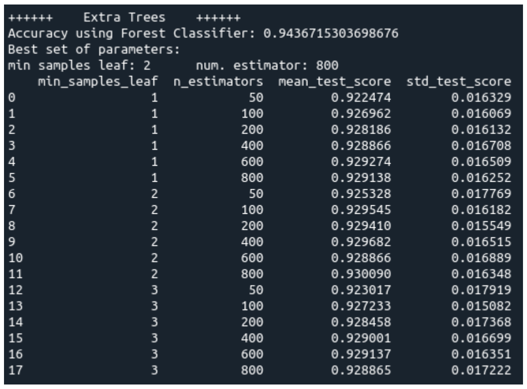
Hyper-parameters tuning

The hyper-parameters tuning is implemented in the file `code/tree_based.py`.

- estimator: `ExtraTreesClassifier`

[1]the number of candidate features is equal to `max_features`

- parameter space: `min_samples_leaf = [1,2,3]`, `n_estimators = [50,100,200,400,600,800]`
- method to choose the candidates: `GridSearchCV`
- cross-validation scheme: stratified 5-fold
- score function: accuracy



```
++++++    Extra Trees    ++++++
Accuracy using Forest Classifier: 0.9436715303698676
Best set of parameters:
min samples leaf: 2      num. estimator: 800
    min_samples_leaf  n_estimators  mean_test_score  std_test_score
0                  1            50         0.922474        0.016329
1                  1           100         0.926962        0.016069
2                  1           200         0.928186        0.016132
3                  1           400         0.928866        0.016708
4                  1           600         0.929274        0.016509
5                  1           800         0.929138        0.016252
6                  2            50         0.925328        0.017769
7                  2           100         0.929545        0.016182
8                  2           200         0.929410        0.015549
9                  2           400         0.929682        0.016515
10                 2           600         0.928866        0.016889
11                 2           800         0.930090        0.016348
12                 3            50         0.923017        0.017919
13                 3           100         0.927233        0.015082
14                 3           200         0.928458        0.017368
15                 3           400         0.929001        0.016699
16                 3           600         0.929137        0.016351
17                 3           800         0.928865        0.017222
```

Figure 3.29: Extremely Randomized Trees Classifier cross-validation results

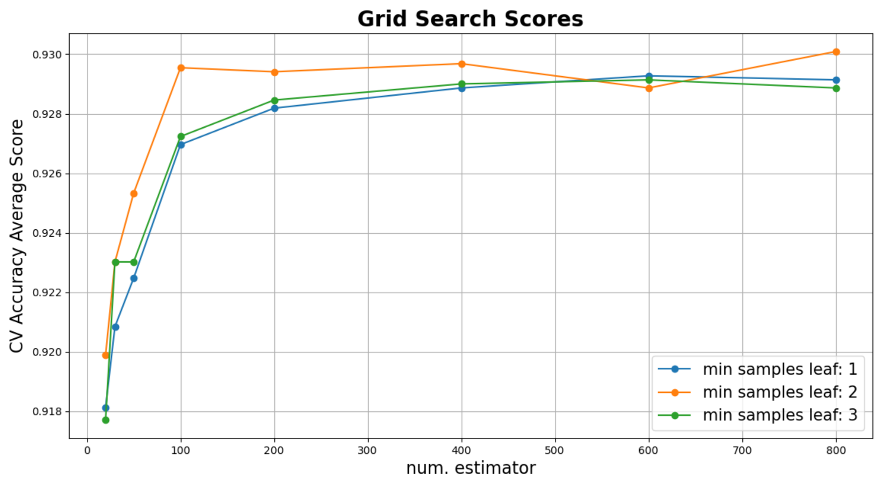


Figure 3.30: Extremely Randomized Trees Classifier cross-validation plot

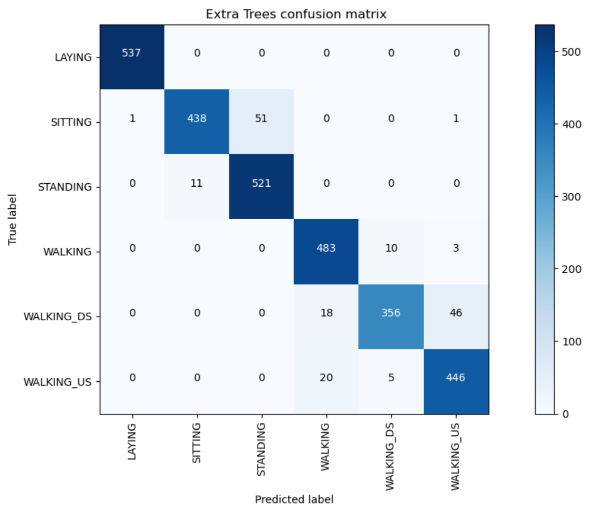


Figure 3.31: Extremely Randomized Trees Classifier best model confusion matrix

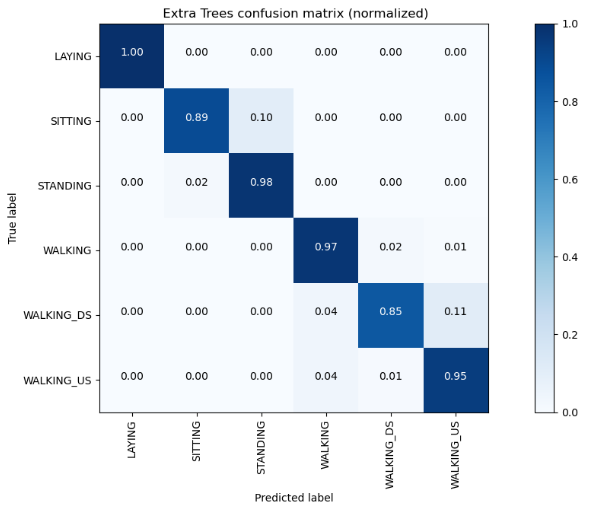


Figure 3.32: Same as Fig.3.31 but normalized

3.7 Stacked generalization

In order to obtain an estimator with less bias can be used the stacked generalization, the stacking classifier is implemented through the function `StackingClassifier`. The stacking classifier is composed by estimators that are called **Base-Models** and by an estimator called **Meta-Model**.

The Base-Models are chosen through the input variable `estimators` that is a list of estimators, meanwhile the Meta-Model is chosen through the input variable `final_estimator`.

The Meta-Model is trained on the prediction produced by the Base-Models through cross-validation, the variable `cv` defines the cross-validation strategy, in this implementation is chosen the 5-fold stratified cross-validation. In other words the original training set is divided into 5 folds (accordingly with the stratified method), then each observation of the original training set is predicted using the Base-Model fitted on the other 4 folds that do not contain the considered observation. Thus each Base-Model provides a prediction that has the role of a feature in the training dataset of the Meta-Model. These cross-validation predictions are computed internally in the `fit` method using the `cross_val_predict` function.

Instead the Base-Models are trained on the whole dataset.

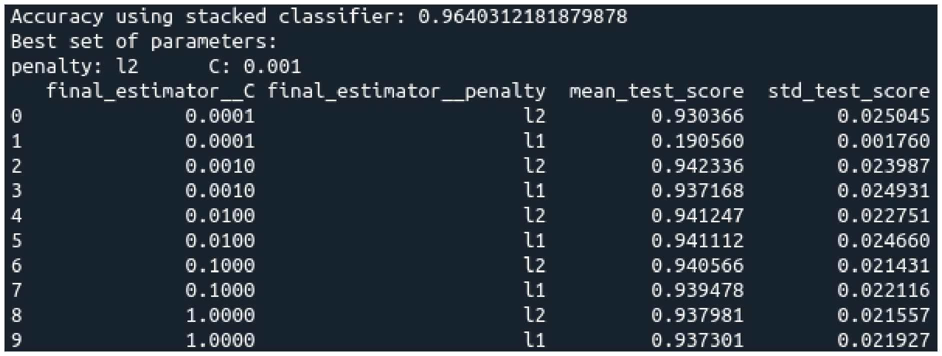
This method combines different estimators and differs from bagging because the models are typically different and are trained on the same dataset. Moreover, is also different from boosting because a single model is used to obtain the final prediction starting from the predictions of the base-models.

All the images and tables shown in this section are obtained by running the `code/plot_stacked.py` file.

3.7.1 Hyper-parameters tuning

The hyper-parameters tuning is implemented in the file `code/stacked.py`. The Base-Models hyper-parameters are not tuned in order to have a smaller parameter space dimension, thus for each Base-Model are used the best hyper-parameters obtained through cross-validation in the previous paragraph. So the underlying assumption is that if the hyper-parameters are a good choice to fit the dataset for the estimator alone, then they are a good choice also to be used as Base-Models in the Stacking Classifier.

- Base-Models:
  `ExtraTreesClassifier`
  `LinearSVC`
  `KNeighborsClassifier` with `PCA` feature selection
- Meta-Model: `LogisticRegression`
- parameter space (parameters of the Meta-Model):
  `C = [0.0001, 0.001, 0.01, 0.1, 1]`
  `penalty = ['l2','l1']`
- method to choose the candidates: `GridSearchCV`
- cross-validation scheme: stratified 5-fold
- score function: accuracy



```
Accuracy using stacked classifier: 0.9640312181879878
Best set of parameters:
penalty: l2      C: 0.001
   final_estimator__C final_estimator__penalty  mean_test_score  std_test_score
0              0.0001                       l2         0.930366        0.025045
1              0.0001                       l1         0.190560        0.001760
2              0.0010                       l2         0.942336        0.023987
3              0.0010                       l1         0.937168        0.024931
4              0.0100                       l2         0.941247        0.022751
5              0.0100                       l1         0.941112        0.024660
6              0.1000                       l2         0.940566        0.021431
7              0.1000                       l1         0.939478        0.022116
8              1.0000                       l2         0.937981        0.021557
9              1.0000                       l1         0.937301        0.021927
```

Figure 3.33: Stacking classifier cross-validation results

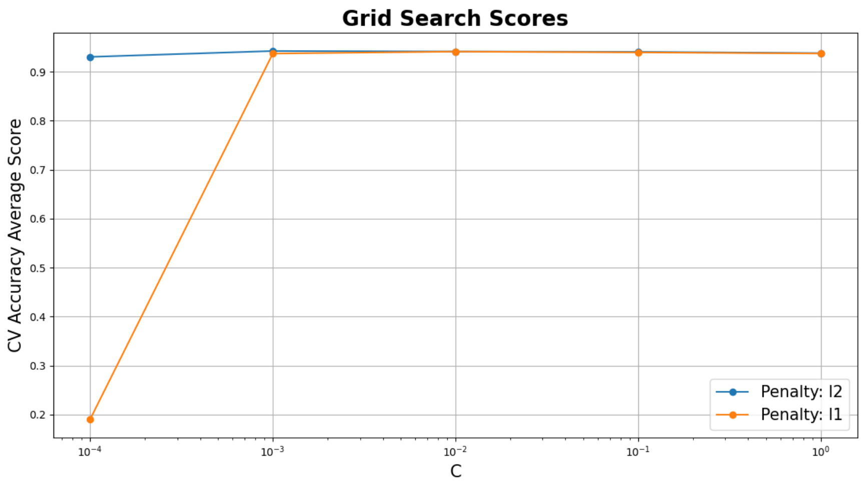
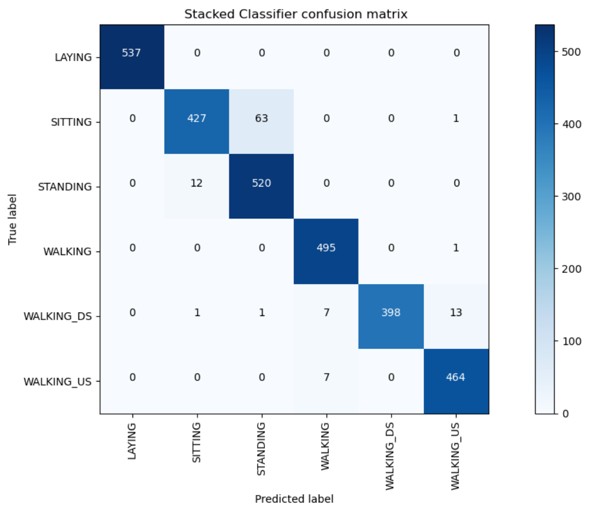


Figure 3.34: Stacking classifier cross-validation plot



True label \ Predicted label	LAYING	SITTING	STANDING	WALKING	WALKING_DS	WALKING_US
LAYING	537	0	0	0	0	0
SITTING	0	427	63	0	0	1
STANDING	0	12	520	0	0	0
WALKING	0	0	0	495	0	1
WALKING_DS	0	1	1	7	398	13
WALKING_US	0	0	0	7	0	464

Figure 3.35: Stacking classifier best model confusion matrix

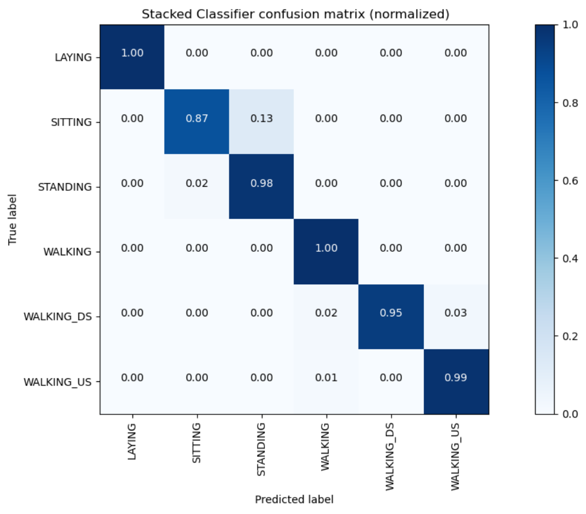


Figure 3.36: Same as Fig.3.35 but normalized

3.8 Comparison of the estimators

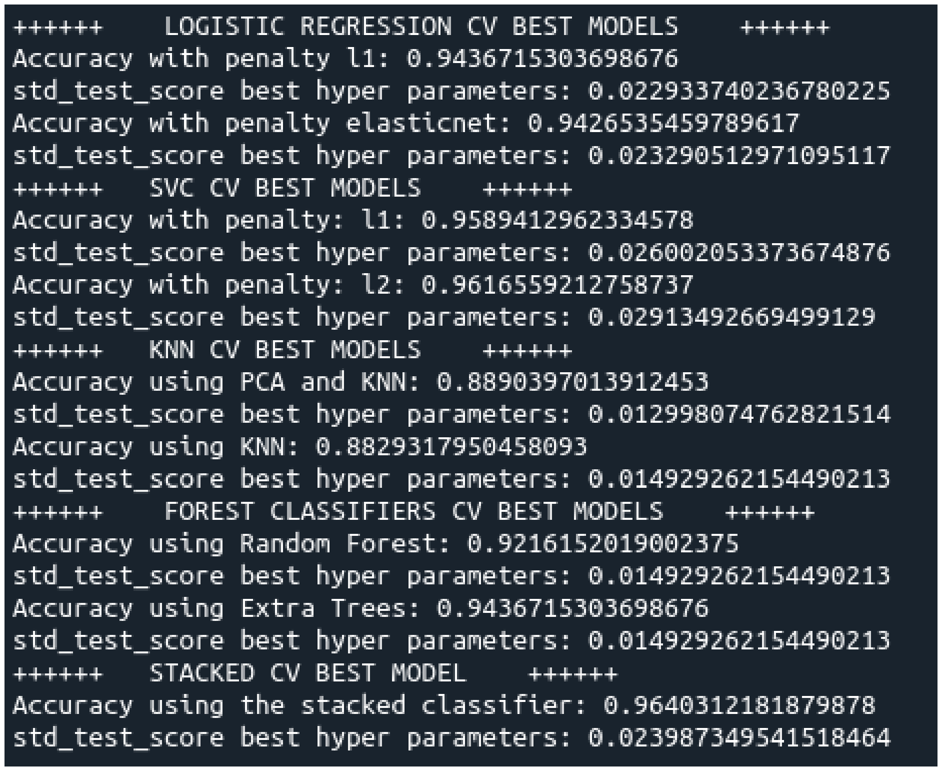
The figures in this paragraph are obtained through the file
`code/plot_estimators_comparison.py`.

3.8.1 Accuracy

The Fig.3.37 provides a comparison of the accuracy scores of the considered estimators. For each accuracy in the next line is reported the variable `std_test_score` associated with the best hyper-parameters, i.e. the hyper-parameters used for the fitting of the final model.

The model with the highest accuracy is the `StackingClassifier`, although the accuracy difference with the `LinearSVC` with penalty `l2` is really small.

Moreover note that the hyper-parameters are chosen via cross-validation considering the parameter `mean_test_score`, but the variable `std_test_score` is not small in most of the presented results, thus for the arguments in Par.3.2.1 the hyper-parameters that gives the highest testing accuracy could be different from the chosen one.



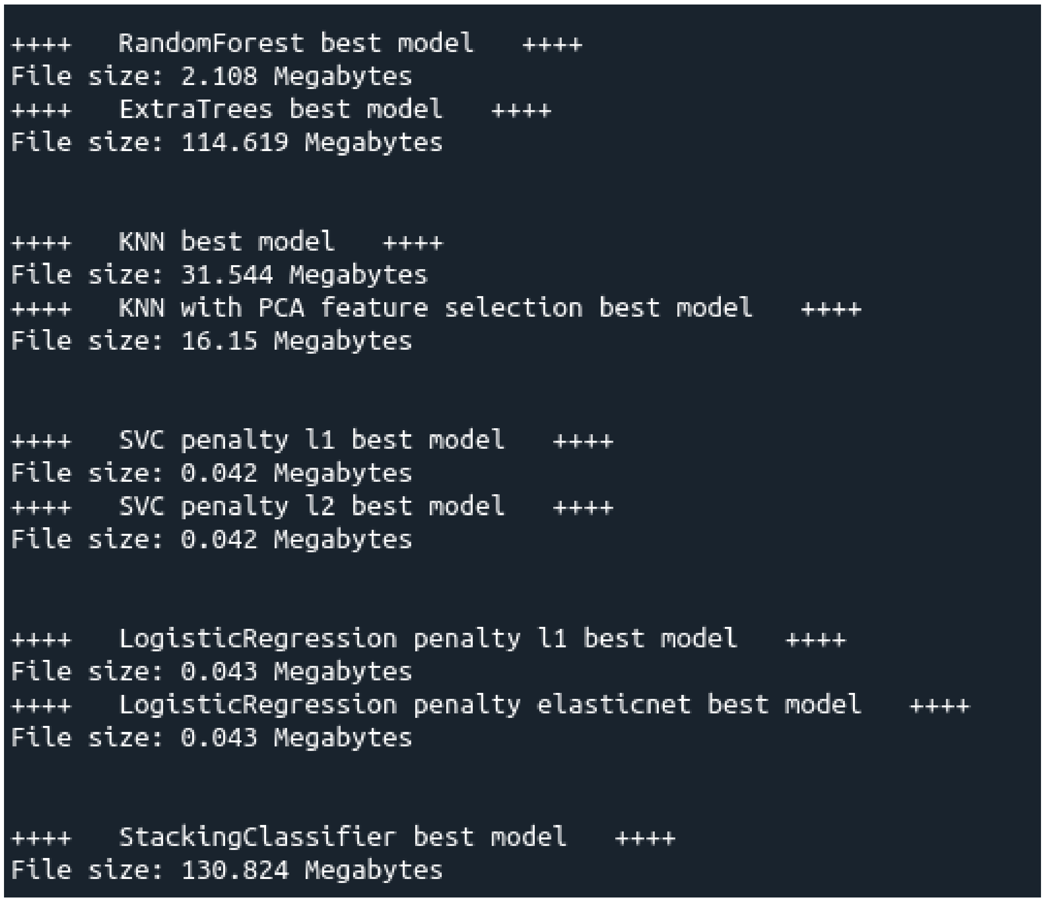
```
++++++    LOGISTIC REGRESSION CV BEST MODELS    ++++++
Accuracy with penalty l1: 0.9436715303698676
std_test_score best hyper parameters: 0.022933740236780225
Accuracy with penalty elasticnet: 0.9426535459789617
std_test_score best hyper parameters: 0.023290512971095117
++++++   SVC CV BEST MODELS    ++++++
Accuracy with penalty: l1: 0.9589412962334578
std_test_score best hyper parameters: 0.026002053373674876
Accuracy with penalty: l2: 0.9616559212758737
std_test_score best hyper parameters: 0.02913492669499129
++++++   KNN CV BEST MODELS    ++++++
Accuracy using PCA and KNN: 0.8890397013912453
std_test_score best hyper parameters: 0.012998074762821514
Accuracy using KNN: 0.8829317950458093
std_test_score best hyper parameters: 0.014929262154490213
++++++    FOREST CLASSIFIERS CV BEST MODELS    ++++++
Accuracy using Random Forest: 0.9216152019002375
std_test_score best hyper parameters: 0.014929262154490213
Accuracy using Extra Trees: 0.9436715303698676
std_test_score best hyper parameters: 0.014929262154490213
++++++   STACKED CV BEST MODEL    ++++++
Accuracy using the stacked classifier: 0.9640312181879878
std_test_score best hyper parameters: 0.023987349541518464
```

Figure 3.37: Considered estimator accuracy

3.8.2 Memory consumption

In Fig.3.38 is shown the memory size occupied by the estimators after the fitting is performed, so is possible to notice that the `LinearSVC` with penalty `l2` has a little lower accuracy respect the Stacking classifier but a much smaller memory size. For this reason in some cases could be preferable to choose `LinearSVC` with penalty `l2` especially if the algorithm needs to be stored in an embedded device in which the memory size of the estimator could be of concern.

However the memory size of the stacking classifier is determined mainly by the presence of the `ExtraTreesClassifier` that has by far the biggest memory size among the considered estimator.



```
++++   RandomForest best model   ++++
File size: 2.108 Megabytes
++++   ExtraTrees best model   ++++
File size: 114.619 Megabytes


++++   KNN best model   ++++
File size: 31.544 Megabytes
++++   KNN with PCA feature selection best model   ++++
File size: 16.15 Megabytes


++++   SVC penalty l1 best model   ++++
File size: 0.042 Megabytes
++++   SVC penalty l2 best model   ++++
File size: 0.042 Megabytes


++++   LogisticRegression penalty l1 best model   ++++
File size: 0.043 Megabytes
++++   LogisticRegression penalty elasticnet best model   ++++
File size: 0.043 Megabytes


++++   StackingClassifier best model   ++++
File size: 130.824 Megabytes
```

Figure 3.38: Memory occupation for each of the best classifier estimator

Chapter 4

Feature Selection

The goal of this chapter is to check if some features can be ignored because their contribution to the prediction task is negligible, thus three methods to select the features are considered.

Each of these three methods provides a feature importance criterion thanks which the features can be sorted by importance. Since the prediction task is of concern, the goodness of a feature reduction method is assessed considering the cross validated accuracy of an estimator that uses the considered method, i.e. once fixed the estimator a feature reduction method is better than another one if using it led to higher accuracy.

The estimator that is fitted on the reduced features dataset is the `ExtraTreesClassifier` because has a small fitting time and since is an ensemble method it will judge the information contained in the features with small bias, moreover is not very sensitive to the hyper-parameters choice (greater number of estimator not led to overfitting).

When the feature reduction is done through the `SelectFromModel` function, the estimator used by this function is use the best hyper-parameters found in the previous cross-validation search. This choice is due to the assumption that a model that predict well then is also good to give the right importance to the features.

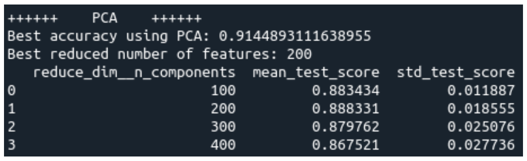
4.1 Chosen methods and results comparison

To consider the variation of the accuracy given by using different numbers of features is used a cross-validation scheme in which the only hyper-parameter to be tuned is the number of features selected by the feature reduction method, namely the numbers of features tested is $[100, 200, 300, 400]$. Thus the estimator has fixed hyper-parameters, namely `ExtraTreesClassifier` has `min_samples_leaf = 1` and `n_estimators = 500`.

The figures in this paragraph are obtained through the file `code/plot_features_importance.py`.

4.1.1 Principal component analysis cross-validation

- estimator: `ExtraTreesClassifier`
- feature selection method: `PCA`
- parameter space (parameters of the feature selection method): `n_components = [100, 200, 300, 400]`
- method to choose the candidates: `GridSearchCV`
- cross-validation scheme: stratified 5-fold
- score function: accuracy



```
++++++    PCA    ++++++
Best accuracy using PCA: 0.9144893111638955
Best reduced number of features: 200
   reduce_dim__n_components  mean_test_score  std_test_score
0                       100         0.883434        0.011887
1                       200         0.888331        0.018555
2                       300         0.879762        0.025076
3                       400         0.867521        0.027736
```

Figure 4.1: Feature reduction with PCA cross-validation results

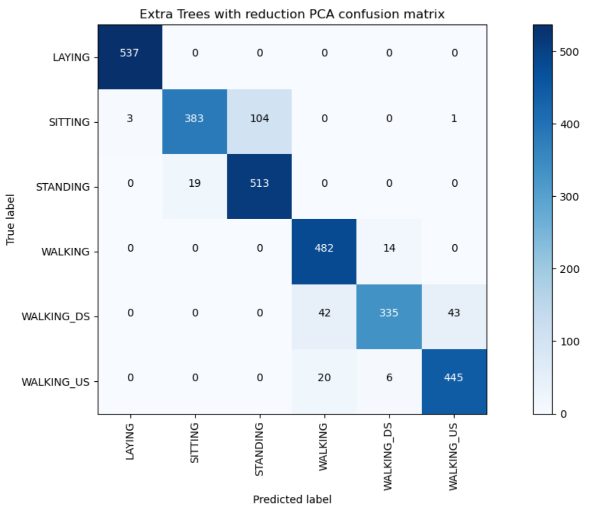


Figure 4.2: Feature reduction with PCA best model confusion matrix

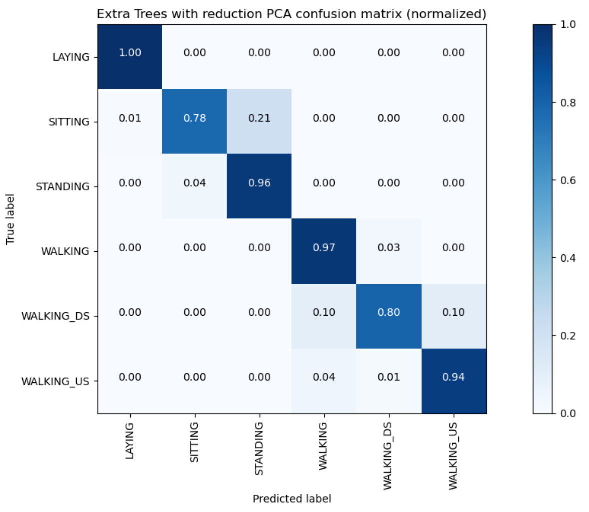
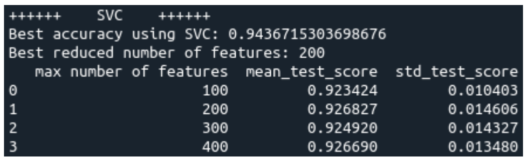


Figure 4.3: Same as Fig.4.2 but normalized

4.1.2 L1-based feature selection cross-validation

- estimator: `ExtraTreesClassifier`

- feature selection method: `SelectFromModel` that uses `LinearSVC` with the best parameters found in Par.3.5.1.
- parameter space (parameters of the feature selection method): `max_features = [100, 200, 300, 400]`
- method to choose the candidates: `GridSearchCV`
- cross-validation scheme: stratified 5-fold
- score function: accuracy



```
++++++     SVC     ++++++
Best accuracy using SVC: 0.9436715303698676
Best reduced number of features: 200
   max number of features  mean_test_score  std_test_score
0                     100         0.923424        0.010403
1                     200         0.926827        0.014606
2                     300         0.924920        0.014327
3                     400         0.926690        0.013480
```

Figure 4.4: Feature reduction with Linear SVC cross-validation results

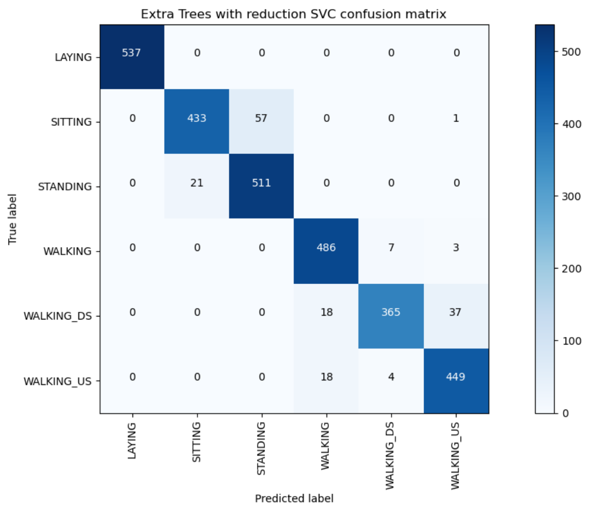


Figure 4.5: Feature reduction with Linear SVC best model confusion matrix

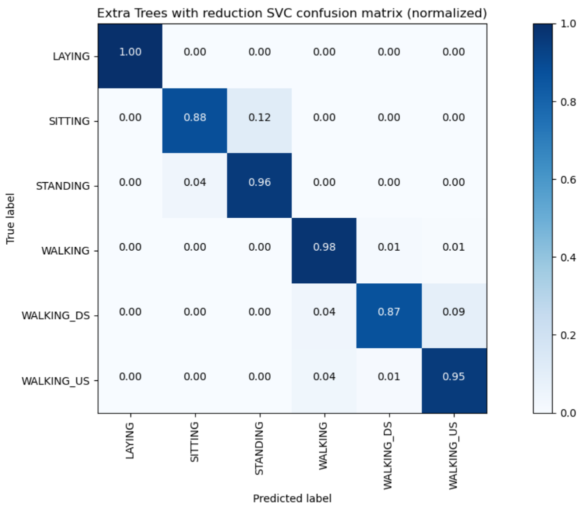
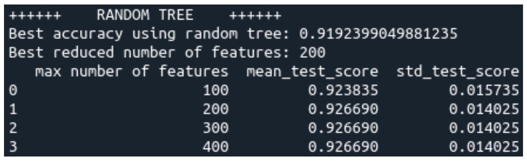


Figure 4.6: Same as Fig.4.5 but normalized

4.1.3 Tree-based feature selection cross-validation

- estimator: `ExtraTreesClassifier`
- feature selection method: `SelectFromModel` that uses `RandomForestClassifier` with the best parameters found in Par.3.6.1.
- parameter space (parameters of the feature selection method): `max_features = [100, 200, 300, 400]`
- method to choose the candidates: `GridSearchCV`
- cross-validation scheme: stratified 5-fold
- score function: accuracy



```
++++++    RANDOM TREE    ++++++
Best accuracy using random tree: 0.9192399049881235
Best reduced number of features: 200
   max number of features  mean_test_score  std_test_score
0                     100         0.923835        0.015735
1                     200         0.926690        0.014025
2                     300         0.926690        0.014025
3                     400         0.926690        0.014025
```

Figure 4.7: Feature reduction with Random Forest cross-validation results

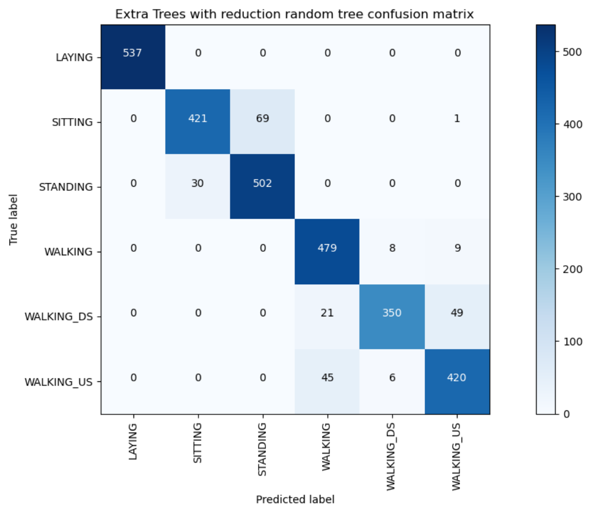


Figure 4.8: Feature reduction with Random Forest best model confusion matrix

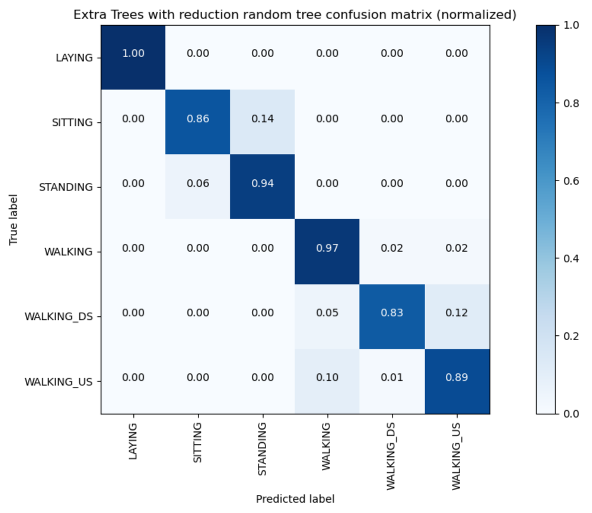


Figure 4.9: Same as Fig.4.8 but normalized

4.1.4 Best features reduction method

In Fig.4.10 are shown the cross-validation results of the three methods, is possible to notice that the better is the `LinearSVC` with `max_features = 200`.

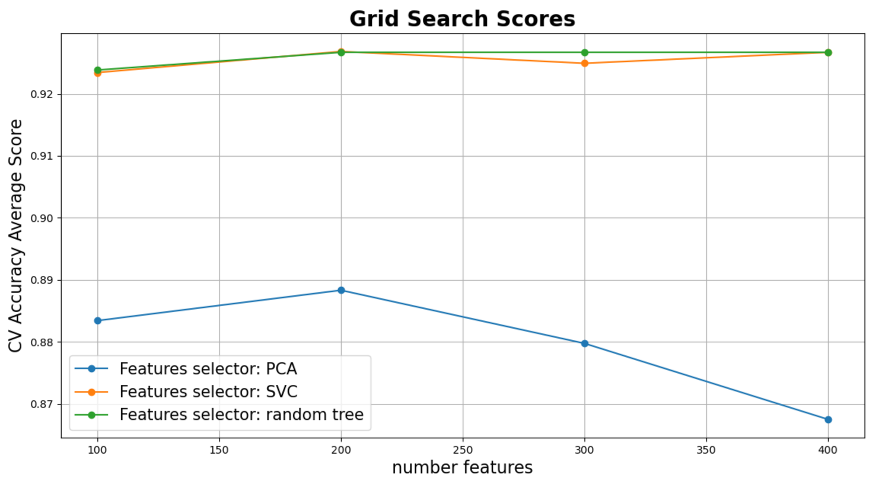


Figure 4.10: Plot to compare the cross-validation result for the three methods

4.2 Features reduction effect on prediction

In this paragraph is considered the effect of the best feature reduction method founded in the Par.4.1 with the best model founded in Chap.3.

The best feature reduction method (i.e. the model that brings a lower reduction of the accuracy respect to the full features case) is the `LinearSVC` with `max_features = 300` while the best estimator in the full features case (i.e. the classifier that has the bigger accuracy) is the Stacking classifier in Par.3.7.

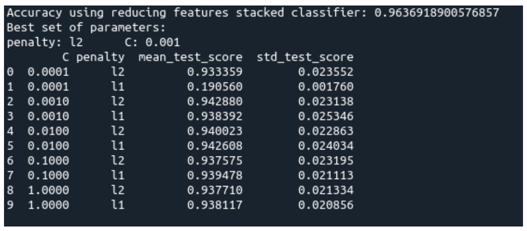
The figures in this paragraph are obtained through the file `code/plot_stacked_reduced_features.py`.

4.2.1 Hyper-parameters tuning

Since the presence of the feature reduction method a cross-validation is performed to check if the best parameters have been changed. The hyper-parameters tuning is implemented in the file
`code/stacked_reduced_features.py`.

The cross-validation scheme is the same of that used in Par.3.7.1:

- Base-Models:
  `ExtraTreesClassifier`
  `LinearSVC`
  `KNeighborsClassifier` with `PCA` feature selection
- Meta-Model: `LogisticRegression`
- parameter space (parameters of the Meta-Model):
  `C = [0.0001, 0.001, 0.01, 0.1, 1]`
  `penalty = ['l2','l1']`
- method to choose the candidates: `GridSearchCV`
- cross-validation scheme: stratified 5-fold
- score function: accuracy



```
Accuracy using reducing features stacked classifier: 0.9636918900576857
Best set of parameters:
penalty: l2      C: 0.001
         C penalty  mean_test_score  std_test_score
0  0.0001      l2         0.933359        0.023552
1  0.0001      l1         0.190560        0.001760
2  0.0010      l2         0.942880        0.023138
3  0.0010      l1         0.938392        0.025346
4  0.0100      l2         0.940023        0.022863
5  0.0100      l1         0.942608        0.024034
6  0.1000      l2         0.937575        0.023195
7  0.1000      l1         0.939478        0.021113
8  1.0000      l2         0.937710        0.021334
9  1.0000      l1         0.938117        0.020856
```

Figure 4.11: Stacking classifier with Feature Reduction cross-validation results

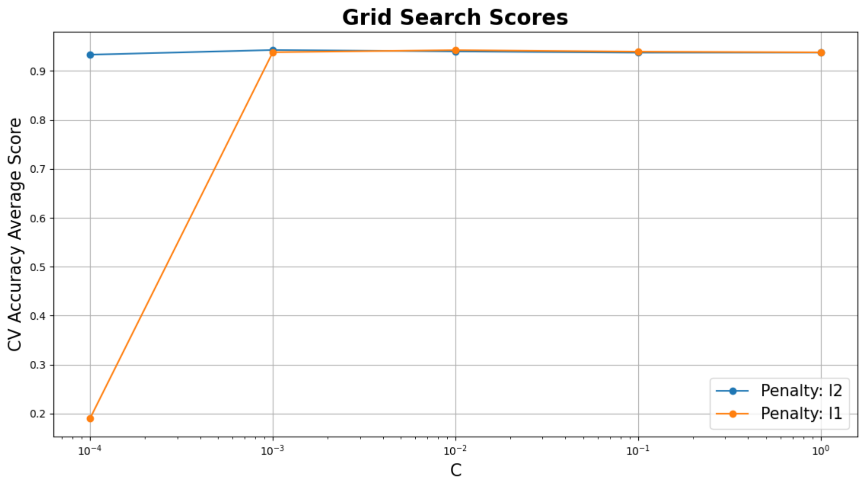
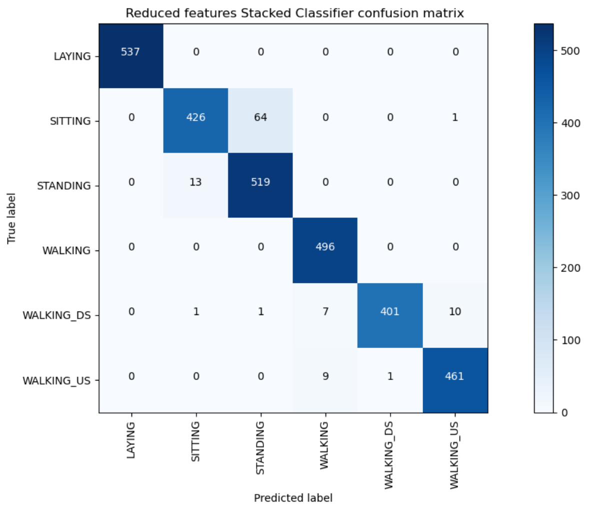


Figure 4.12: Stacking classifier with Feature Reduction cross-validation plot



True label \ Predicted label	LAYING	SITTING	STANDING	WALKING	WALKING_DS	WALKING_US
LAYING	537	0	0	0	0	0
SITTING	0	426	64	0	0	1
STANDING	0	13	519	0	0	0
WALKING	0	0	0	496	0	0
WALKING_DS	0	1	1	7	401	10
WALKING_US	0	0	0	9	1	461

Figure 4.13: Stacking classifier with Feature Reduction best model confusion matrix

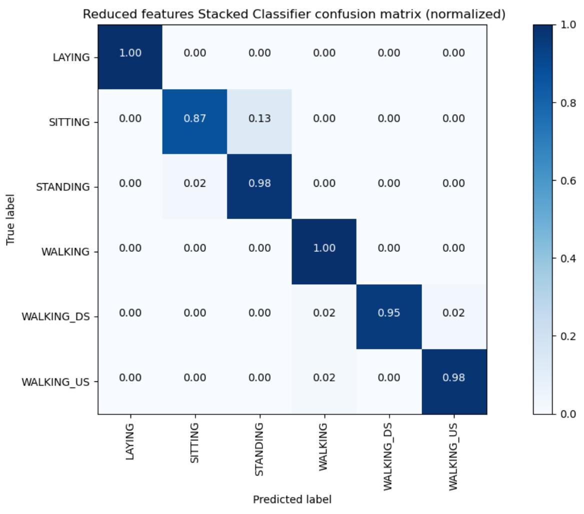


Figure 4.14: Same as Fig.4.13 but normalized

How is possible to see in Fig.4.11 the accuracy reduction is not relevant respect the accuracy in Fig.3.33.

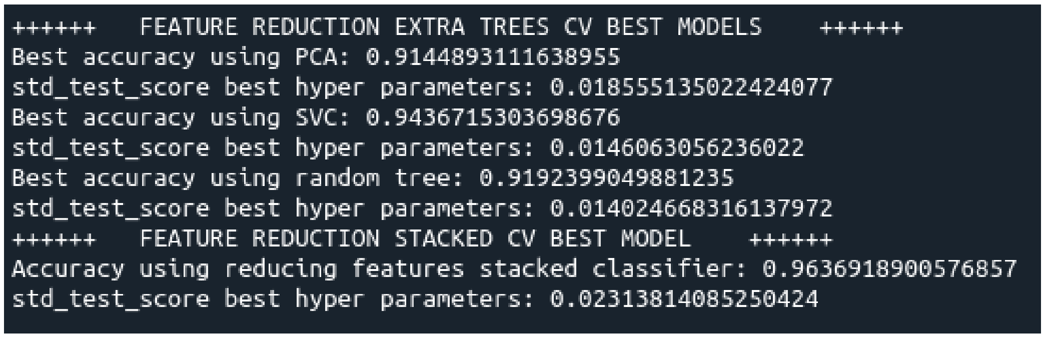
4.3 Comparison of the estimators

The figures in this paragraph are obtained through the file `code/plot_estimators_comparison.py`. The accuracy and memory size of the obtained estimators are shown as done in Par.3.8, and a comparison with the full features case will be done.

4.3.1 Accuracy

Considering the Fig.4.15 is possible to notice that the `LinearSVC` is the model that allows the best feature selection for the `ExtraTreesClassifier`, for this reason is used as feature selector for the Stacking Classifier.

For the variable `std_test_score` can be done the same observation done in Par.3.8.1.



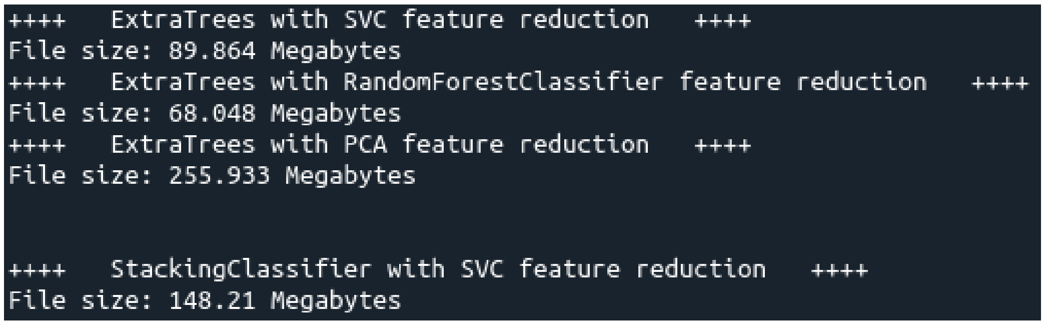
```
++++++   FEATURE REDUCTION EXTRA TREES CV BEST MODELS    ++++++
Best accuracy using PCA: 0.9144893111638955
std_test_score best hyper parameters: 0.018555135022424077
Best accuracy using SVC: 0.9436715303698676
std_test_score best hyper parameters: 0.0146063056236022
Best accuracy using random tree: 0.9192399049881235
std_test_score best hyper parameters: 0.014024668316137972
++++++   FEATURE REDUCTION STACKED CV BEST MODEL    ++++++
Accuracy using reducing features stacked classifier: 0.9636918900576857
std_test_score best hyper parameters: 0.02313814085250424
```

Figure 4.15: Considered estimator accuracy

4.3.2 Memory consumption

Comparing the Fig.3.38 with the Fig.4.16 is possible to notice that the Stacking classifier has a bigger dimension in the feature reduced case. This result seem counter intuitive but since the dimension of the Stacking Classifier is depends mainly by the dimension of the `ExtraTreesClassifier`, then is reasonable to consider the number of terminal nodes of this classifier in the two cases, as shown in Fig.4.17.

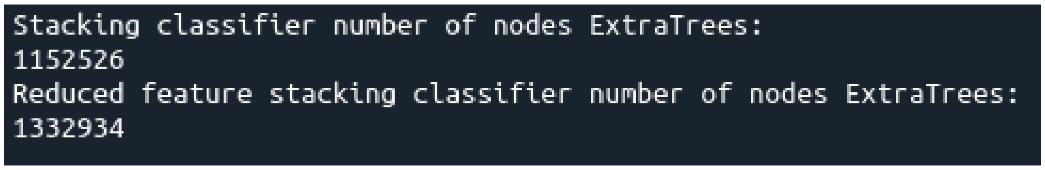
The Fig.4.17 shows the numbers of nodes of the `ExtraTreesClassifier` in the Stacking Classifier for the two considered cases: full features case and features reduction performed via `LinearSVC`. Thus is possible to notice that the number of nodes is bigger in the featured reduced case, and causes the bigger dimension of the estimator.



```
++++   ExtraTrees with SVC feature reduction   ++++
File size: 89.864 Megabytes
++++   ExtraTrees with RandomForestClassifier feature reduction   ++++
File size: 68.048 Megabytes
++++   ExtraTrees with PCA feature reduction   ++++
File size: 255.933 Megabytes


++++   StackingClassifier with SVC feature reduction   ++++
File size: 148.21 Megabytes
```

Figure 4.16: Memory occupation for each of the best estimators



```
Stacking classifier number of nodes ExtraTrees:
1152526
Reduced feature stacking classifier number of nodes ExtraTrees:
1332934
```

Figure 4.17: Number of nodes ExtraTrees in the Stacking Classifier

Chapter 5

Conclusion

Among the considered estimators the `LinearSVC` with penalty `l2` has shown the best performance for the considered dataset. Despite the `StackingClassifier` brings to a little higher accuracy, the memory size of the `LinearSVC` is so small compared with the other estimator that imposes its choice for embedded applications.

Note that the `StackingClassifier` memory size has suffered the presence of the `ExtraTreesClassifier` among its Base-Models. Thus a further development could be to substitute the `ExtraTreesClassifier` with others estimators with low memory size. Maybe this choice will not led to a downgrade of the accuracy score. Anyway analyzing the confusion matrices reported in the previous chapters is possible to notice that the best estimator for the prediction of non-dynamic activities[1] is the `ExtraTreesClassifier`, instead the `LinearSVC` is the most effective for the dynamic activities. This suggests that the use of these two estimator as Base-Models of a Stacking Estimator could led to good results.

All the considered estimator do not have satisfactory performances with non-dynamic activities, this problem could be tackled through the introduction of new features how highlighted in Ref.[1].

The feature reduction analysis shows that some features can be neglected

[1]dynamic activities: WALKING, WALKING UPSTAIRS, WALKING DOWNSTAIRS; non-dynamic: SITTING, STANDING, LAYING

without a significant decreasing of the accuracy, nevertheless a significant reduction in memory size is not achieved.

In conclusion this work shows the capability of the machine learning algorithms to retrieve high level information (as the considered ADL) even using cheap sensors like an accelerometer and a gyroscope embedded in a smartphone.

References

[1] A Public Domain Dataset for Human Activity Recognition Using Smartphones. "Transition-Aware Human Activity Recognition Using Smartphones. Neurocomputing." In: *21th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning* (2013).

[2] UCI Machine Learning. *Human Activity Recognition with Smartphones.* 2019. URL: https://www.kaggle.com/uciml/human-activity-recognition-with-smartphones (visited on Mar. 21, 2021).

[3] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python." In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[4] Scikit-learn. *User Guide.* URL: https://scikit-learn.org/stable/user_guide.html (visited on Mar. 21, 2021).

[5] Smartlab. *Activity Recognition Experiment Using Smartphone Sensors.* URL: https://www.youtube.com/watch?v=XOEN9W05_4A (visited on Mar. 21, 2021).