Biotouch

Project report for the course of Biometric System by

Luca Moschella

Federica Spini

Sommario

Introduction

This document is the report of our project for the course of biometric systems. In this first, brief, chapter our aim is to explain the project idea and how the realization has been done at high level.

Project idea

First of all, to realize our project we had to choose which biometric trait we wanted to use. When we saw the presentation of the project “Biopen”, that has been realized by some students last year, we’ve thought that doing something similar would have been an interesting idea, for a hypothetical comparison of the results and because we liked the topic.

The Biopen project consists in verification of users' identity based on the way they write the word “Biosys”, for Biometric System, thanks to an electronic pen that registers data about single user’s style of handwriting.

The idea of our project, as for Biopen, is to recognize users relying on the way he writes a determined word, but we wanted to use the smartphone touchscreen as recording device. So, we wanted to obtain a system which, given the way a user writes “Biosys” on a smartphone touchscreen can both verify and identify him.

The input data must be taken writing the word dragging the finger on a smartphone touchscreen. So, information about how the user writes a word must be obtained by the smartphone and passed to our biometric system.

We’ve thought this system could be useful to be studied because in hypothetical future similar mechanisms may be used to recognize users in real applications.

Realization process

Here there is a schematisation of how we realized our project and of what this report is about:

- **Realization of the dataset**: we realized an Android application to collect our dataset. It allows users to register the way they write the word “Biosys” and thanks to it we’ve created a dataset. This is the topic of the first chapter of this report
- **Training of the system**: we trained a system using the collected dataset, making it suitable for doing both verification and identification. To do it we used SVMs with scikit-learn and some customized ensemble methods. This is the topic of the second chapter of this report
- **Evaluation of the system:** we analysed the performance quality. In particular we created CMC plot to evaluate the identification task and ROC and FAR vs FRR for the verification. This is the topic of the third chapter of this report
- **Writing the report and presentation:** we wrote this report and some slides that suits for a hypothetical oral presentation of our work

In the last chapter of this report we provide our final conclusions about how our system works and some ideas about future works that can be done to extend this work in the future.

Dataset

The system will use a supervised approach, so we'll need a dataset: pairs of words, written on a smartphone touchscreen, and users who wrote them. Indeed, for a biometric system, a dataset is necessary both to train the model and to test the capability of our program to recognize users.

We've looked for a dataset containing information about how different users usually write the same word on a smartphone touchscreen, but there isn't an available one, so we created the dataset on our own.

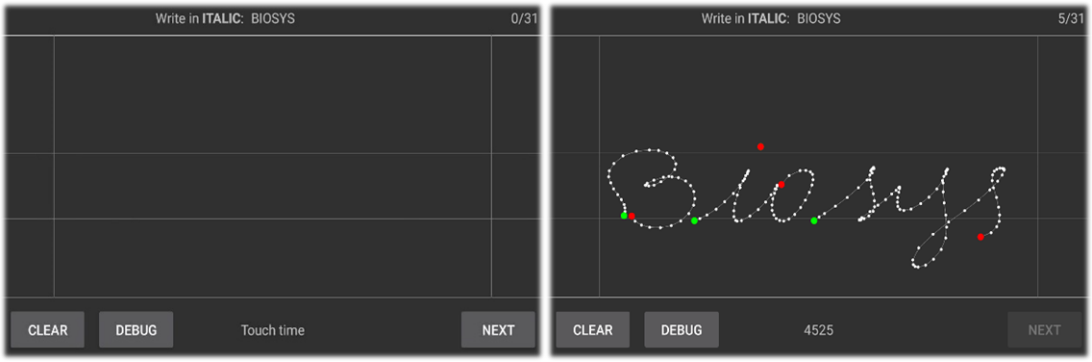
The reasons for doing that should be clear: despite the amount of time required to create the dataset, building our own application would have allowed us to collect data exactly in the form and with the features of the word we needed. So, we've build an android application in order to collect data in the way we established.

In the next paragraphs there are the descriptions of the android application used to create the dataset and its structure.

Android application for collecting data

In this report we won't go deeper through the code of the android application (built with Android Studio), but it is interesting to give a look to the interface.

Home screen of the application where the session data can be insert



Layout where the words can be written

There are few instructions to follow to register data:

- Write the word “Biosys” according to session settings (in italic or block letters)
- Press *clear* if the written word should not be stored (for example if there are errors or there is something wrong)
- Press *next* if the written word is completed and go on recording the next one

When all the 32 words will be registered the application automatically returns to the home screen, so a new session can be started.

The code of the application is available on GitHub here: https://goo.gl/hy66VZ

Data structure

To understand what the application had to do we needed to define the criteria for collecting data. To do so, we asked ourselves the following questions:

- Which are the relevant characteristics that we can extract from a word?
- Which is the better type of data structure to store information in a readable way?
- How many data is appropriate to store?

Answering to these questions allowed us to understand how we wanted the dataset, after that we’ve build the application to collect data.

How many data are in the dataset and why

Our dataset contains the data of 32 different people. Indeed, to estimate how much good a system is in the evaluation phase, it is a good practice to collect at least 30 different classes. Obviously, if we had only two people to distinguish, the problem would be easier and it would become more and more difficult with the growth of the number of people. With at least 30 people we can assume to have a good enough approximation of the complexity of the problem: the difference in complexity between our estimation and the same system trained on a larger dataset is negligible.

We asked to our users to write the word “Biosys” 32 times in italic and 32 times in capital letters. In the end, the dataset is composed by 2048 words, half of which in italic and the other half in capital letters.

What is stored of a word in the dataset

The data collected from a user is subdivided in sessions. Actually, each user did only one session of data gathering, but in general it is useful to support multiple session in order be able to get the data of how a user writes a word in different moments. In a single session the user writes on a smartphone the word “Biosys” 32 times in italic with capital “B” or 32 times in uppercase capital letters. For each session we’ve stored some information about the user:

- **Name and surname**: this represents a class in our system. Also, it helps us to recognize people. Obviously if the dataset would be published the various names should be anonymized
- **Age and gender**: we don’t actually use these features, but we’ve thought that in future analysis it could be interesting to analyse patterns discriminating by age and gender
- **Id**: an id-number of the session which allows to distinguish between session with exactly the same fields
- **Handwriting style**: it indicates if the 32 words collected in the session are written in italic or in block letters

- **Smartphone model**: it indicates the model of the smartphone used to register the session

The application is user-friendly: the user has a feedback on what he's doing, the line starts where he touches the screen, it moves together with his finger and ends when he lifts the finger. The system memorizes a screenshot of the written word, but it doesn't use the image. However, from the data stored from a word an image of the written word can be reconstructed: we don't lose any information.

Words rebuilt with the stored information

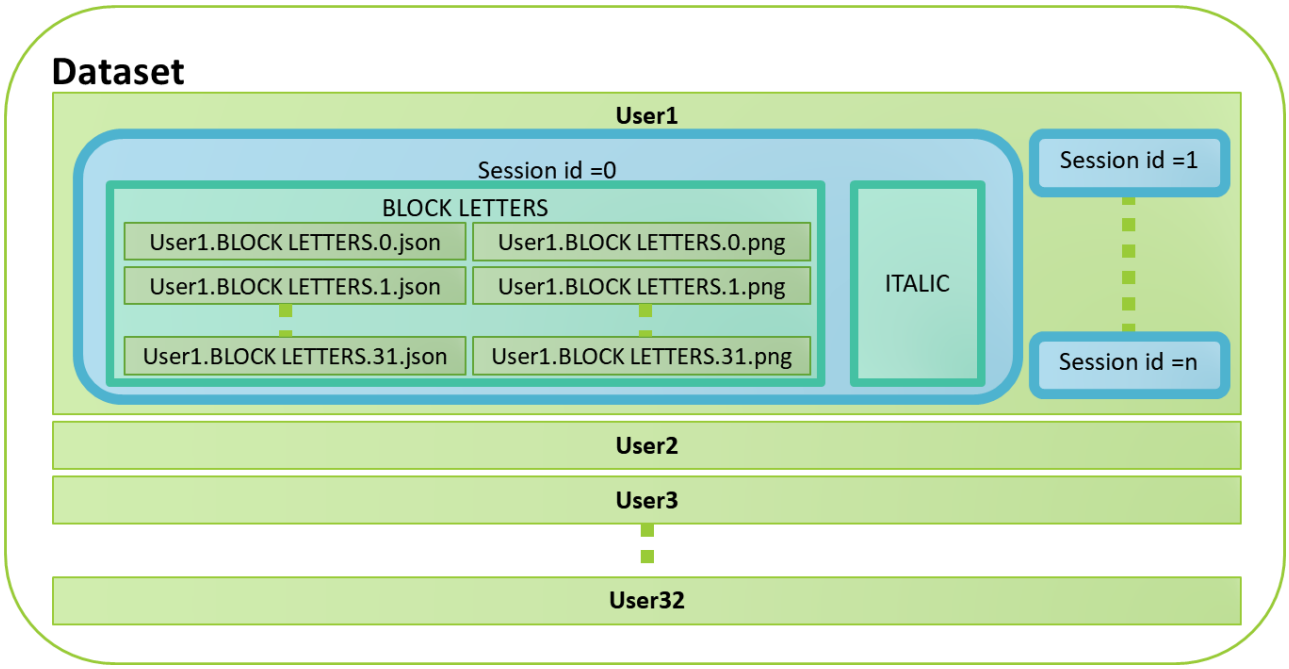
This is possible because for each word the system stores:

- **Movement points**: are the points which are sampled by the application. While the subject is writing a word, with a given frequency determined by the smartphone performances, the points where the finger is going through are extracted. These points are also used to approximate a quadratic Bezier curve, that is displayed on the screen to give a feedback about the written word. To note that the movement points include both the touch down and touch up points
- **Touch-down points**: are the movement points that are the first ones in a certain connected component in a word. Indeed, handwriting is composed by several lines, as "trunk" and the "point" in lowercase "i". So, the points in which the finger start touching the screen are the touch-down points and in the application are green
- **Touch-up points:** are the movement points where the user lifts the finger from the screen. They identify the last point of a connected component in a word and in the application are red
- **Sampled points:** are the points sampled on the quadratic Bezier curve each five pixels, approximated by the movement points. We don't actually use them because they have not been necessary.

For every point its coordinates (x, y) have been collected, they identify the position of the point on the screen. Furthermore, it is memorized the connected component the point belongs to, and the time in which it was written. Note that for the sampled points the time is not available. When a user begins to write a word a chronometer is started and for every point the elapsed time is stored. This characteristic is very important: it transforms the x, y and the connected component in a time series. We'll deepen it in the following chapter.

How information are stored

To explain how our project works it is necessary to say how data are stored by the system. The android application creates a folder called "Biotouch" which contains all the information of the dataset. In "Biotouch" there is a folder for each person whose data have been registered and in each of them there are other folders labelled with session id. Each session contains "ITALIC" folder and "BLOCK LETTERS" folder and here we can find the 32 words stored.

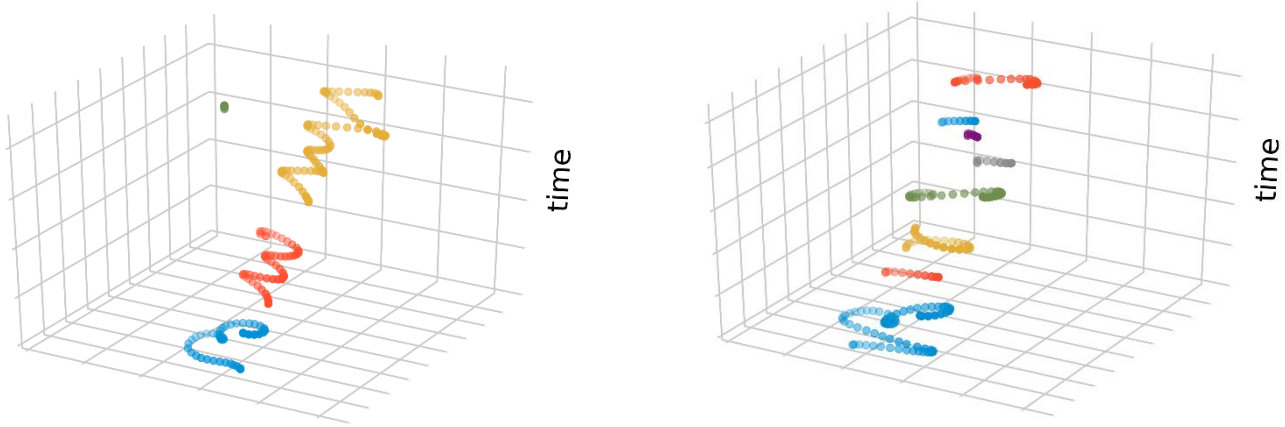


Organization of the dataset

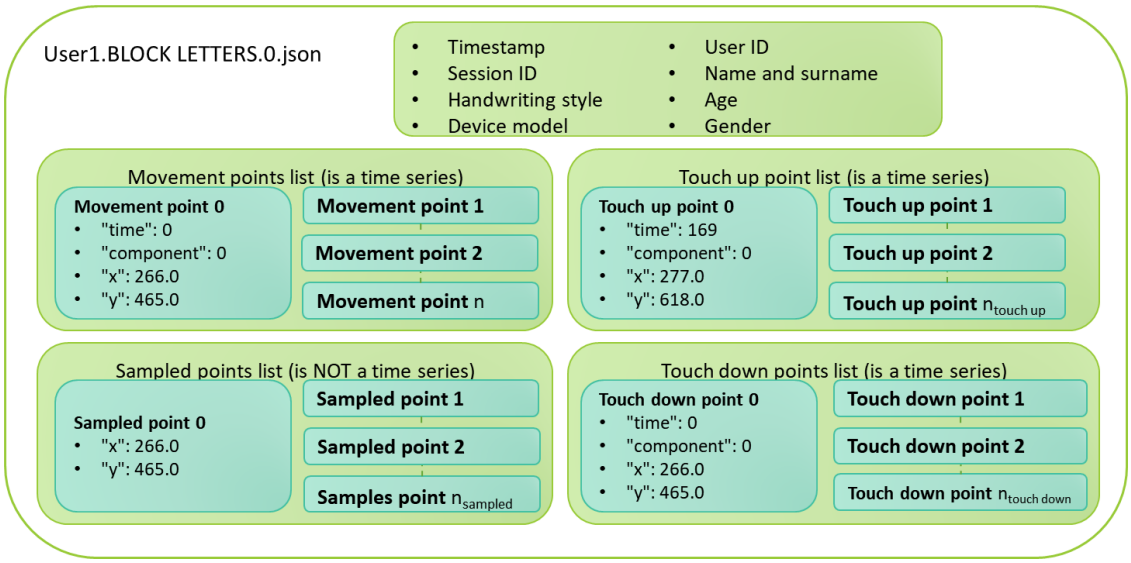
For each word we have stored a screenshot *.png* (that is only to visualize the word, but it will be not used by the system) and a file *.json* that stores all the information of the sampled word:

- **Timestamp value**
- Series of **movement points**, **touch-down points**, **touch-up points**, in which each point has:
  - x value
  - y value
  - time (in millisecond)
  - connected component
- Series of **sampled points**, each of them having:
  - x value
  - y value

So information contained in the .json files are all we need to know about a word. From the time series contained there, we will extract the relevant features.



3D plots to visualize the time



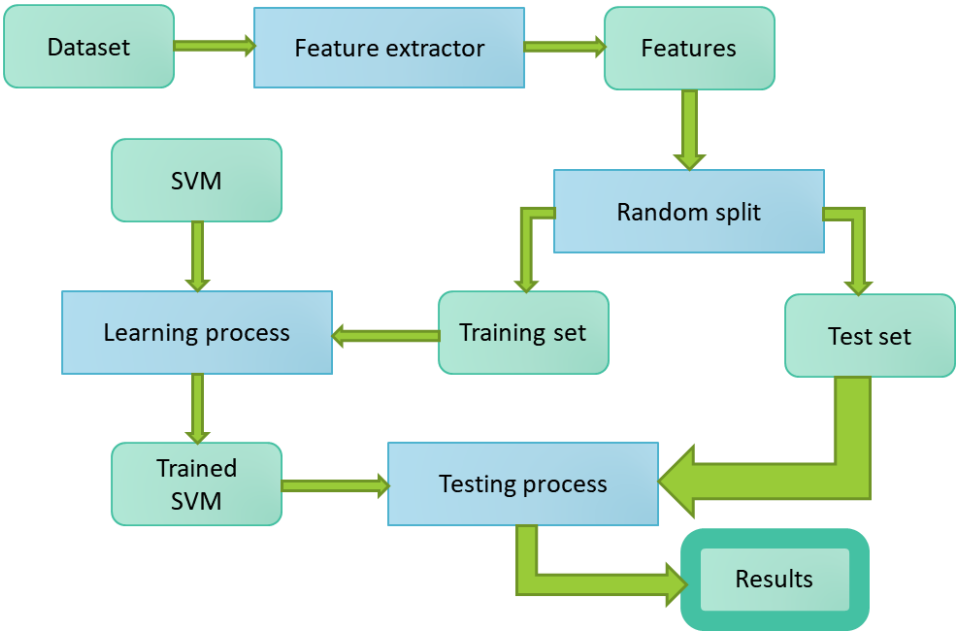
.json file structure schema

In the following section we'll talk about how we have trained a biometric system thanks to the obtained dataset.

Training phase

After explaining how we've collected data, the next step is to make the reader understand how the whole system has been trained. We have used some classical techniques to extract features from the dataset and after that we divided them into training set and test set. Our learner acquired knowledge thanks to the training data, so it has become able to distinguish among people given a sample of a written word on a smartphone touchscreen. To verify how good the resulting system is, we'll use the test set.

To understand in a better way what follows in this chapter, this is general idea of how the system works:



The system follows a supervised learning approach and the classes represent the users who enrolled.

So, the following part of this Chapter will treat:

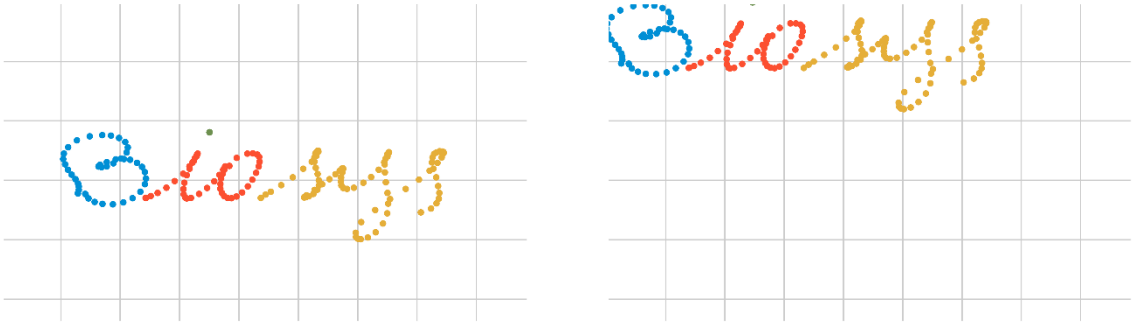
- How the features have been extracted from the dataset
- How the extracted features have been used as training and test set
- Which type of learners we've used

Preprocessing

We were worried that the initial position from where the user starts writing could influence the performances of the system. While it is true that the initial point can be seen as a biometrical trait of the user, it is also true that it can be easily, maybe too easily, influenced by external factors. For example, if a user has seen another one writing on the touchscreen he is inclined to adopt a similar starting point, a user may misunderstand what the guide lines mean, and so on. Moreover, if a user is recognized mostly from where he starts writing, it will surely happen that someday he will write the word in a different position and he won't be recognized.

So we thought that analysing how much the starting point influences the performance would be a point of interest.

We expanded the dataset, adding entries similar to the ones already in the dataset. The only difference is that all the couple of coordinates (x,y) for any point (movement, touch up, touchdown and sample) are shifted as if the word has been dragged to the left upper corner of the screen. In a more technical way we can say that we consider coordinates not related to the smartphone screen, but related to the smaller rectangle which contains the word.



A word before and after the shifting

Features

Summing up what we've said about the dataset structure: we've 1024 words, 32 for each of the 32 people we asked to write "Biosys" on a smartphone touchscreen, both in italics and in block letters.

Three time series of points are registered for each word: movement points, touch up points and touch down points. Other than these, there are the three shifted time series. For every registered point we know:

- the coordinates (x, y) on the screen
- the connected component, that is the stroke it belongs to
- a time value that represent how many milliseconds elapsed since the beginning of the recording of the word

Data have been collected with the same device in order to take points and time with the same policy, indeed the frequency of the sampling is a function of the speed of the smartphone, so it isn't only a matter of resolution.

There are mainly two ways on how to work with time series:

- Dedicated time series model: where the machine learning algorithm incorporates the time series directly, the most famous and used technique is the KNN with DTW
- Feature based approach: here, the time series are mapped to another, possibly lower dimensional, representation. This means that the feature extraction algorithm calculates characteristics such as the average or maximum value of the time series. The features are then passed as a feature matrix to a "normal" machine learning algorithm such as a neural network, random forest or support vector machine. This approach has the advantage of a better explainability of the results. Further it enables us to use a well developed theory of supervised machine learning.

We've chosen a feature based approach, so our first problem has been to extract features from the dataset. To do it, we used tsfresh: this library contains methods for the automatic extraction of features from time series and in particular we used the method *extract_relevant_features*. It has been particularly easy and convenient, because beyond the extraction of the features, this method has also allowed us to distinguish the relevant features from the irrelevant ones. So, we used only the discriminative ones.

For example, from the movement points time series we can extract 2114 discriminative features, involving x, y and the connected component.

It is important to note that it isn't necessary to actually know what the features represent or how they are computed, it is enough to know which are the discriminative ones.

Training and test set

Obviously, from our dataset, we needed to obtain both a training data set and a testing data set. Common sense suggests that samples on which the system has been trained cannot be used as sample to test too. Since most machine learning algorithm are consistent with the training data, it doesn't make sense to perform the testing with the training data. To do this, we've split our dataset into two: the train set and the test set.

We've used methods offered by scikit-learn that allows to make this split in an easy way:

- We divided data using the method *train_test_split* of scikit learn. In particular, we decided to make a partition that, for each class, has the same number of instance in the training set and so also in the test set. A balanced splitting is better to evaluate the resulting system. So, for each class value, that is for each people, we have 22 samples in the training set and 10 samples in the test set. This number has been chosen because it is a good practice to divide data in a 70% for the training and a 30% for the testing phase. We'll train our systems, both for italic and block letters handwriting, on 704 samples and we'll test on 320 samples
- We normalized the features values using a *StandardScaler*, a class of scikit. Using the scaler we preprocessed the features values in the standard way suggested by scikit-learn

It is important to notice that every time we'll rerun the system the split is random, so there isn't a sample that belongs to training set or to the test set: it is decided at runtime.

Type of learner

For biometric systems a machine learning algorithm that has been proved to be very effective are the SVMs, so we decided to use them.

Support Vector Machines can solve binary classification problems, finding a separating hyperplane in order to linearly separate the data. If the data is not linearly separable a kernel trick can be used and, moreover, outliners can be handled with the use of slack variables.

In order to solve a multiclass problem we must use one of these two approaches:

- OneVsRest: it is trained a single SVM for each class, with n classes we get n SVMs.
- OneVsOne: it is trained a single SVM for each pair of classes, with n classes we get (n (n-1))/2 SVMs.

The result is decided through a voting process, in a sort of ensemble method. We decided to use the OneVsRest approach, because the smaller number of SVMs trained implies less time to run the system. Furthermore, we didn't use just an SVM, but we trained several of them and "mixed" their results with different techniques. Basically, we trained an SVM for each of our time series in the dataset, so for movement points, touch up points and touch down points and their shifted version. So we have 6 SVMs and in the code they're called:

- **MOVEMENT**: the SVM trained on the features extracted from the movement points
- **UP**: the SVM trained on the features extracted from the touch up points
- **DOWN**: the SVM trained on the features extracted from the touch down points
- **XY_MOVEMENT**: the SVM trained on the features extracted from the shifted movement points
- **XY_UP**: the SVM trained on the features extracted from the shifted touch up points
- **XY_DOWN**: the SVM trained on the features extracted from the shifted touch down points

SVM tuning

We use the class *GridSearchCV* of scikit-learn to estimate which are the better parameter to configure each one of the SVMs we've talked about. Indeed, it allows to make a search over specified parameter values for an estimator. The parameters of the estimator used are optimized by cross-validated grid-search over a parameter grid. The parameters we've optimized with this method are:

- **Kernel type:** it is always "rbf", Radial Basis Function. After several experiments we noticed that the "polynomial" was never chosen, so we decreased the time of execution discarding this option. Moreover, we studied that for a small value of the gamma the result of a rbf kernel is very similar to a linear model, so we discarded also this parameter
- **Gamma value**: the gamma parameter defines how far the influence of a single training example reaches, with low values meaning 'far' and high values meaning 'close'. When gamma is very small, the model is too constrained and cannot capture the complexity or "shape" of the data. The region of influence of any selected support vector would include the whole training set. The resulting model will behave similarly to a linear model with a set of hyperplanes that separate the centers of high density of any pair of two classes. It can be "auto" or a value from [0, 1e-1, 1e-2, 1e-3, 1e-4, 1e-5, 1e-6] where "auto" means 1/(*number of classes)*
- **C value**: the C parameter trades off misclassification of training examples against simplicity of the decision surface. A low C makes the decision surface smooth, while a high C aims at classifying all training examples correctly by giving the model freedom to select more samples as support vectors. In our system it can be a value from [0.001, 0.1, 1, 10, 100, 500, 1000, 5000, 10000]

Every time that one of our SVMs is trained the grid search tries to offer the best combination of the parameters described above using a k-fold cross validation technique, with k=5.

How to use an SVM

In scikit-learn all the classifiers have always three methods which are essential to use them:

1. **Fit:** performs the training of the model on the given training set
2. **Predict:** returns the best class for each one of the given samples
3. **Predict_proba:** returns a list of probabilities for each one of the given samples. The i-th probability in the j-th list represents the confidence of the j-th sample being of the i-th class

Remember that the SVMs do not provide probabilities! So, to use the *predict_proba* method, we need to estimate them. Scikit-learn supports the estimation of the probabilities through the Platt calibration process.

To note that since the probabilities are estimated, they may be inconsistent with the correspondent predicted class. That is, the predicted class may not have the maximum confidence. Empirical experiments have shown that the probabilities are inconsistent only when the confidence level of the predicted class is similar to the one of, at least, another class.

SVM combined together

To obtain better performances we tried also to combine the previously described SVMs. Indeed, it would have been interesting to analyse how performances would change and we'll see it in the evaluation chapter.

All the considered ensemble methods take a decision relying on the various output of the simple SVMs used. The SVMs we previously described (so MOVEMENT, UP, DOWN, XY_MOVEMENT, XY_UP and XY_DOWN) can be used by these ensemble methods. We can use one of the following policy to obtain a new list of probabilities from which take a decision:

- **Majority**: the final decision of the system is taken with a simple voting method: each SVM votes for its result and the one with the greater number of votes is considered the best. The method *predict_proba* of this type of ensemble will return the result of the *predict_proba* of the SVM which has the highest confidence on the final value. This method, despite being very simple, can be interesting because it can happen that in certain cases an SVM that in general behaves better can get wrong, while two or more SVMs which usually make more error can return the correct output
- **Average:** in this case the final list of probabilities, from which the final decision will be taken, is calculated after *predict_proba* has been called on all the used SVMs. For each class, the resulting probability, that the item belongs to it, is the average of the same probability calculated by the different SVMs. This method works because it is a thoughtful way to mix the results in a fair way. If all the SVMs behave pretty well it returns a sort of shared decision for which every system has the same weight. So if for SVM1 pr(item_class = class_value_i) = 40%, for SVM2 pr(item_class = class_value_i) = 70% and for SVM3 pr(item_class = class_value_i) = 8% we have that the resulting probability that item_class = class_value_i is 39%. The final decision is taken relying on the resulting probability vector
- **Weighted average:** this is similar to the previous method, but what would it happen if we had an SVM better than all the others? In general the SVM that behaves better has greater probability to return the correct answer. For this reason we can obtain a new probability vector in a similar way to the previous one, but giving a weight to the best SVM. We decided to give a higher weight to the movement points. Indeed, after some experiments, we observed that MOVEMENT behave better than the other SVMs.

Our six SVM can be combined in a lot of different ways: precisely we could have tried each of the approaches described above with any combination of SVMs, so we can train $\sum_{i=2}^{6}\binom{6}{i}$*=97* per average and weighted average methods, so 194 SVMs, and $\sum_{i=3}^{6}\binom{6}{i}$=22 per majority method, that makes no sense with only two SVMs. Added the 6 original, we have in total 222 SVMs to train and to compare. They are too many! We've decided to use only a small number of the possible ensembles for reason of time and also to make easier the comparison of results. In particular we've decided to use only the following SVMs:

- **MAJORITY:** majority method with the SVMs MOVEMENT, TOUCH_UP and TOUCH_DOWN
- **AVERAGE:** average method with the SVMs MOVEMENT, TOUCH_UP and TOUCH_DOWN
- **WEIGHTED_AVERAGE:** weighted average method with the SVMs MOVEMENT, TOUCH_UP and TOUCH_DOWN
- **XY_MAJORITY:** majority method with the SVMs XY_MOVEMENT, XY_TOUCH_UP and XY_TOUCH_DOWN
- **XY_AVERAGE:** average method with the SVMs XY_MOVEMENT, XY_TOUCH_UP and XY_TOUCH_DOWN
- **XY_WEIGHTED_AVERAGE:** weighted average method with the SVMs XY_MOVEMENT, XY_TOUCH_UP and XY_TOUCH_DOWN
- **ALL_MAJORITY:** majority method with all the 6 SVMs
- **ALL_AVERAGE:** average method with all the 6 SVMs
- **ALL_WEIGHTED_AVERAGE:** weighted average method with all the 6 SVMs

So, we have in total 6+9=15 SVMs. Indeed, we've used the three ensemble strategies with SVMs trained on the feature extracted from: the standard dataset, the dataset with the shifted words and both of them together.

In the next session we'll explain how it was possible to use these SVMs to perform both identification and verification.

Evaluation

Testing modalities

As said we split the dataset into two independent sets: the training set, used to train the model, and the test set. The performances are evaluated testing the system on the test set.

There are two main task that our biometric system may perform:

- **Identification**: given a sample word the system tries to guess who has written that word
- **Verification**: given a sample word and a user, the system tries to understand if that user wrote that word or not. It uses a threshold to delimitate how much the system must be confident to accept or reject

Both these tasks are implemented using the method *predict_proba*. Given a sample, this method returns a list of probabilities: the probability in the i-th position of the list is the confidence that the system gives to the sample being of the i-th class.

Identification

Performing the identification task is pretty straightforward using the scikit-learn library. Each classifier has a predict method that, given a sample, returns the class value it thinks the sample belongs to. The test set needed is simply the original one, each sample and its true class can be considered a single test; the test is successful if the predicted class corresponds to the true class of the sample.

Using this method, we could only have the recognition rate as a performance evaluator. In order to better evaluate the performances, we've chosen to use a different method.

We use the *predict_proba* method so that we can get a ranking of the classes, from the best one to the worst one.

Verification

In order to perform the verification task it was used the *predict_proba* method, again. Given a threshold, the verification of the couple (sample X, user U) is positive if the probability of the sample X being of class U is greater than the threshold.

In this way we have somehow transformed the multiclass problem in a binary one.

To note that it is a bit more difficult to perform the testing in verification, because in order to test this modality we need couples ((word, user), user). That is a user that claims that the word is his word, and the true author of the word. The verification must be positive if the claimed user is the same as the true one, false otherwise. We generated the test set for the validation in two ways:

- **Balanced**: for each word there is a positive test, the one where the user who claims to have written the word is the true user, and a negative test, where the user who claims to have written the word is randomly chosen among the remaining users.
- **Not balanced**: it has a single positive test, as the balanced one, meanwhile the negative tests are the couples formed by that word and all the remaining users. We called it unbalanced because the negative tests are a lot more than the positive ones.

Results

The plots in this section are generated automatically with *matplotlib*. We'll show some of the plots that we consider more appropriate and important, but the system generates plots for every SVM we talked about. Moreover, must be noticed that every plot is based on an independent simulation, remembering that the split between training and test set is random, it comes with no surprise that results fluctuate a little.

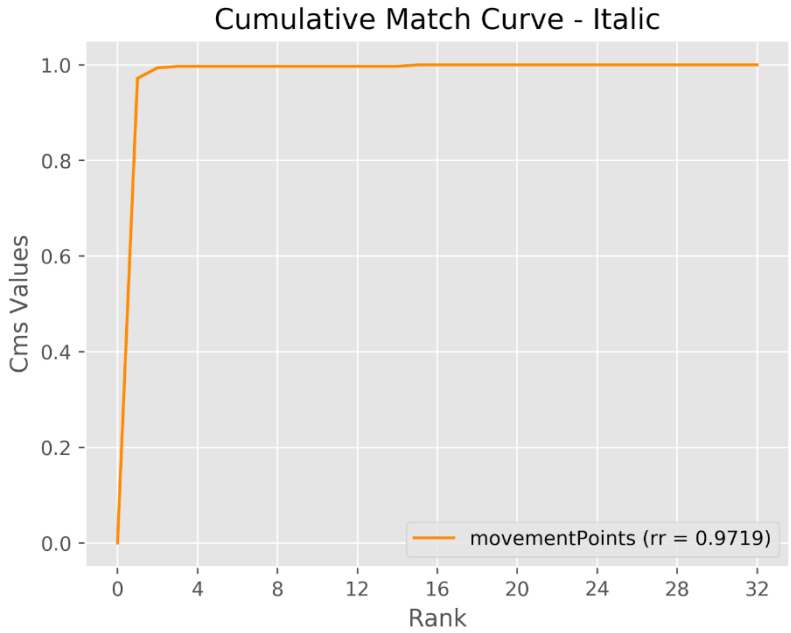
Identification

The ranking of the classes can be easily used to build a Cumulative Match Curve (CMC) to visualize the performances of the system in the identification task.

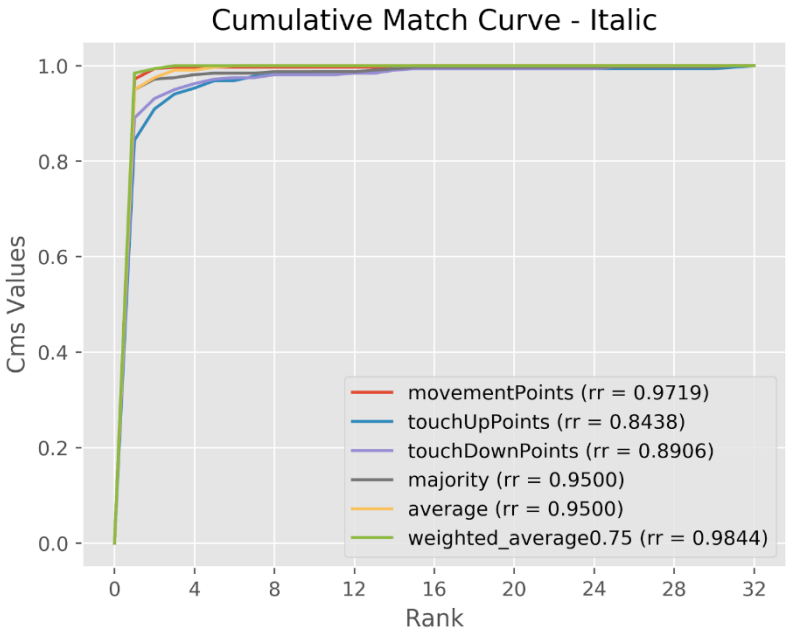
The CMC curve gives an idea on how much the system is getting wrong: it shows for each "rank", a number that goes from 0 to the number of classes, the ratio in which the correct class is in the first rank-positions. This value is also called CMS. Obviously at rank 0 there aren't classes, so the CMS value is assumed to be zero, and at rank 32, the number of classes, the CMS value is always 1. At rank 1 the CMS value represents the Recognition Rate (rr).

We'll see that the system performs extremely well both with Italic and Block letters words.

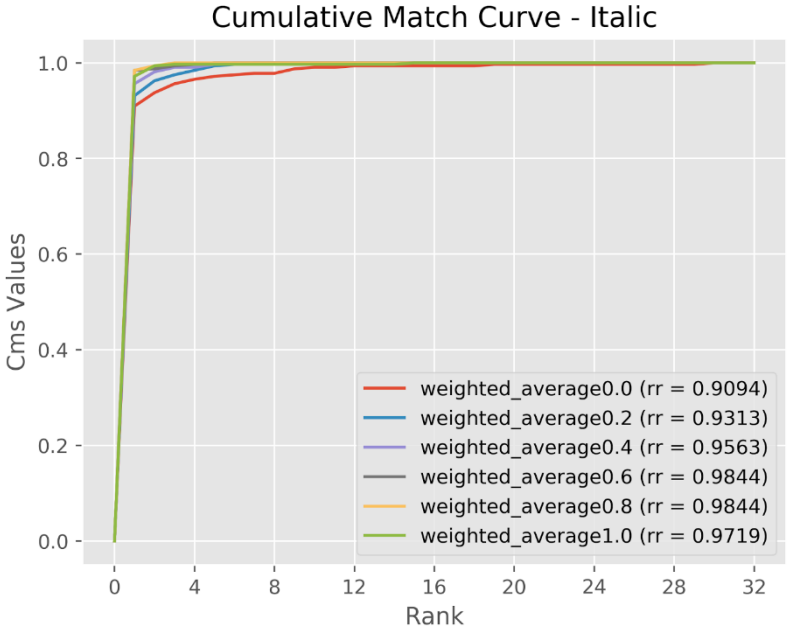
Italic



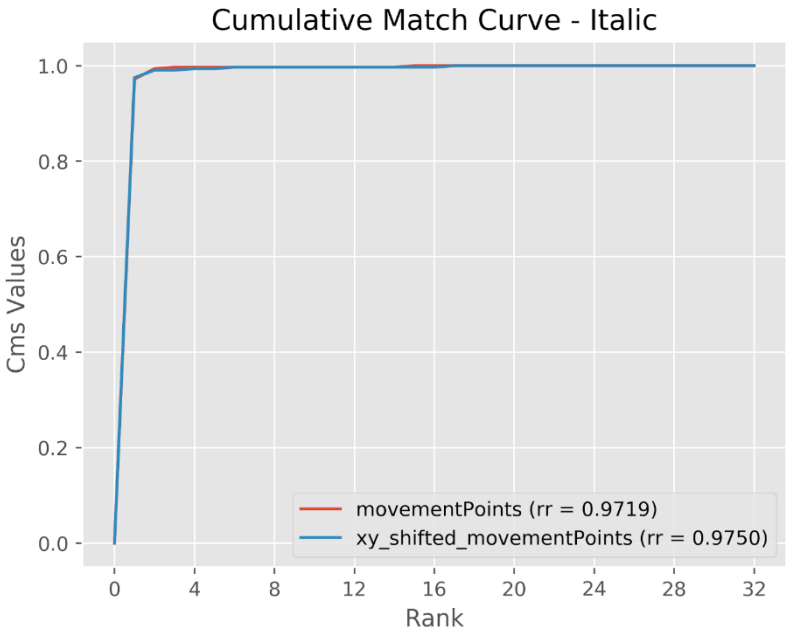
Here we can see that using only the SVM movement points the CMS value goes close to one at rank two. This indicates that this is a very reliable identification system: when it's wrong it is not wrong so much.



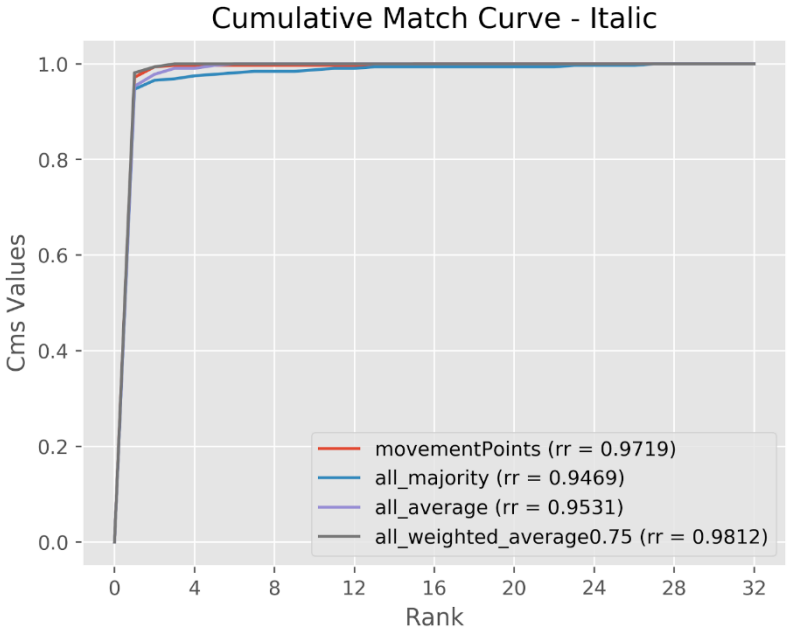
We can see how the movement points is the best simple SVM, meanwhile the best ensemble method is the one that gives more importance to the best simple SVM, the weighted average.



Here, we can see how the weight influences the performances. With a weight equal to zero the movement points SVM is not considered, instead with a weight of 1 the ensemble method is equal to the movement points SVM.

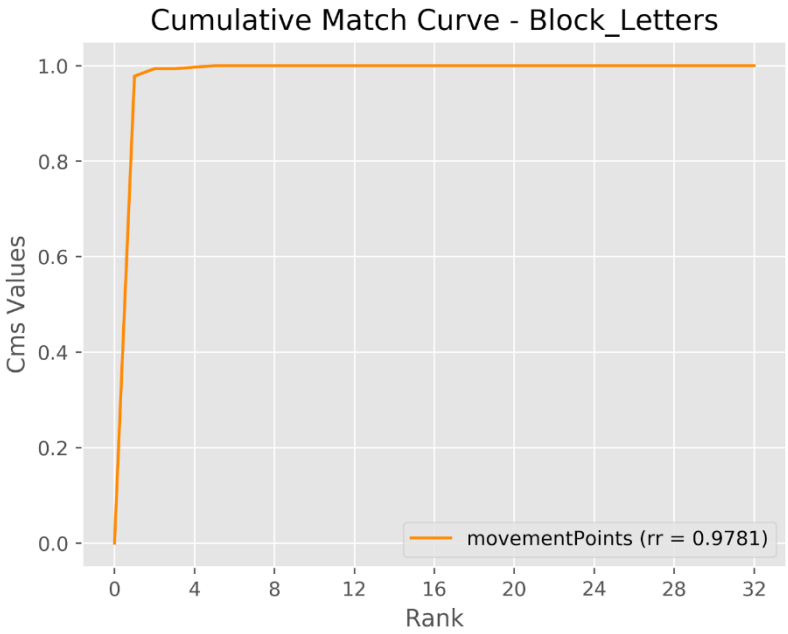


Here, we can see that the performances seem to be independent from the starting point.

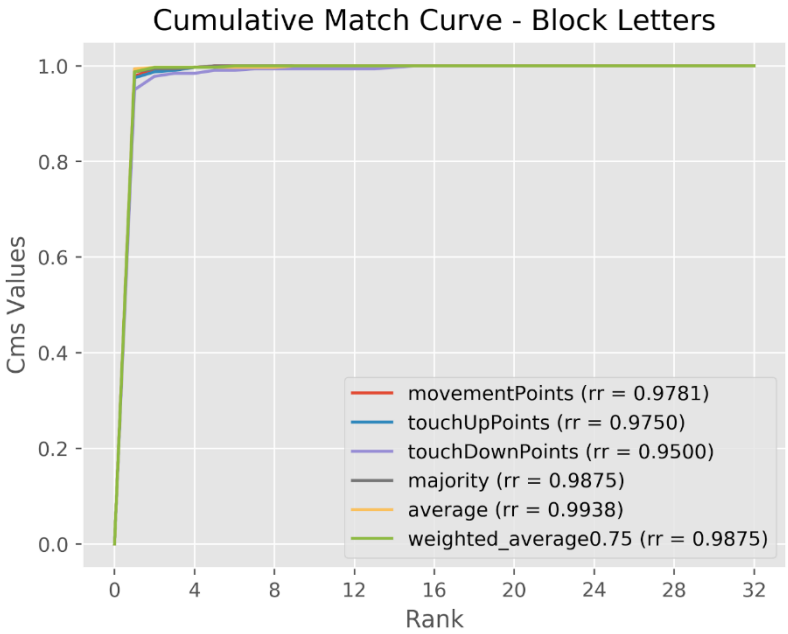


Overall the ensemble methods perform quite well, even the ones that involve all the SVMs.

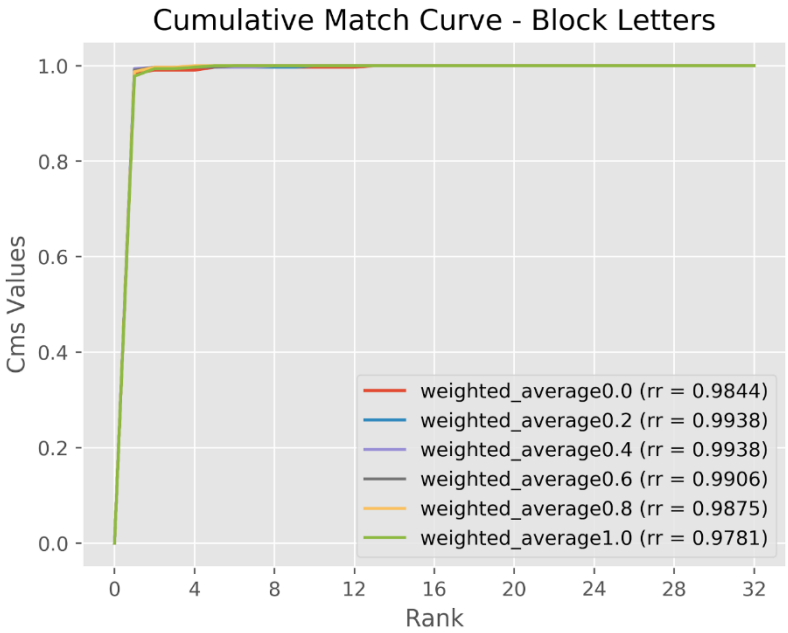
Block letters



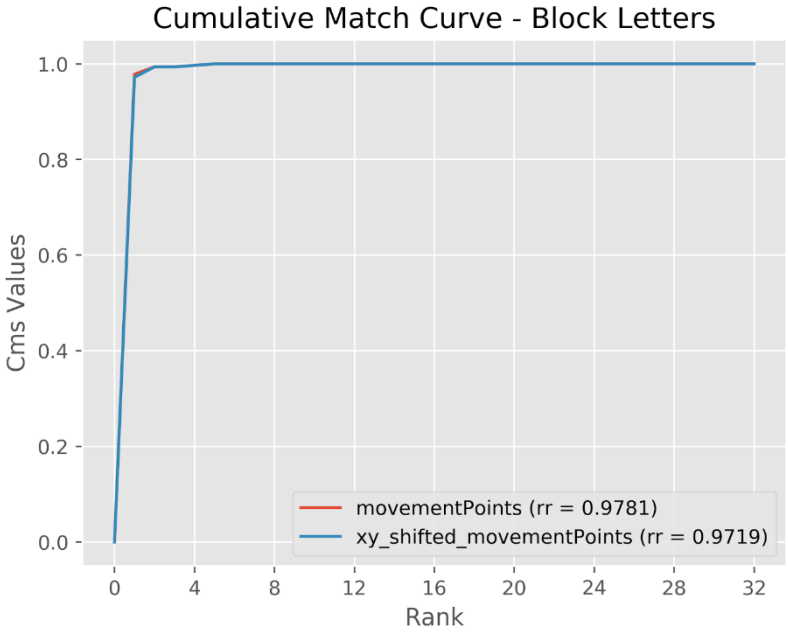
We can do the same analysis for the words written in block letters. Also in this case we can see that movement point SVM works well.



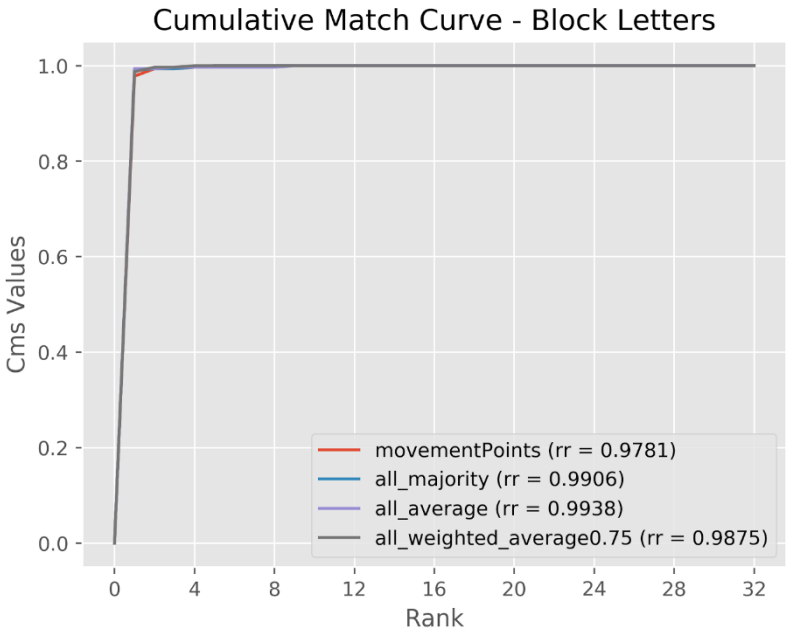
It is interesting to note that usually here the SVMs working on the touch up and touch down points work better, probably because here the correspondent time series are longer and thus give more information about the user.



It is interesting that, since all the simple SVMs perform better, even the ensemble methods perform better! In particular we can see how the best weight is lower than the one used in the Italic setting, and the overall performance is better. In the previous case the best weight was 0.8, now the best weight is about 0.3.



Here as well the influence of the starting point is negligible.



Moreover, the global ensemble SVMs:

- ALL_MAJORITY
- ALL_AVERAGE
- ALL_WEIGHTED_AVERAGE

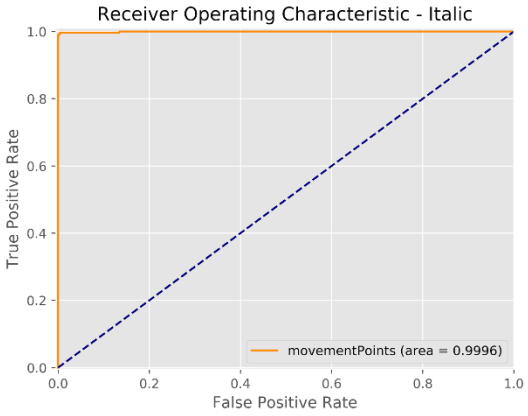
do not provide substantial improvements over the focused ensembled method

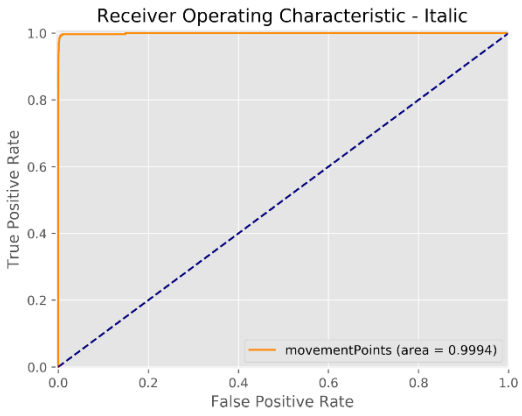
Verification

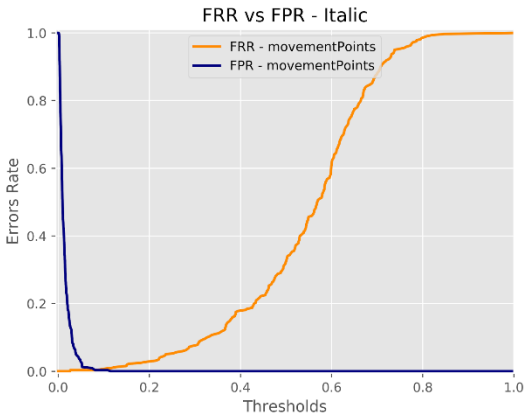
It turns out that scikit-learn provides useful methods to automatically compute the false positive rates, the true positive rates at different threshold values. Given these values it is easy to compute the false rejection rate.

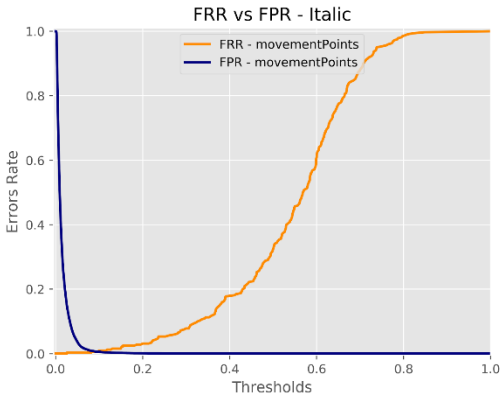
With this information the ROC curve and the FAR vs FRR curve can easily be plotted. The ROC curve is a graphical plot that illustrates the performances of a binary classifier system, as its discrimination threshold is varied. A ROC curve can be summed up by its area-under-curve value, the greater the better. The FAR vs FRR curve is simply the plotting of the FAR and FRR as the discrimination threshold is varied. The point where the two curves meet is the Equal Error Rate, and it is usually the best value for the threshold. The lower the EER the better.

First of all, we can observe that the balanced and not balanced modality do not change the performance estimation of the system. On the left there is a balanced test, meanwhile on the right a not balanced one.





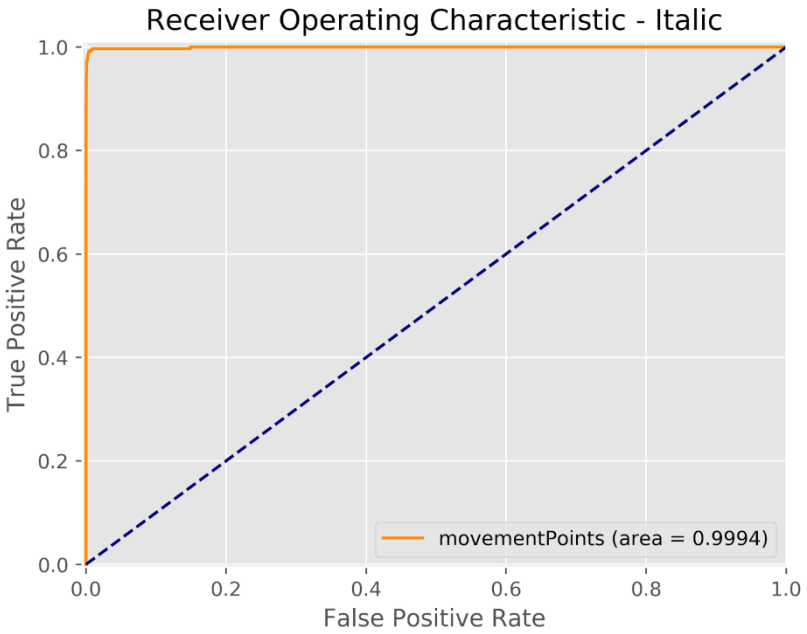




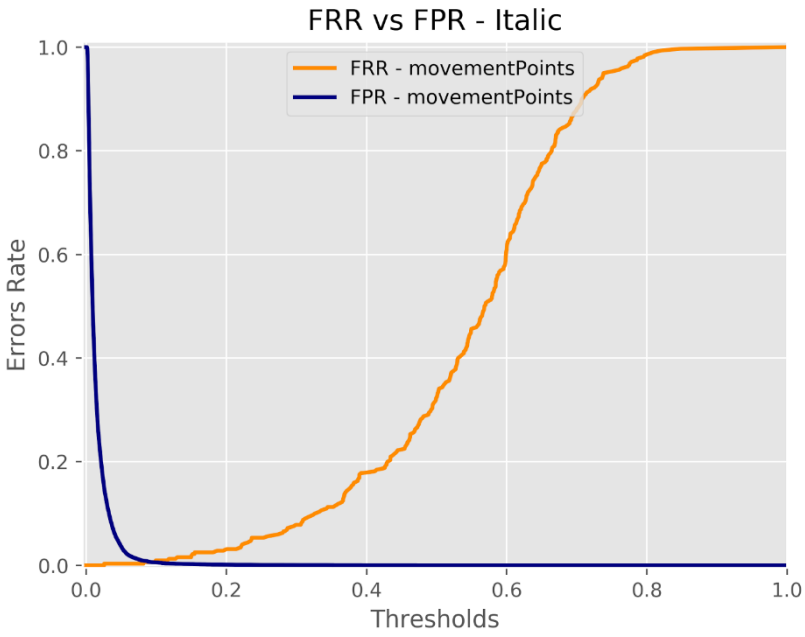
It is interesting to notice that the FPR is smoother on the not balanced case: obvious, because there are more examples. Since the difference between the balanced and not balanced cases is negligible, we'll focus on the not balanced case, because it gives smoother and better approximated curves.

We'll see how the system performs extremely well both in italic and in block letters even in verification mode.

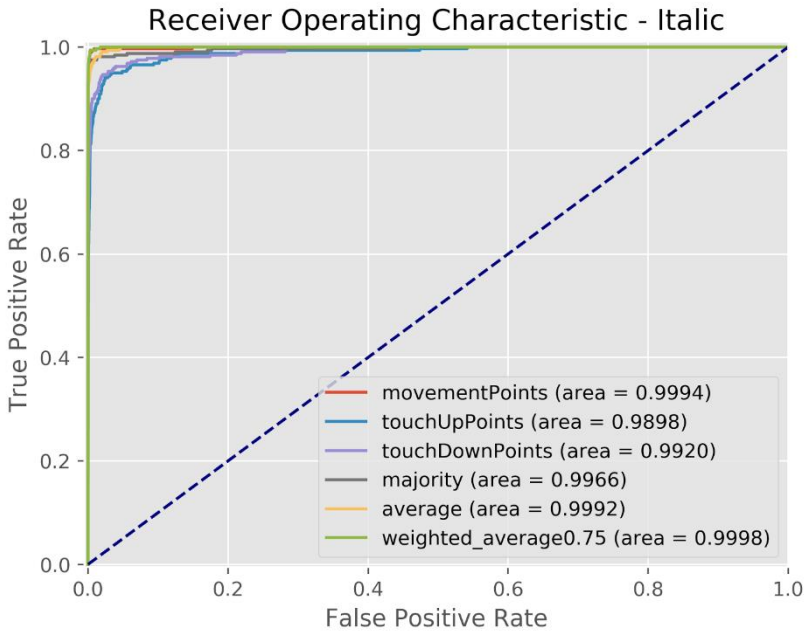
Italic



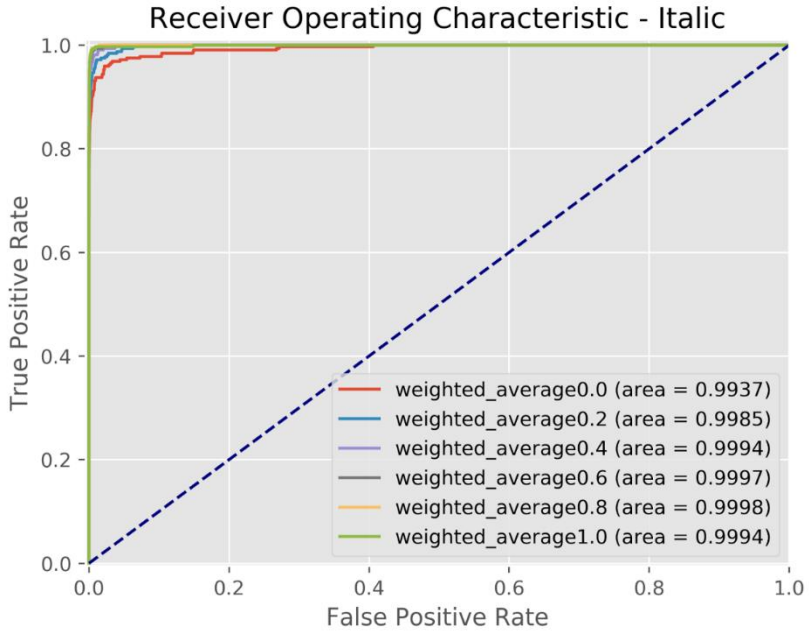
Even in this case the best performances are given by the movement points. The ROC curve has an area close to 1.



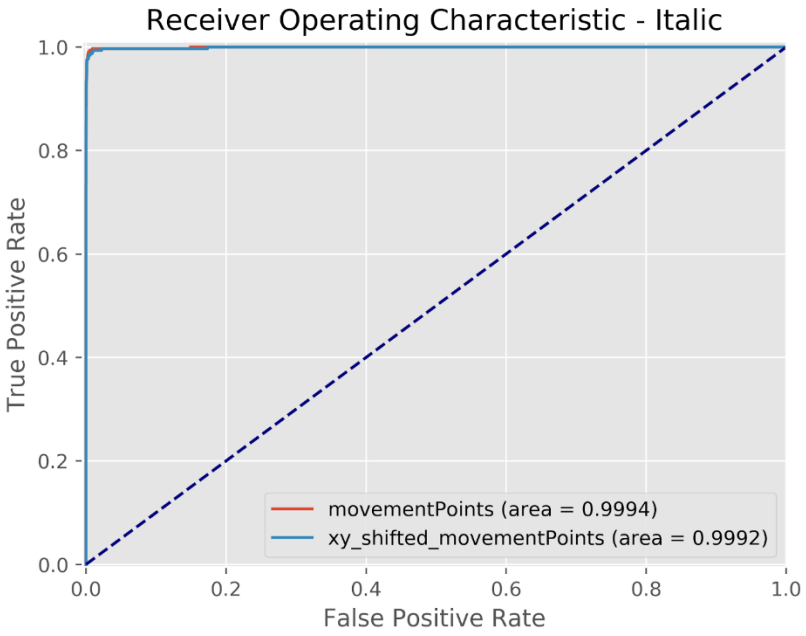
According to what is said above, we confirm that movements points behave well: the ERR is close to 0.



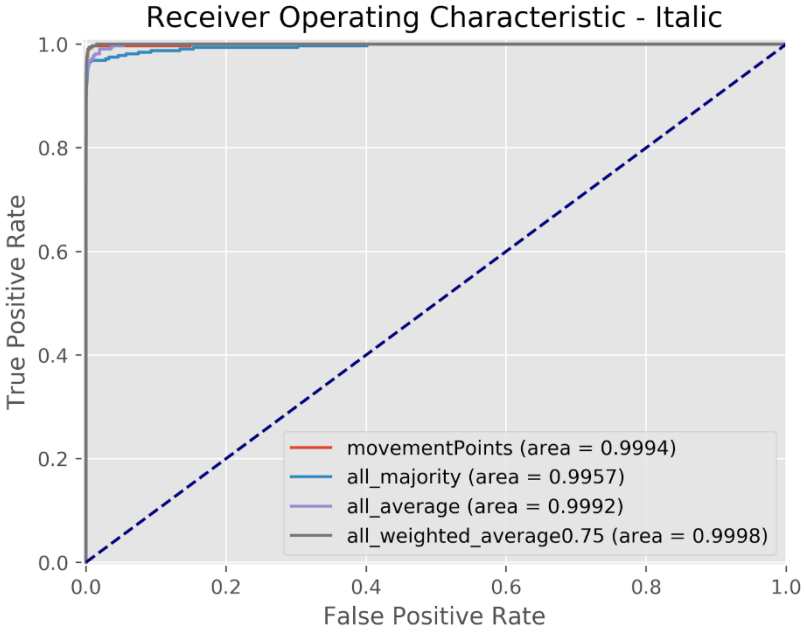
We can compare the movement points with the other SVMs: even here the best results come with the movement points SVM or the ensemble method that gives more importance to the movement points SVM.



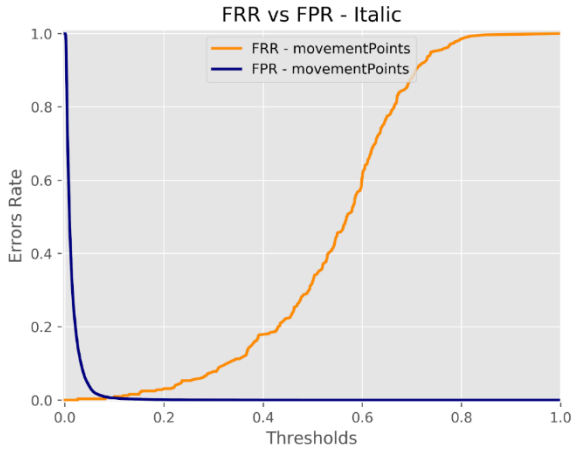
We can see how the weight influences the performances. The best weight seems to be 0.8. We can see that the behaviour is very similar to the one in the Identification case.



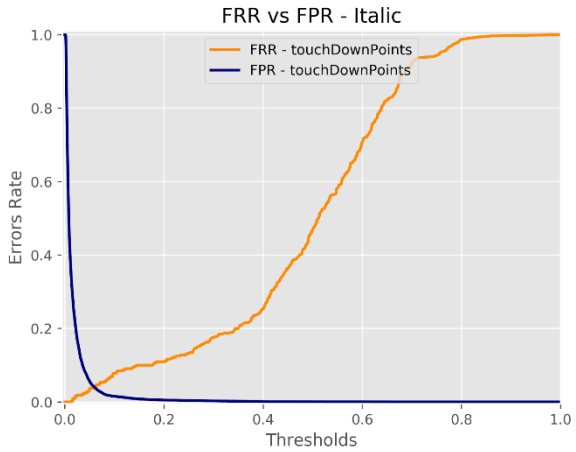
Even here the starting point is not important, the difference in performances is negligible.

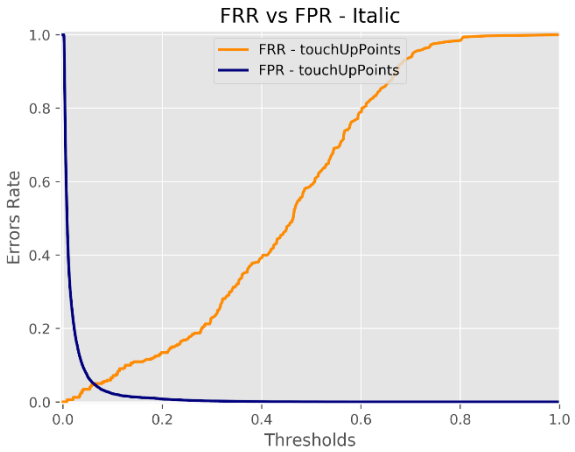


The overall ensemble SVMs do not perform bad:

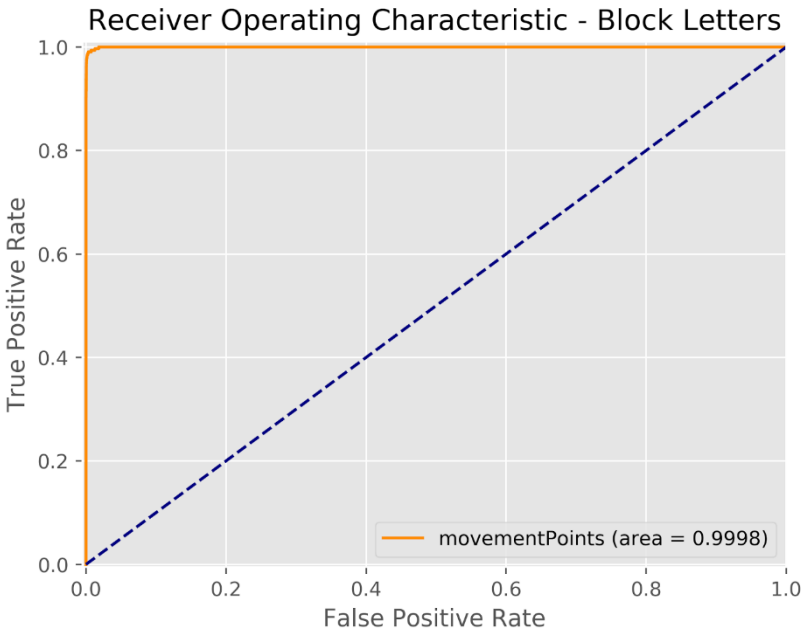


Looking at these curves is maybe even more clear that the SVM trained on the movement points is better that the ones trained on the touch down or touch up points.

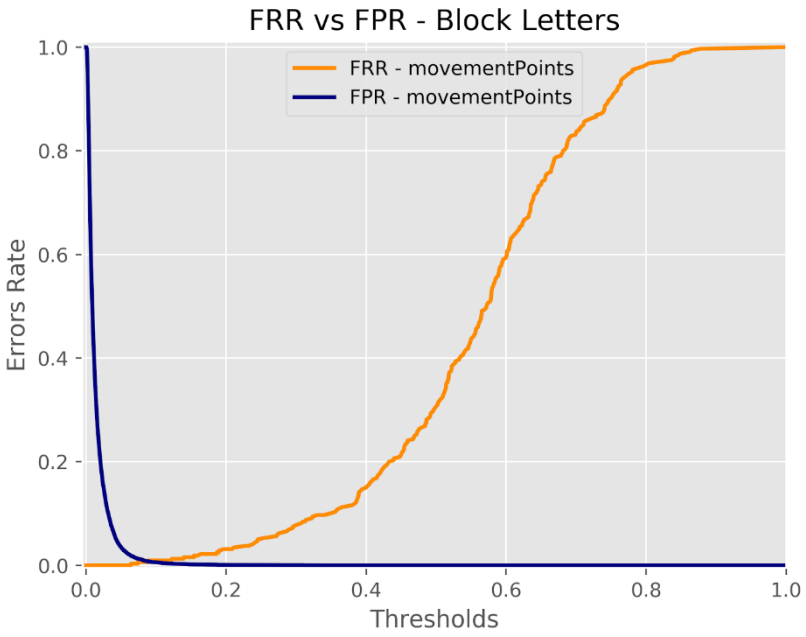


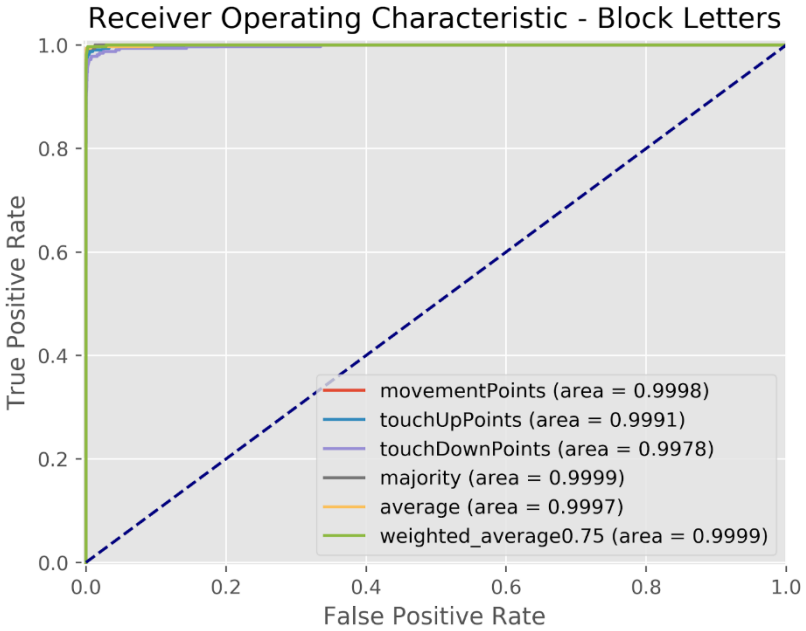


Block letters

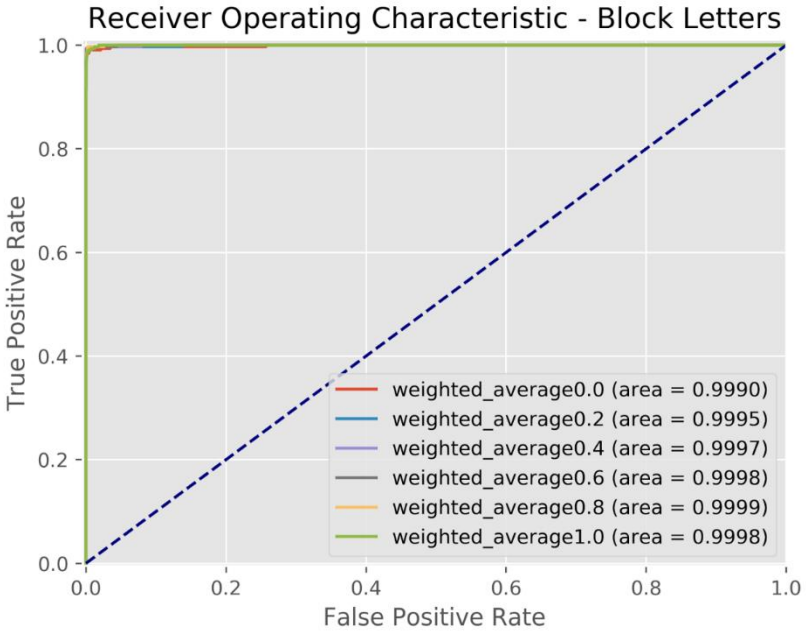


Even with the block letters we obtained very good results.

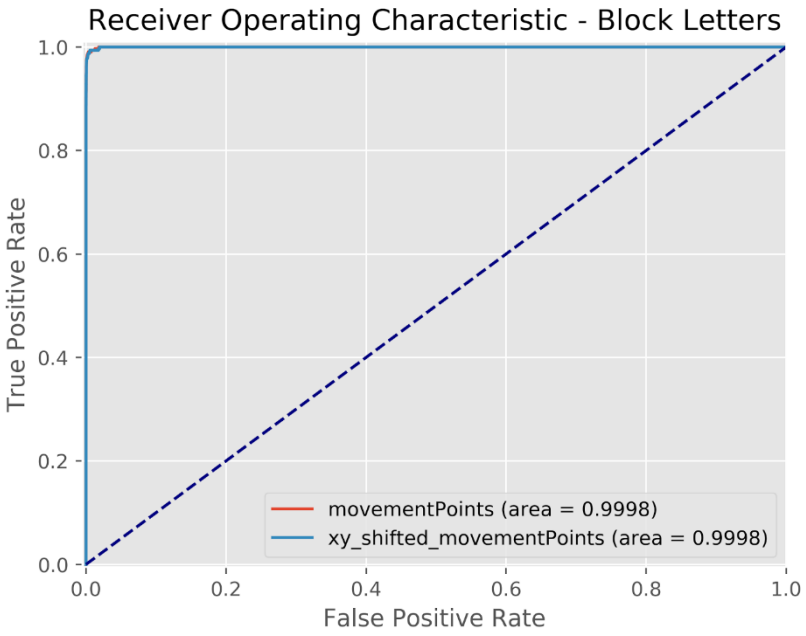




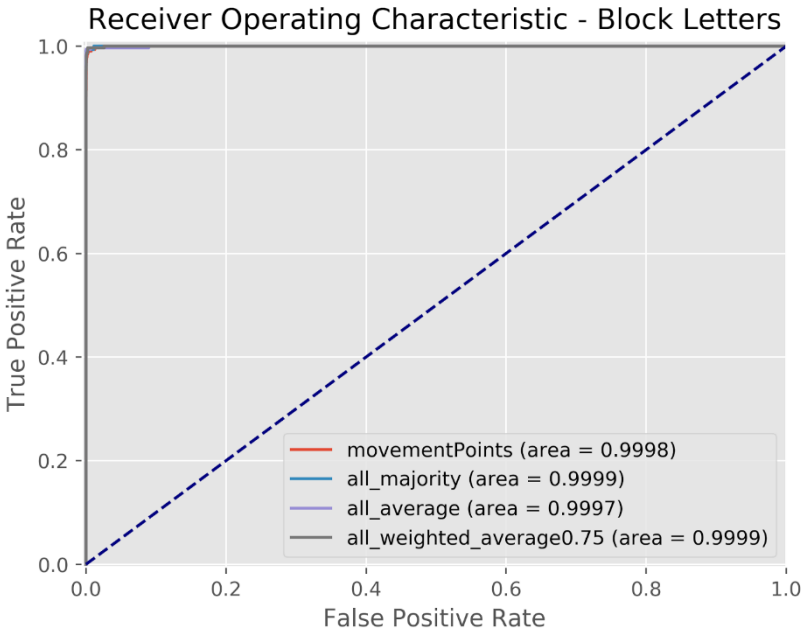
As in the identification, the SVMs on the touch down and touch up perform better than for the italic handwriting.



Thus, also the ensemble methods work better. In particular we can see, again, how the weight influences the performances.



Again, we can see that the starting point doesn't influence the performances.

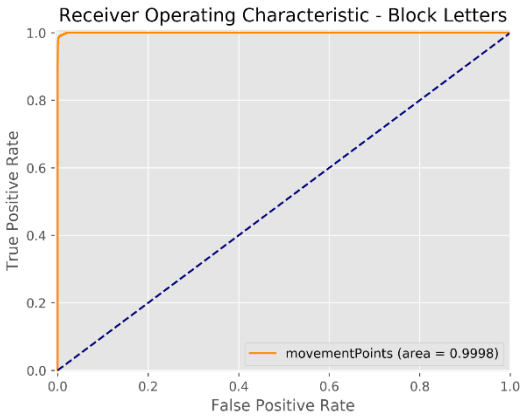


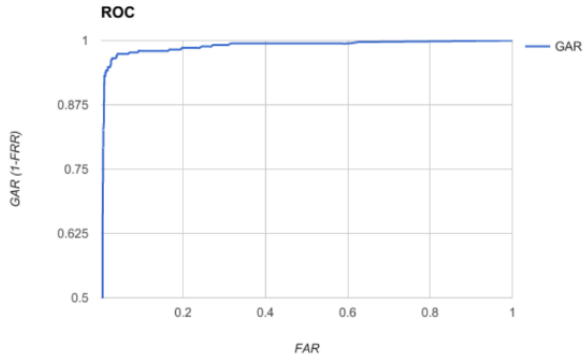
The overall ensembles perform quite well.

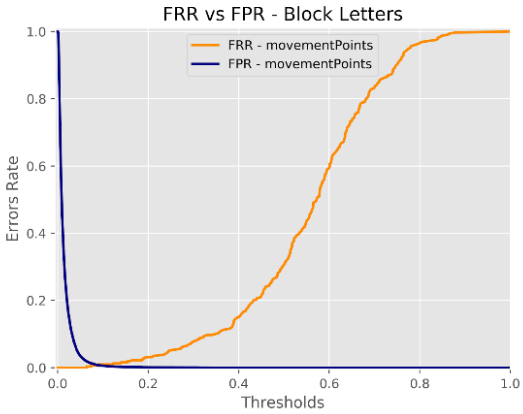
Comparison with Biopen

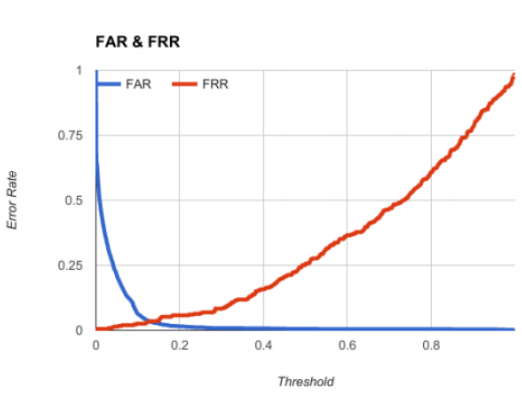
It has been said that this system was inspired by Biopen. We thought that it could be interesting to compare the results of the two systems. The available performance evaluation of Biopen concerns the verification mode, with the word “BIOSYS” written in block letters 32 times by 30 users.

The images to the left are relative to Biotouch, the images to the right are relative to Biopen.









Our system seems to perform better, probably because the problem that it must solve it’s easier:

- The elapsed time is relative to the beginning of the word, so there aren’t time shifting problems
- It has exact information about the connected components
- It has precise pixel-coordinates of the written points

Conclusions

The system seems to work very well both in identification and in verification. Performances decrease only when the time series are too short, and there isn't enough information: in this situation the quality of the system decreases. It is the case of touch up and down points in italic handwriting, because usually there are less connected components. Apart from this, the system behaves surprisingly well.

Obviously, the situation in which we are it is easier that a hypothetical application in the real word: there can be a far greater number of users, the attacks can be made trying to carefully falsify the pretended identity, and so on. Moreover, we tested the system only on words written consecutively.

However, these results are pretty encouraging for the hope that other systems, based on the same mechanism, could be developed. In the next session, the last one of this report, we'll expose some ideas about how this project could be continued.

Future works

Some ideas we have about what to do in the future to continue and improve the system are:

- Testing on dataset gathered after some time on the same users
- Testing on a bigger dataset to see if the performances are stable
- Testing on different approaches of SVM (OVO instead of the OVR)
- Testing different machine learning algorithms, that may give a better estimation of probabilities
- Testing boosting techniques, as Adaboost, that may perform well if there is some user particularly difficult to recognize
- Try to use a similar approach to detect age range or gender
- Explore a KNN DTW approach

Bibliografia

[1] «DeveloperAndroid web page,» [Online]. Available: https://developer.android.com/.

[2] P. J., «Probabilistic Outputs for Support Vector Machines and Comparisons to Regularized Likelihood Methods».

[3] F. Scozzafava e F. Ponzi, «BioPen: Dynamic Gesture-Tracking Signature Recognition System».

[4] M. Bicego, «Riconoscimento e recupero dell'informazione per bioinformatica, Classificazione : validazione».

[5] D. Griffiths e D. Griffiths, Head First Android Development (second edition).

[6] «pandas documentation,» [Online]. Available: https://pandas.pydata.org/pandas-docs/stable/.

[7] «scikit-learn documentation,» [Online]. Available: http://scikit-learn.org/stable/documentation.html.

[8] «tsfresh documentation,» [Online]. Available: https://tsfresh.readthedocs.io/en/latest/.

[9] K. Alexandre, Support Vector Machines succintly.

[10] «Matplotlib documentation,» [Online]. Available: https://matplotlib.org/.