**WEB AND SOCIAL INFORMATION EXTRACTION PROJECT Report**

lUCA mOSCHELLA, 1594551 AND Federica Spini, 1588482

Summary

[Introduction 2](#_Toc530928493)

[Technologies 2](#_Toc530928494)

[Dataset handling 2](#_Toc530928495)

[Modelling 3](#_Toc530928496)

[Acquisition 3](#_Toc530928497)

[Analytics 3](#_Toc530928498)

[Semantic processing 4](#_Toc530928499)

[Babelnet 5](#_Toc530928500)

[Babelfy 5](#_Toc530928501)

[Extraction of categories and domains 5](#_Toc530928502)

[WikiMID 5](#_Toc530928503)

[Tweet contents 5](#_Toc530928504)

[Items 6](#_Toc530928505)

[Limitations 6](#_Toc530928506)

[Twitter Information Extraction (task 4) 6](#_Toc530928507)

[What we downloaded with Twitter4j 6](#_Toc530928508)

[How we extracted the features 6](#_Toc530928509)

[Representation with latent categories 7](#_Toc530928510)

[Wikipedia pages 7](#_Toc530928511)

[Tweets 7](#_Toc530928512)

[Users (task 1) 7](#_Toc530928513)

[Why this formula? 8](#_Toc530928514)

[Curse of dimensionality 8](#_Toc530928515)

[Notes about the efficiency 9](#_Toc530928516)

[Adjacency matrix exponentiation 9](#_Toc530928517)

[SUPER CLUSTER TABLEs 9](#_Toc530928518)

[Tabella dei risultati della clusterizzazione al variare del numero dei cluster 11](#_Toc530928519)

[Continuous variables discretization 13](#_Toc530928520)

[Binarization 14](#_Toc530928521)

[Tuning parameters 15](#_Toc530928522)

[Decision trees 15](#_Toc530928523)

[Using a decision tree with class\_weight = {0:3, 1:7} and a preprocessing of all features except for capital gain and capital loss that have not been discretized: 16](#_Toc530928524)

[Using a decision tree with default class\_weight and a preprocessing of all features except for capital gain and capital loss that have not been discretized: 16](#_Toc530928525)

[Using a decision tree with class\_weight = {0:3, 1:7} and a complete preprocessing: 17](#_Toc530928526)

[Using a decision tree with default class\_weight and a complete preprocessing: 17](#_Toc530928527)

[Using a decision tree with class\_weight = {0:3, 1:7} and no discretization: 18](#_Toc530928528)

[Using a decision tree with default class\_weight and no discretization: 19](#_Toc530928529)

[SVM 19](#_Toc530928530)

[Change in class\_weight and preprocessing 20](#_Toc530928531)

[Change in C 21](#_Toc530928532)

[Change in gamma 22](#_Toc530928533)

[Random forest 22](#_Toc530928534)

[Change in max\_depth 22](#_Toc530928535)

[Change in n\_estimators 22](#_Toc530928536)

[Conclusions 23](#_Toc530928537)

# Introduction

DOBBIAMO FINIRLA DI SCRIVERE SUL GOOGLE DOC, POI COPIAMO E INCOLLIAMO

## Technologies

We decided to use different technologies depending on the task we had to solve.

* **Java**: to read the datasets, merge the information from the different sources, and generate a unique dataset
* **Python**: to perform algebraic computations on sparse data, clusterize and recommend items.
* **Twitter4j**: to download informations from Twitter
* **Babelnet**: to found the categories and the domains associated to a given Wikipedia page
* **Babelfy**: to disambiguate the text in the tweets
* **Scikit-learn**: to normalize, reduce the dimensionality, clusterize and evaluate the clusters
* **Scipy**/**numpy**: to represent sparse matrices and perform vectorized computations
* **fastutils**: provides primitive collections in Java, avoiding the auto boxing and un-boxing of primitve types.

We didn’t use Lucene since we don’t perform any text search.

# Dataset handling

The first problem that we had to face was how to represent the data in a way that it could be easily queried without being slow. We found that Lucene was not adequate, since its main goal is to index text to perform fast textual searches. In our approach we do not perform any textual search. We could have use a database such as MongoDB, but we found out that modelling the relationships directly in Java, using primitive collections (provided by the fastutils library, developed in the University of Milan), yielded good enough performance for our purposes, and provided a very straightforward access to the informations needed.

In the following sections, and in this report in general, we tried to abstract from the details of the implementation where possible, for this purpose there is the code documentation.

So, let’s shortly see how we represented wikimid, s21, s22 and s23. In other words, how we model our data.

## Modelling

The dataset model contains informations about users, Wikipedia pages, interests and tweets. This dataset will be expanded with the informations obtained from Babelnet, Babelfy and  the twitter API, and we will talk about how this has been done in the following sections.

INSERISCI IMMAGINE

A quick overview of the main objects and their purpose:

* **UserModel:** it represents Twitter users.This java model keeps trace of the known following and followers of users, of his tweets, and of the Wikipedia pages he likes. Moreover a given user could be a public figure and in this case we define him “famous”, and we keep track of the corresponding Wikipedia page that talks about him.
* **WikiPageModel:** it represents any Wikipedia page that is inherent to our Twitter content.
* **InterestModel:** it represents the interests of the Twitter users, so something that corresponds to a Wikipedia page, that we associate to each one of these object.
* **TweetModel:** it represents the tweets written by our users. To each tweet the author and the corresponding interest are associated.

All these objects are collected together into an object of the class **Dataset**.

This model has been made serializable, so that it could be saved in cache, without the need to process the original datasets each time.

## Acquisition

We’ve been given several dataset files. Three of them give information that are “independent” from the other ones. Indeed S21, S22 and S23 require operations ad hoc for them. On the other side we have four different files that compose the WikiMID dataset, that have to be integrated together.

We built a dataset model for each one of the given dataset and one that merges together all the datasets. The union of all the datasets will be the most useful model, since there is some intersection with the information available in the different datasets.

## Analytics

INSERISCI DESCRIZIONE DELLA TABELLA

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | | WIKIMID | S21 | S22 | S23 |
| USERS | Number of common users | 443346 |  |  |  |
| Number of famous users | 58789 |  |  |  |
| Total number of users | 502135 |  |  |  |
| TWEETS STATS | Total number of tweets | 1758685 |  |  |  |
| Greatest number of tweets | 13462 for 1 user |  |  |  |
| Smallest number of tweets | 0 for 214839 users |  |  |  |
| Median number of tweets | 1 for 161520 users |  |  |  |
| Mean of number of tweets | 3,5024 |  |  |  |
| Number of tweets per user variance | 1360,5557 |  |  |  |
| Number of tweets standard deviation | 36,8857 |  |  |  |
| FRIENDSHIP STATS | Total number of friendships | 66129104 |  |  |  |
| Greatest number of friendships | 106757 per 1 user |  |  |  |
| Smallest number of friendships | 0 per 39452 |  |  |  |
| Median number of friendships | 39 per 2703 users |  |  |  |
| Mean of number of friendships | 131,6959 |  |  |  |
| Number of friendships per user variance | 453893,3085 |  |  |  |
| Number of friendship standard deviation | 673,716 |  |  |  |
| FOLLOWING STATS | Total number of followings (incoming edges) | 33067306 |  |  |  |
| Greatest number of followings | 106757 per 1 user |  |  |  |
| Smallest number of followings | 0 per 443346 users |  |  |  |
| Median number of followings | 0 per 443346 users |  |  |  |
| Mean of number of followings | 65,8534 |  |  |  |
| Number of followings per user variance | 447614,1076 |  |  |  |
| Number of followings standard deviation | 669,0397 |  |  |  |
| FOLLOWER STATS | Total number of followers (outcoming edges) | 33067306 |  |  |  |
| Greatest number of followers | 2388 per 1 user |  |  |  |
| Smallest number of followers | 0 per 96853 |  |  |  |
| Median number of followers | 22 per 4190 users |  |  |  |
| Mean of number of followers | 65,8534 |  |  |  |
| Number of followers per user variance | 13887,9 |  |  |  |
| Number of followers standard deviation | 117,8469 |  |  |  |
| Total number of interests | | 282303 |  |  |  |
| Total number of Wikipedia pages | | 143257 |  |  |  |
| Total number of categories | | 169611 |  |  |  |
| Total number of domain | | 34 |  |  |  |

INSERISCI ESEMPIO DI CARATTERE NON UTF8 IN S2\*

# Semantic processing

Scrivi introduzione

## Babelnet

BabelNet is both a multilingual encyclopedic dictionary, with lexicographic and encyclopedic coverage of terms, and a semantic network which connects concepts and named entities in a very large network of semantic relations, made up of about 15 million entries, called Babel synsets. Each Babel synset represents a given meaning and contains all the synonyms which express that meaning in a range of different languages. inserisci reference

## Babelfy

Babelfy is a unified, multilingual, graph-based approach to Entity Linking and Word Sense Disambiguation based on a loose identification of candidate meanings coupled with a densest subgraph heuristic which selects high-coherence semantic interpretations.

A language-agnostic setting is available. In this setting, Babelfy takes into account all 271 languages without assuming or trying to infer the language of the input text giving the further possibility of annotating text written in multiple languages.

We think that the agnostic setting is the best possible approach in trying to disambiguate the tweets, since it is common writing tweets in multiple language (e.g. in english and in italian, depending on the context). inserisci reference

## Extraction of categories and domains

We used these semantic resources to extract categories and domains from the available informations. The number of domains is very small (34), the number of categories is very high (BabelNet obtains most of them through the processing of the Wikipedia’s categories). So, there is a great difference between categories and domains: the categories are very different, they can be more or less generic (for example a singer can be in the categories "[1940 births](http://en.wikipedia.org/wiki/Category:1940_births)“, “[English rock singers](http://en.wikipedia.org/wiki/Category:English_rock_singers)” and “[Murdered male actors](http://en.wikipedia.org/wiki/Category:Murdered_male_actors)”), while the domains are very few (every type of entertainment activity is simply included into “Media” domain).

### WikiMID

In the WikiMID dataset there are Wikipedia pages expressed as the interest in a tweet or as the personal page of a famous user. It turns out that it is possible to associate to each Wikipedia page a Synset. Moreover, we know that in BabelNet each Synset may be in one or more *domains* and in one or more *categories*.

So, we are able to obtain the categories, or domains, of each tweet and the categories, or domains, of the page of famous users.

These informations will be used to create an embedding of the users and the pages.

### Tweet contents

We have been given users IDs, and asked to download their information through the Twitter API. We downloaded some tweets for each user.

We have been able to disambiguate the downloaded tweets using Babelfy in AGNOSTIC mode, obtaining Synsets, whichever was the language of the user/tweet.

So, yet again, we’re able to obtain the categories, or domains, of each Tweet.

Actually, we didn’t disambiguate each tweet. It turns out that merging together in a single string the tweets of a given user (forcing each tweet in a new paragraph) gives more context to Babelfy, which is able to better disambiguate the words.

### Items

Obviously, we have been able to associate categories/domains even to the items in the dataset S22 and S23, since they are Wikipedia pages too.

### Limitations

This approach however has some negative aspects: not all the Wikipedia pages have an associated synset, and not all synsets have associated categories.

Moreover a problem with the encoding of non-ascii characters in the S22 and S23 datasets caused the loss of some information, since we haven’t been able to find the synset of some pages that actually have a synset in Babelnet (around 50 pages).

More insight on the data is available in the analytics section.

# Twitter Information Extraction (task 4)

One of our task (the fourth) was to use the Twitter API to download information for the Twitter users of the dataset S21. In this section we explain which type of information we have chosen to download, and how we have done it.

## What we downloaded with Twitter4j

The first thing we’ve found useful has been the *following relationships* of the users of S21. Twitter4j allows to download the complete list of the twitter ids of the **following** of a user. Almost all the users of S21 have some *follow-out* friends that already are in WikiMID, so they’re a precious information, so they will be easier to cluster. Moreover, often these users that already are in WikiMID represent famous people, so they can be directly associated with a personal page.

The other Twitter information we decided to download are the **tweets** of the users. Infact, thank to Babelnet API we are able to associate a tweet with a series of categories and this is precious to merge the S21’s users in clusters with the wikimid users. To do it we had to deal with two problems:

* Some of the users present in the dataset have been deleted from Twitter, their profile is private or they are suspended. So their information is not available. These exceptions has been captured, but we ignored the corresponding users.
* The time these operations require is long due to the limits of the free Twitter API. In particular the requests for a user’s following can be done every 15 minutes, so for 1500 users we need more than a day.

Downloading the tweets’ texts thanks to the Twitter API and using Babelfy on them, allowed us to obtained a set of synsets (INSERISCI REFERENCE DI BABELNET). Because of a synset is associated to a Wikipedia page, we extracted the categories from the corresponding pages as previously explained. These new categories are stored in the model of the author of the tweet and we’ll use them as tweets categories to assign some categories also to the users.

## How we extracted the features

The classes in our code that have to manage this part are the ones in the package **TwitterOperation**, with the extractor that makes the correct Twitter4j requests and the two type of responses, one for the friends of the users, one for the tweets. To have more details about the implementations see the documentation of the code.

# Representation with latent categories

We used BabelNet to extract the categories/domains, and we represented every object in the dataset in the space of the categories, merging together all the available informations. The way to do that is explained in the following sections. The number of dimension, that is the number of categories obtained from the data, is about 190’000.

We reduced the dimensionality of the space to 300, normalizing before and after the dimensionality reduction. In this way we didn’t deal with real categories, but with “latent” categories.

## Wikipedia pages

Each Wikipedia page is represented as a vector of numbers: one point in the space of the categories. Note that this vector will be very sparse, because a Wikipedia page has only a bunch of all the 190’000 categories.

## Tweets

In the WikiMID dataset each tweet is associated to only a wikipedia page. So each tweet can be represented as a point in the space of categories too.

Note that the tweets disambiguated through Babelfy don’t fall in this category, since they are merged together before the disambiguation.

## Users (task 1)

To make a correct representation of the users, we’ve decided to use Babelnet. Moreover as we have already said:

* How to associate categories to the Wikipedia pages
* How to associate categories to tweets

These two things are really important for the method we created to represent users as vectors of categories.

In this section we will see how we embedded all the informations available for a given user in a vector. Now the challenging part of the chapter comes.

The vector representation of a given user must embed the information about:

* The categories of the tweets that the user wrote
* The categories of the Wikipedia pages that the user likes
* The categories of the Wikipedia page that represents a user. As we said, some of our users are public figures and the categories of the pages that talk about them are important, but this is a different concept compared to the pages they like as Twitter users
* The categories of the users following: if a user admires and follows someone we expect that he’s interested also in his categories (it’s a recursive formulation)

Moreover, we thought that these various characteristics are all important, but not in the same way. Given a user, the Wikipedia pages he likes seemed to us a stronger evidence of his interests with respect to his tweets.

Another example is, for famous people, let’s say a singer, a personal page surely represents him and we want to take it into account. However, it contains the categories of the “public figure” of the singer. If the singer is also an amateur winemaker, even if nobody knows about it, he as a user may appreciate pages wine related and makes tweets about this topic. For this reason we gave different weights to the categories of different types (as for the categories of the personal pages which have less importance than the liked Wikipedia pages).

Now let’s see in detail how we embedded these information together.

We defined three matrices:

* Let T be the tweet matrix. It  contains a row for each user and a column for each existing category. A row contains the categories associated to all the tweets of a user. So  T\_i\_j=x means that the user i has x times the category j in his tweets.
* Let P be the personal page matrix. It  contains a row for each user and a column for each existing category. So  T\_i\_j=x means that the user i has x times the category j in his personal page. Note that each value can be only 0 or 1, because a user can have at most only one page. Moreover, consider the fact that most of the entries of this table are made of zeros, because only a little percentage of the users has a personal page, and each page doesn’t have a lot of categories.
* Let L be the liked item matrix. It  contains a row for each user and a column for each existing category. So  T\_i\_j=x means that the user i has x times the category j in the Wikipedia pages he likes.
* Let F be the friendship matrix, it is a boolean adjacency matrix. It  contains a row for each user and a column for each user too. A row represents the friendship relation, so that the user that corresponds to a row is a follower of the users corresponding to the columns set to one. So  T\_i\_j=x says if the user i follows or not the user j.

Now we can define M, a matrix that contains a row for each user and a column for each existing category, each i-th row will be the vector representation of the i-th user. A row represents the categories associated to all the various objects related to a user: so  T\_i\_j=x means that the characteristic j has importance x for the user i.

SUPERMAXIFORMULONA

Where:

* alpha is the weight (or the importance) given to the categories of the tweets
* beta is the weight (or the importance) given to the categories of the personal Wikipedia pages
* gamma is the weight (or the importance) given to the categories of the liked Wikipedia pages
* Capital delta iis the weight (or the importance) given to the categories of the following. Note that Capital delta depends on the friendship level. SPIEGARE LA COSA DELL’ESPONENTE: NON LO SCRIVO QUI PERCHE’ C’E’ BISOGNODI SCRIVERE LE FORMULE SPECIFICARE CHE: quelle dei following avranno alpha', beta' e gamma'

### Why this formula?

Spiegazione da scrivere ancora

## Curse of dimensionality

Until now, in describing how we’ve dealt with the problem of assign categories to the users, we’ve talked of matrices whose dimensions often depends on the categories number.

Once we obtained M, a 500k \* 190k matrix, we reduced its dimensionality to 500k \* m.

To reduce the dimensionality we defined  the following pipeline in sklearn:

* Normalize the rows such that the norm is one
* Reduce the dimensionality with TruncatedSVD (maintain only the m most important singular values)
* Normalize again the rows such that the norm is one

The definition of such pipeline, instead of applying manually each fit and trasformation, will help when dealing with test data. It will be enough to retrieve the already fit pipeline and just transform the new data.

ALGORITMO PER ABBASSARE LA DIMENSIONALITA’ DEI DATI, Spiega come funziona TruncatedSVD …? non so se serve, forse un copia incolla dell’abstract della documentazione di sklearn

## Notes about the efficiency

We are dealing with big matrices. The adjacency matrix has dimension 500k\*500k, even if it’s a boolean matrix, storing it entirely in memory would mean to require  58 GB of RAM (assuming no overhead). The situation is even worse when we consider the other matrices that have dimension 500k\*190k, each one of them would require 353 GB of RAM (assuming no overhead) since they contain float32 and not booleans.

It’s obvious that it is not feasible to store in memory those dense matrices. Fortunately we can exploit one common property: they are sparse matrices.

We used scipy.sparse to work with these matrices, and we were able to fit them, and the computations, in a 32GB machine.

However, using this approach lead to some limitations: not all the functions made available by sklearn accept a sparse matrix, and not all the functions scale well enough memory wise. So our choices were limited by the available options. One of the main motivations that lead us to continue the project in python and not in java, was that java hasn’t a good enough, easy to use, library to handle sparse matrices, algebraic computations and machine learning tools. Instead in python we could leverage scipy, numpy and sklearn that, together, met all our needs.

### Adjacency matrix exponentiation

Must be noted, as already explained, that performing the i-power of the adjacency matrix gives us the nodes at distance i. Given the structure of the graph that one could expect from twitter users (some people have a lot of edges, following the Zipf law) it is straightforward to see that the initially sparse adjacency matrix would become exponentially less sparse at each exponentiation. On our 32GB machine we were able to compute only the square of the adjacency matrix. Probably the operations could be optimized in some way, but we think that the importance of informations decreases exponentially as it travels from friend to friend, and we enforced this behaviour setting an exponential decay in the formula. So, what we computed should give us a good enough approximation of an optimal computation (where higher powers of the adjacency matrix are computed).

## SUPER CLUSTER TABLEs

Configurazioni fatte:

* 0a6f03a1c12693418641bed9efafbdcea3dd3644 (in corso)

Configuration values:

|  |  |  |
| --- | --- | --- |
| FEATURE NAME | VALUE | DESCRIPTION |
| Dataset | The union of WikiMID, S21 and S22 | The dataset from which we have extracted information for cluster |
| Dimension | Complete | If this value is "complete" we've used all the data from the dataset. If it is "small" we've limited the read values, to increase the speed of the clusterization |
| Cluster\_over | Categories | The clusterization can be based on categories values or on domain value |
| Max\_user\_distance | 2 | The neighbour of at most distance=2, considering only outcoming edges, affect the clusterization of the users |
| Tweet\_importance | 0.3 | The weight of the user's tweets in clustering him |
| Personal\_page\_importance | 0.15 | The weight of the user's personal Wikipedia page in clustering him |
| Liked\_item\_importance | 0.55 | The weight of the Wikipedia pages liked by the user in clustering him |
| Rate\_of\_decay | 0.5 | The weight of the user's following at , where at distance I the weight is |
| Follow\_out\_tweet\_importance | 0.15 | The weight of the user's followings tweets in clustering him |
| Follow\_out\_personal\_page\_importance | 0.55 | The weight of the user's followings personal Wikipedia page in clustering him |
| Follow\_out\_liked\_items\_importance | 0.3 | The weight of the user's followings liked Wikipedia pages in clustering him |
| Reducer | Truncated\_svd | The type of reducer that is used |
| Matrix\_dimensionality | 300 | The dimensionality of the characteristics representation. |
| Clusterer | Minibatch\_kmeans | The cluster strategy used |
| N\_clusters | **300** | **The number of clusters that are created** |
| Max\_iter | 1000000 | The maximum number of iterations |
| Batch\_size | 5000 | The amount of data that are processed each iteration |
| Max\_no\_improvement | 10000 | The maximum number of iteration that can be executed without an improvement |
| Init\_size | 50000 | The number of the initial group of user that are clustered |
| N\_init | 100 | The maximum number of users combination that are considered as starting centroids |
| Reassignment\_ratio | 1e-06 | The ratio of the allowed deviation from the centroids of the previous centroids |

## Tabella dei risultati della clusterizzazione al variare del numero dei cluster

|  |  |  |
| --- | --- | --- |
| NUMBER OF CLUSTERS | CALINSKY | DAVIES |
| Baseline | 636.842581 | 3.167859 |
| 50 | 7070.23259 | 2.754809 |
| 100 | 4624.01507 | 2.565989 |
| 150 | 3657.44981 | 2.439572 |
| 300 | 2505.07177 | 2.177000 |
| 350 | 2284.19245 | 2.246862 |
| 400 | 2133.40002 | 2.191238 |
| 450 | 1968.63127 | 2.220589 |
| 500 | 1839.38744 | 2.181618 |
| 550 | 1734.50881 | 2.193601 |
| 600 | 1633.38348 | 2.203257 |
| 750 | 1403.98572 | 2.229943 |
| 800 | 1339.58456 | 2.245939 |

This is the table of the results of three methods, with the report that it is possible to print with the function classification report of scikit. It is only an initial simple look at a first approach quality

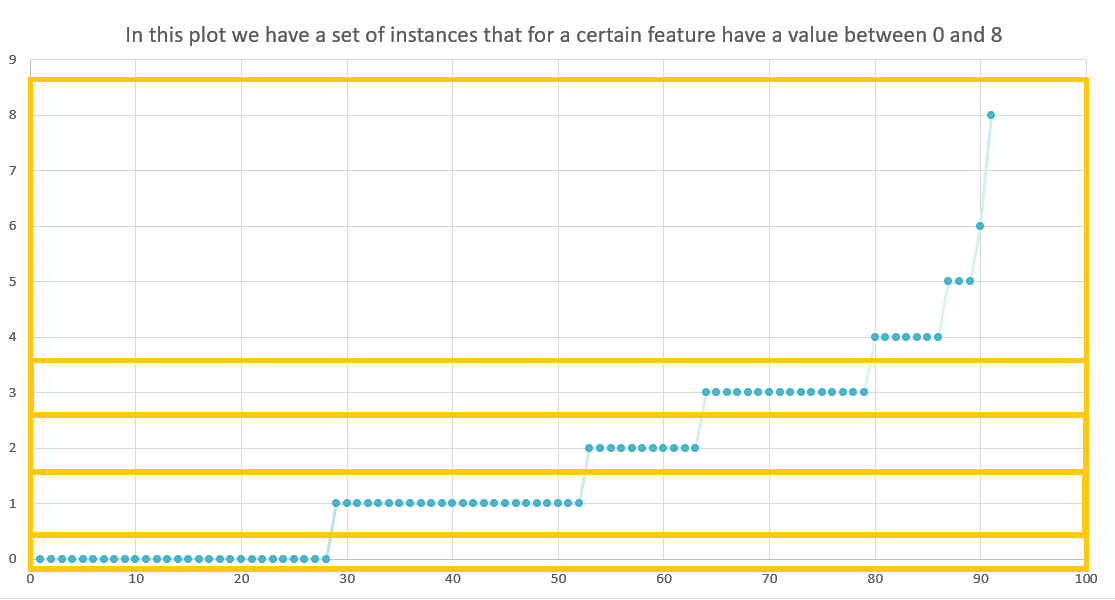
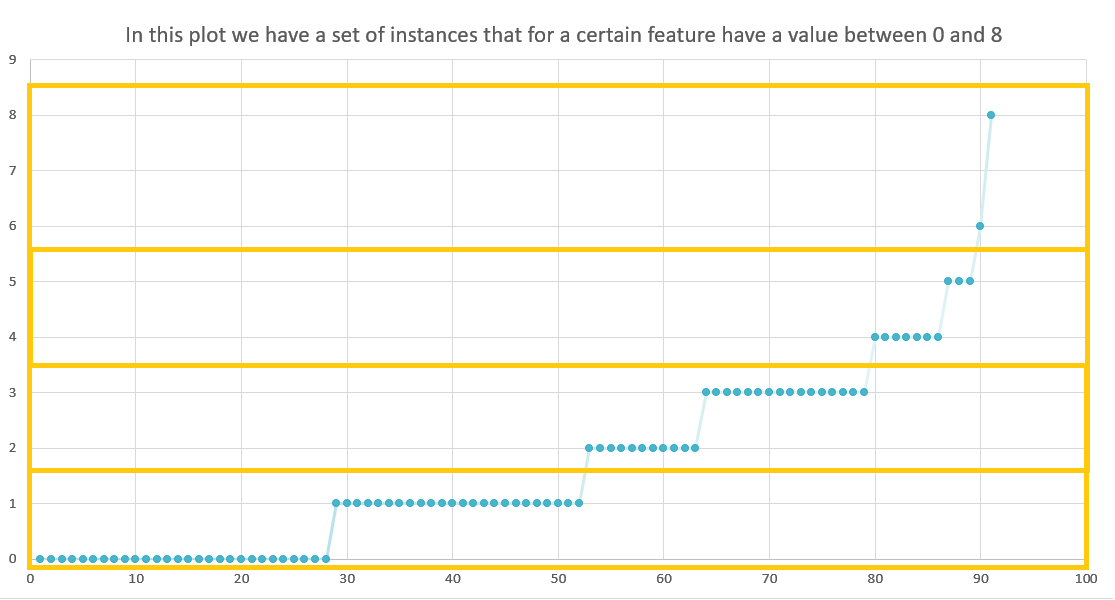
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | decision\_tree precision | decision\_tree recall | decision\_tree f\_score | decision\_tree support |
| <=50K | 0,88 | 0,88 | 0,88 | 12435 |
| >50K | 0,61 | 0,62 | 0,61 | 3846 |
| avg/total | 0,82 | 0,81 | 0,82 | 16281 |
|  | **svm precision** | **svm recall** | **svm f\_score** | **svm support** |
| <=50K | 0,76 | 1 | 0,87 | 12453 |
| >50K | 0,57 | 0,01 | 0,01 | 3846 |
| avg/total | 0,72 | 0,76 | 0,66 | 16281 |
|  | **random\_forest precision** | **random\_forest recall** | **random\_forest f\_score** | **random\_forest support** |
| <=50K | 0,88 | 0,88 | 0,88 | 12453 |
| >50K | 0,61 | 0,62 | 0,61 | 3846 |
| avg/total | 0,82 | 0,82 | 0,82 | 16281 |

Note that a model that decides randomly guesses the correct result with a probability of 50%, while a model that always says than the classification value is "<=50K" on the considered data, guesses about the 75% of the data. So, while the decision tree and the random forest already get results that are in a certain way "better" than a trivial approach, the SVM maybe needs a more sophisticated preprocessing or other parameters. Considering this as a starting point, I've continued my project.

## Continuous variables discretization

Another problem that can be solved in the preprocessing phase is that keeping variables with continuous variables in a very sparse range, it's not easy to cluster the various instances. In particular there are 4 values that cause that:

* **Age**: in the training data the minimum value is 17 and the maximum is 90
* **Fnlwgt**: in the training data the minimum value is 12285 and the maximum is 1484705
* **Capital-gain**: in the training data the minimum value is 0 and the maximum is 99999
* **Capital-loss**: in the training data the minimum value is 0 and the maximum is 3770
* **Hours of work**: in the training data the minimum value is 1 and the maximum is 99

So, I've written a method to discretize a variable in the python file **data\_preprocessor** that is called **balanced\_discretize\_feature**. Its purpose is to limit the possible values of a feature to a certain number that is a parameter, that is called discretization\_factor. Its default value is 10, simply because it seemed to me a good choice, but later I will show how the results can change with this value. So, the point is: how could I divide a continuous range of values into a set of only ten values? At first, I thought to simply split the maximum value of a feature into the discretization factor value, but this should not be correct in my opinion: obtaining a set of value bands in this way is a risk because this can lead to poorly balanced value. So, I preferred divided the value bands, making sure that they're equally distributed across the training data. For example, in the first image of the previous page, the band 6-8 corresponds to only 2 values, while the band 0-1 to more than the 50% of all the values. A more balanced way to split the values it's the one I used: the bands are created in spite of contain about the same number of values, in a way similar to the second image. It quite obvious, but however important to say, that if a value higher than any training data the model have seen is found in the predict phase, it is preprocessed and put in the last band.

Has that been a good choice? It depends on the discretized feature. In a first moment I've tested the function and trained the three models, always with the default parameter, to analyse what would happen discretising separately each of the four features mentioned before. The result is that discretizing the age the performances improve, in particular for the SVMs. The same happen for the fnlwgt value and the number of hours of job. Combining their discretization, those are the best results I obtained in this phase, only with default parameters:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | decision\_tree precision | decision\_tree recall | decision\_tree f\_score | decision\_tree support |
| <=50K | 0,88 | 0,88 | 0,88 | 12435 |
| >50K | 0,61 | 0,61 | 0,61 | 3846 |
| avg/total | 0,82 | 0,82 | 0,82 | 16281 |
|  | **svm precision** | **svm recall** | **svm f\_score** | **svm support** |
| <=50K | 0,87 | 0.95 | 0,91 | 12453 |
| >50K | 0,77 | 0,54 | 0,64 | 3846 |
| avg/total | 0,85 | 0,85 | 0,84 | 16281 |
|  | **random\_forest precision** | **random\_forest recall** | **random\_forest f\_score** | **random\_forest support** |
| <=50K | 0,84 | 0,98 | 0,91 | 12453 |
| >50K | 0,84 | 0,41 | 0,56 | 3846 |
| avg/total | 0,84 | 0,84 | 0,82 | 16281 |

At this point it seems clear that with the good preprocessing techniques the method with the greater chance of improve is the SVM, that at first seemed the worst method. This probably happen because data were too sparse.

It is important to notice that in these trainings I didn't discretize the capital-gain and the capital-loss values. This is because discretizing them decreases the models' performances. But why? This is because of the distribution of their values, in particular:

* For the value capital-gain the 91,25% of the training instances have value 0, the others have high values sparse in a very wide range.
* For the value capital-loss the 95,35% of the training instances have value 0, the others have high values sparse in a very wide range.

Try to split those two values in bands is counterproductive, so I've binarized their values as explained in the next paragraph.

## Binarization

As I have said in the previous paragraph, there are two feature, capital-gain and capital-loss, that can't be discretized: almost all values are zeros, and the ones that are not 0 are too sparse (about 2000 values that are between 100 and 1000000). For this reason, I decided to binarize them, but not with the built-in methods of scikit. The scikit binarization creates a new feature for each value that appear in the data and set, for each element, all zeros and 1 only in the correct value. This actually is a one hot encoder of the values. What I want is not to create new feature increasing the dimensionality of the data. Looking at the percentage of the zero values in the two feature I wrote a function that binarizes the value, setting the value to zero if it already is so, and setting it to one in any other case.

These are the result training the three models always with sci-kit default parameter:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | decision\_tree precision | decision\_tree recall | decision\_tree f\_score | decision\_tree support |
| <=50K | 0,86 | 0,87 | 0,86 | 12435 |
| >50K | 0,56 | 0,55 | 0,55 | 3846 |
| avg/total | 0,79 | 0,79 | 0,79 | 16281 |
|  | **svm precision** | **svm recall** | **svm f\_score** | **svm support** |
| <=50K | 0,86 | 0.93 | 0,90 | 12453 |
| >50K | 0,71 | 0,52 | 0,60 | 3846 |
| avg/total | 0,83 | 0,84 | 0,83 | 16281 |
|  | **random\_forest precision** | **random\_forest recall** | **random\_forest f\_score** | **random\_forest support** |
| <=50K | 0,85 | 0,96 | 0,90 | 12453 |
| >50K | 0,76 | 0,45 | 0,57 | 3846 |
| avg/total | 0,83 | 0,84 | 0,82 | 16281 |

The quality of the result is worst than without the binarization and this had been an unexpected problem. Thinking about which could be the cause I think that it happened because it is true that there are few instances that don't have the value 0 in capital-gain and capital-loss feature, but the variety of values that can be there are in a very wide range. So, I've set the band of the division in this way: [0, 50, 100, 500, 1000, 5000, 10000, 50000, 100000] and in a certain way this can be considered a more classical discretization style. Now, if the initial value of an instance in capital gain is 4789, it becomes 4 because is more than 1000 and less than 5000 in this discretization.

Those are the results of the three type of models run with default parameter and this type of preprocessing:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | decision\_tree precision | decision\_tree recall | decision\_tree f\_score | decision\_tree support |
| <=50K | 0,85 | 0,87 | 0,86 | 12435 |
| >50K | 0,55 | 0,49 | 0,52 | 3846 |
| avg/total | 0,78 | 0,78 | 0,78 | 16281 |
|  | **svm precision** | **svm recall** | **svm f\_score** | **svm support** |
| <=50K | 0,83 | 0.95 | 0,89 | 12453 |
| >50K | 0,70 | 0,37 | 0,49 | 3846 |
| avg/total | 0,80 | 0,81 | 0,79 | 16281 |
|  | **random\_forest precision** | **random\_forest recall** | **random\_forest f\_score** | **random\_forest support** |
| <=50K | 0,82 | 0,97 | 0,89 | 12453 |
| >50K | 0,78 | 0,29 | 0,42 | 3846 |
| avg/total | 0,81 | 0,81 | 0,78 | 16281 |

Once again, our results have become worse. Anyhow the binary preprocessing seems to be better, so in the next phase, when it will be done a comparison of the results obtained with different parameter, I'll use both discretizes and not discretized capital-gain and capital-loss value. It is a not elegant choice to not preprocess a value, but if the results are better and it doesn't lead to particularly challenging requirements of processing time, I think it's the right choice.

# Tuning parameters

To understand which could be the better classificatory for my data I've tuned some of the parameter of scikit. Indeed, maybe a method that doesn't seem very effective could reveal to be perfect if appropriately set. In this process I didn't tuned all the parameters with all the possible values: it is a very long operation and it is not necessary to have an idea of which can be the correct way to train a model. I changed the parameter that seemed more interesting to approach the problem and that logically can lead to a change of the results. In this chapter, unlike what has been done so far, I'll talk about the three methods of decision trees, SVMs and random forest severally, to explain my choices for each one.

## Decision trees

In scikit learn there are different parameters that can be set in decision trees. I choose not to try to change the default value of most of them: with those type of data, that have only 14 features, they're perfect.

The parameter that I really found interesting are 2: the first one is **class\_weight** that allows to specify if the classes values are not balanced. This happens in the dataset, because the first value is commoner than the second one in a ratio of about 70% against 30%. So, I've tried to change this value giving more weight to the second value. It hasn't led to a great improvement: both with a complete and a partial preprocessing only the recall value results about the 1% with a large amount of tests. The second parameter is **max\_depth** that allows to specify the maximum depth of the decision tree that the function returns out. This one is an important feature to avoid overfitting on data. So, I tried to tune it with the following values: 5, 10, 15, 30, 60.

The results are in the following tables, where I've deleted the support because it doesn't change and the max\_depth = 60 because it turns out results that are little worse than max\_depth=30. For each type of preprocessing parameter it's highlighted the better result in green.

### Using a decision tree with class\_weight = {0:3, 1:7} and a preprocessing of all features except for capital gain and capital loss that have not been discretized:

|  |  |  |  |
| --- | --- | --- | --- |
|  | max\_depth=5  precision | max\_depth=5  recall | max\_depth=5  f\_score |
| <=50K | 0,93 | 0,78 | 0,85 |
| >50K | 0,53 | 0,80 | 0,64 |
| avg | 0,83 | 0,78 | 0,80 |
|  | **max\_depth=10**  **precision** | **max\_depth=10**  **recall** | **max\_depth=10**  **f\_score** |
| <=50K | 0,91 | 0.81 | 0,86 |
| >50K | 0,55 | 0,75 | 0,64 |
| avg | 0,83 | 0,80 | 0,81 |
|  | **max\_depth=15**  **precision** | **max\_depth=15**  **recall** | **max\_depth=15**  **f\_score** |
| <=50K | 0,91 | 0,81 | 0,85 |
| >50K | 0,54 | 0,74 | 0,62 |
| avg | 0,82 | 0,79 | 0,80 |
|  | **max\_depth=30**  **precision** | **max\_depth=30**  **recall** | **max\_depth=30**  **f\_score** |
| <=50K | 0,85 | 0,86 | 0,86 |
| >50K | 0,54 | 0,52 | 0,53 |
| avg | 0,78 | 0,78 | 0,78 |

### Using a decision tree with default class\_weight and a preprocessing of all features except for capital gain and capital loss that have not been discretized:

|  |  |  |  |
| --- | --- | --- | --- |
|  | max\_depth=5  precision | max\_depth=5  recall | max\_depth=5  f\_score |
| <=50K | 0,83 | 0,96 | 0,89 |
| >50K | 0,74 | 0,38 | 0,50 |
| avg | 0,81 | 0,82 | 0,80 |
|  | **max\_depth=10**  **precision** | **max\_depth=10**  **recall** | **max\_depth=10**  **f\_score** |
| <=50K | 0,86 | 0.93 | 0,89 |
| >50K | 0,70 | 0,51 | 0,59 |
| avg | 0,82 | 0,83 | 0,82 |
|  | **max\_depth=15**  **precision** | **max\_depth=15**  **recall** | **max\_depth=15**  **f\_score** |
| <=50K | 0,86 | 0,91 | 0,88 |
| >50K | 0,63 | 0,50 | 0,56 |
| avg | 0,80 | 0,81 | 0,81 |
|  | **max\_depth=30**  **precision** | **max\_depth=30**  **recall** | **max\_depth=30**  **f\_score** |
| <=50K | 0,85 | 0,87 | 0,86 |
| >50K | 0,55 | 0,50 | 0,52 |
| avg | 0,78 | 0,78 | 0,78 |

### Using a decision tree with class\_weight = {0:3, 1:7} and a complete preprocessing:

|  |  |  |  |
| --- | --- | --- | --- |
|  | max\_depth=5  precision | max\_depth=5  recall | max\_depth=5  f\_score |
| <=50K | 0,94 | 0,77 | 0,85 |
| >50K | 0,53 | 0,85 | 0,66 |
| avg | 0,85 | 0,79 | 0,80 |
|  | **max\_depth=10**  **precision** | **max\_depth=10**  **recall** | **max\_depth=10**  **f\_score** |
| <=50K | 0,93 | 0,82 | 0,87 |
| >50K | 0,58 | 0,81 | 0,68 |
| avg | 0,85 | 0,82 | 0,83 |
|  | **max\_depth=15**  **precision** | **max\_depth=15**  **recall** | **max\_depth=15**  **f\_score** |
| <=50K | 0,93 | 0,81 | 0,87 |
| >50K | 0,57 | 0,81 | 0,67 |
| avg | 0,85 | 0,81 | 0,82 |
|  | **max\_depth=30**  **precision** | **max\_depth=30**  **recall** | **max\_depth=30**  **f\_score** |
| <=50K | 0,89 | 0,87 | 0,88 |
| >50K | 0,60 | 0,64 | 0,62 |
| avg | 0,82 | 0,81 | 0,82 |

### Using a decision tree with default class\_weight and a complete preprocessing:

|  |  |  |  |
| --- | --- | --- | --- |
|  | max\_depth=5  precision | max\_depth=5  recall | max\_depth=5  f\_score |
| <=50K | 0,87 | 0,96 | 0,91 |
| >50K | 0,78 | 0,52 | 0,62 |
| avg | 0,85 | 0,85 | 0,84 |
|  | **max\_depth=10**  **precision** | **max\_depth=10**  **recall** | **max\_depth=10**  **f\_score** |
| <=50K | 0,88 | 0,94 | 0,91 |
| >50K | 0,76 | 0,60 | 0,67 |
| avg | 0,85 | 0,86 | 0,85 |
|  | **max\_depth=15**  **precision** | **max\_depth=15**  **recall** | **max\_depth=15**  **f\_score** |
| <=50K | 0,88 | 0,92 | 0,90 |
| >50K | 0,71 | 0,61 | 0,65 |
| avg | 0,84 | 0,85 | 0,84 |
|  | **max\_depth=30**  **precision** | **max\_depth=30**  **recall** | **max\_depth=30**  **f\_score** |
| <=50K | 0,89 | 0,87 | 0,88 |
| >50K | 0,60 | 0,64 | 0,62 |
| avg | 0,82 | 0,81 | 0,82 |

### Using a decision tree with class\_weight = {0:3, 1:7} and no discretization:

|  |  |  |  |
| --- | --- | --- | --- |
|  | max\_depth=5  precision | max\_depth=5  recall | max\_depth=5  f\_score |
| <=50K | 0,94 | 0,78 | 0,85 |
| >50K | 0,54 | 0,83 | 0,66 |
| avg | 0,84 | 0,79 | 0,81 |
|  | **max\_depth=10**  **precision** | **max\_depth=10**  **recall** | **max\_depth=10**  **f\_score** |
| <=50K | 0,94 | 0,80 | 0,87 |
| >50K | 0,57 | 0,84 | 0,68 |
| avg | 0,85 | 0,81 | 0,82 |
|  | **max\_depth=15**  **precision** | **max\_depth=15**  **recall** | **max\_depth=15**  **f\_score** |
| <=50K | 0,93 | 0,81 | 0,87 |
| >50K | 0,57 | 0,80 | 0,67 |
| avg | 0,84 | 0,81 | 0,82 |
|  | **max\_depth=30**  **precision** | **max\_depth=30**  **recall** | **max\_depth=30**  **f\_score** |
| <=50K | 0,89 | 0,87 | 0,88 |
| >50K | 0,60 | 0,64 | 0,62 |
| avg | 0,82 | 0,81 | 0,82 |

### Using a decision tree with default class\_weight and no discretization:

|  |  |  |  |
| --- | --- | --- | --- |
|  | max\_depth=5  precision | max\_depth=5  recall | max\_depth=5  f\_score |
| <=50K | 0,87 | 0,95 | 0,91 |
| >50K | 0,78 | 0,52 | 0,63 |
| avg | 0,85 | 0,85 | 0,84 |
|  | **max\_depth=10**  **precision** | **max\_depth=10**  **recall** | **max\_depth=10**  **f\_score** |
| <=50K | 0,88 | 0,94 | 0,91 |
| >50K | 0,75 | 0,59 | 0,66 |
| avg | 0,85 | 0,86 | 0,85 |
|  | **max\_depth=15**  **precision** | **max\_depth=15**  **recall** | **max\_depth=15**  **f\_score** |
| <=50K | 0,88 | 0,92 | 0,90 |
| >50K | 0,70 | 0,60 | 0,65 |
| avg | 0,84 | 0,84 | 0,84 |
|  | **max\_depth=30**  **precision** | **max\_depth=30**  **recall** | **max\_depth=30**  **f\_score** |
| <=50K | 0,88 | 0,88 | 0,88 |
| >50K | 0,61 | 0,62 | 0,62 |
| avg | 0,82 | 0,82 | 0,82 |

It is interesting to see how the different setting of the decision trees have the same reaction at the variation of the max\_depth parameter. The better value among the ones I've tried surely is 10. Probably the more this value gets higher than 10 the more is the overfitting on the training data. The better results are obtained without any type of preprocessing or with all the preprocessing techniques I've talked about in the previous chapter. It is not a surprise that the decision trees work well without preprocessing on these types of data, because most of the feature have a finite domain. But We've to think that this could lead to very heavy operations if the data quantity grows. Think that this is a sampling of the population taken in just one year. What would happen if we took data from an entire city big as Roma (so about 5 million of people) for something like 10 years? The resulting tree of 50 million instances without any preprocessing could be enormous and too specific. So, the better decision tree I've built, is the one with a complete preprocessing, with max\_depth=10, that improves the baseline of the 10%.

## SVM

As for the decision trees make a tuning of all the parameters that scikit allows to customize is a too long operation, so I've decided to modify and combine only some of them. These are the parameters on which I've tried some variation:

* **C**: is the parameter that defines how much an error on the training data costs: the less is C the less is the overfitting, but maybe the system could make more errors
* **Gamma**: default is 1/n\_featues.
* **Kernel**: specifies if the kernel function is linear, poly, a sigmoid or rbf (the default value)
* **Degree**: if the kernel is 'poly' specifies the degree of the function that splits the values
* **Class\_weight**: the same that for the decision trees
* **Max\_iter**: the maximum number of iterations in training the SVM

Putting here all the tables with all the combination is quite dispersive (they would be about 400 tables) , so only the more interesting results will be in this report.

### Change in class\_weight and preprocessing

As for the decision tree is important to understand what type o preprocessing is better for the SVMs. I've tried to rerun them without any preprocessing, with all the features preprocess except capital-gain and capital-loss and all the features preprocessed. Then I've tried to understand if the SVMs are better with or without the specification of the class\_weight. In particular I've considered the weight {first\_class\_value:3, second\_clas\_value:7}. The using of class\_weight doesn't change the resulting values, but the preprocessing makes an enormous difference:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Without preprocessing  precision | Without preprocessing  Recall | Without preprocessing  f\_score | Without preprocessing  support |
| <=50K | 0,76 | 1,00 | 0,87 | 12435 |
| >50K | 0,57 | 0,01 | 0,01 | 3846 |
| avg/total | 0,72 | 0,76 | 0,66 | 16281 |
|  | **Partial preprocessing**  **precision** | **Partial preprocessing**  **recall** | **Partial preprocessing**  **f\_score** | **Partial preprocessing**  **support** |
| <=50K | 0,87 | 0.95 | 0,91 | 12453 |
| >50K | 0,77 | 0,54 | 0,64 | 3846 |
| avg/total | 0,85 | 0,85 | 0,84 | 16281 |
|  | **Complete preprocessing precision** | **Complete preprocessing recall** | **Complete preprocessing f\_score** | **Complete preprocessing support** |
| <=50K | 0,83 | 0.95 | 0,89 | 12453 |
| >50K | 0,70 | 0,37 | 0,49 | 3846 |
| avg/total | 0,80 | 0,81 | 0,79 | 16281 |

It is clear that for the SVMs the better choice is to make a partial preprocessing, that allows them to improve the performance about of the 8-10%.

### Change in C

For each type of kernel, I've tried a c value among 0.001, 0.01, 0.1, 1(default), 10 and 100. Those are the tables that show how the value changes when C changes:

From those graphs maybe, it is clear that the better kernel is the 'rbf'. This is not completely true, because the training of 'poly' and 'linear' SVM employs a very large amount of time. For this reason, I stopped the training after one million of iteration and probably it is the cause of their terrible results. So, these results are not "so real": with my computer the 'linear' kernel employs about 5 minutes with one million of iteration obtaining the showed results. Always the linear kernel, with a c value of 0,1 employs 25 minutes to train without any bound of time or of number of iterations. Tre result is that the average precision is 0,84 instead of 0,38 , the average recall is 0,84 instead of 0,23 , and the average of f-measure value is 0,83 instead of 0,14. For the 'poly' kernel I've trained it without a bound with the minimum degree (set at 2, because if it would be 1 it is linear) for more than two hours and it didn't converge, so probably it is completely unsuitable to the considered data.

Apart from unreliable results of 'poly' and 'linear' kernels, the important matter here is making a couple of consideration about the value C: in the 'rbf' kernel the better one is the default (c=1) that is a "middle way" c-value. For the sigmoid it seems better to use a little value, and this probably happen because with a large c the model wants "very tight" bounds; this is a problem because this favours the overfitting. Considering that I've **chosen c=1 with 'rbf' kernel**, **c=0,001 with 'linear'** and **c=0,01 with 'sigmoid'**. I didn't use poly anymore because foe the dataset dimension is too slow.

### Change in gamma

For each type of kernel I've tried a c value among (1e2, 1, 1e-2, 1e-3, 1e-4, 1e-5). The obtained results show that:

* For the 'rbf' kernel used with c=1 the variation of gamma value doesn't make any change in the result, unless it is a value minor than 0,001: in this case the smaller is the value, the worst are the performances. Precisely, with gamma= 0,0001 all the evaluation measure get worse by 1% and with gamma=0,00001 they get worse by 2%.
* Both for the 'linear' kernel and for the 'sigmoid' kernel there isn't any variation with the change of gamma value

## Random forest

The random forest is composed by decision trees, so some of the parameter I've considered are the same. Precisely I've tried at first to understand which type of the preprocessing is the better one, then I've tuned, as for the decision tree, the **max\_depth** of the trees and the **class\_weight**.

Moreover, this time there are multiples decision trees so another important hyperparameter to consider is **n\_estimators**, that is how many decision tree do we use to build the model.

The results are in the following tables and each type of preprocessing parameter it's highlighted the better result in green.

### Change in max\_depth

Variation of the values with max\_depth = {5, 10, 15, 30, 45, 60}. I've used here the dataset completely preprocessed with exception of capital-gain and capital-loss features. Note that the better value is 10, that allows to reach the better result until now. Using a **class\_weight** value ={0:3, 1:7} almost all the performance measures get worse about of 1%. Using a complete preprocessing procedure all the values get worse about of the 2-3% and the better value for max\_depth is 15. Without any preprocessing performance are almost like with the complete preprocessing and the better value for max\_depth is 10. It is logica if we think about it, that the better value of max\_depth in the random forests coincides with the optimum for the decision tree, because a random forest is composed by different decision trees.

### Change in n\_estimators

Variation of the values with n\_estimators = {2, 6, 10, 15, 20, 25, 40, 100}: I've tried all those values on the better result of the previous section, but both precision, recall and f-measures still among 84-86%. This probably is due to the fact that the decision trees are all similar one to the other.

# Conclusions

This project has been an opportuny for me to study and use different Machine Learning algorithm looking for their different effects. Sometime is happened the opposite respect to what I expected, but the useful thing is that when there is something strange I've to explain why it happened. The results are however quite good because the dataset respect some of the feature is maybe not big enough. Capital-in and capital-gain are, in my opinion, two feature that are difficult to deal with. It has also been important to use an instance if it has a missing value: at the very beginning I discarded them, but results were strongly worse.

Of all the attempt I've described the better one surely is the random forest used with max\_depth=10: I'm satisfied of the precision and recall values I've obtained, which are 0,86 and 0,87. The goodness of this result however is mostly due to the preprocessing phase that have improved the results largely. The tuning of the various hyperparameters allowed to improve only a few the quality of the model.