UNIVERSITAT DE BARCELONA

Treball final de grau

GRAU DE MATEMÀTIQUES

Facultat de Matemàtiques
Universitat de Barcelona

DAILY ACTIVITY RECOGNITION FROM EGOCENTRIC IMAGES

From manual annotation to automatic classification

Autor: Juan Marín Vega

Director:	**Dr. Mariella Dimiccoli**
Realitzat a:	**Departament de Matemàtica aplicada i anàlisi**

Barcelona, **30 de juny de 2016**

Acknowledgments

I would like to acknowledge the people who made possible this work. First of all, I would like to thank Petia Radeva, who offer me to opportunity to start this work. I would also like to thank Mariella Dimiccoli, my supervisor, who always had a good piece of advice and pushed me in order to improve my work.

I would also want to acknowledge my parents, who have always trusted in me, perhaps too much. And my sister who has always been there to lend a hand and willing to share her experience.

To my friends and colleagues from Universitat de Barcelona. They know I just did it for the lulz because *Together we lose*.

Finally, I would like to thank Neurcar. She did not want me to write these acknowledgments, she is the reason why I did it though.

Abstract

By analysing people way of life we can create methods of prevention and intervention for human behaviour derived diseases. Lifelogging allow us to obtain information, through image capture, of the daily life and the environment in which we move. However, we need to classify those images in order to obtain information, and then to analyse that data to detect behaviour patterns that may be affecting people. But how can we classify thousand of images in a quick way? Automatic classification algorithms, such as convolutional neural networks based techniques and deep learning have shown promising results when classifying images. This work introduces the challenge, first, of realizing a tool for a manual classification, with a website showing images that allow us to easily classify images using batches. Such a tool allows us to create a data set of nearly 20.000 images to, in the second part of the project, realize a fine-tuning over a convolutional neural network trained with ImageNet. After that fine-tuning, the convolutional neural network is combined to obtain the features from the images in order to train a Random Decision Forest classifier. Finally the results are studied. The global accuracy for the CNN system based is that of 58%. A better solution is obtained when combining CNN's and RDF's reaching up to 85% of global accuracy. Thus concluding that the classification system based on training a RDF with the data provided by the CNN, image features and probabilities, is the system offering better results.

Resumen

Analizando el estilo de vida de las personas podemos crear metodos de prevencion e intervencion para enfermedades que derivan de la conducta humana. El lifelogging nos permite obtener informacion, mediante capturas de imagenes, del dia a dia y el entorno en el que nos movemos. Es necesario poder clasificar estas imagenes, si queremos extraer informacion, y analizar estos datos para asi poder detectar que patrones de conducta podrían estar afectando a las personas. ¿Pero como podemos clasificar miles de imagenes de una forma rapida? Algoritmos de clasificacion automatica y tecnicas basadas en redes convolucionales y deep learning han demostrado resultados prometedores a la hora de clasificar imagenes. Este trabajo presenta el reto, primero, de realizar una herramienta de clasificacion manual, una web que muestra imagenes y con la cual es sencillo clasificar imagenes por *batches*. Dicha herramienta permite generar un dataset de aproximadamente 20.000 imagenes para, en la segunda parte del proyecto, realizar un fine-tuning sobre una red neuronal convolucional entrenada con ImageNet. Tras el fine-tuning, se combina la red convolucional de la cual se extraen las *features* de las imagenes con las que se entrena un sistema basado en arboles de decision. Finalmente se estudian los resultados. El porcentaje global de acierto en el sistema basado en una CNN es de un 58%. Se obtiene el mejor resultado con el sistema basado tan solo en la CNN, y de un 85% combinando la CNN con RDF's. Con lo que se concluye que el clasificador basado en entrenar un RDF con los datos que provee la CNN, de la que se extraen las *features* de las imagenes, y se combinan con las probabilidades que tambien ofrece la CNN, es el sistema que ofrece mejores resultados.

Index terms— Machine Learning, Deep Learning, Lifelogging, Annotation tool, Random decision forests, RDF, Convolutional neural networks, CNN, egocentric images, Caffe framework, Automatic classification, activity recognition, activity classification

Contents

Introduction

People way of life is closely related with the diseases that they might suffer. Collecting information and being able to classify it, in order of defining behaviour patterns in an automatic way, opens the path to prevention methods and intervention based in a big volume of data analysis.

Lifelogging is one of the essential pillars when looking for solutions based in data analysis. Using cameras prepared for that purpose, we can collect hundreds of images per day. But those images are useless if we are not able to extract relevant information from it. Analysing behaviours or the way of life of a person implies examining those images in order of obtaining the places where this person goes or the activities that he or she realizes.

Nevertheless it would be absurd to try to analyse the images obtained by a person during weeks or months in a manual way. For that reason we need algorithms that allow us, after a training period, to classify big amounts of images in an autonomous way.

Background

Neural networks are a learning paradigm inspired in how the brain works, and how the neurons are interconnected to produce a stimulus. Deep learning is a set of machine learning algorithms which use neural networks.

Training a neural network usually requires a considerable dataset. Convolutional Neural Networks need to be feed with images, and each image needs to be associated with a particular class. For instance, famous networks such as AlexNet [15] or GoogLeNet [19] were trained over ImageNet LSVRC-2010, and ImageNet LSVRC-2014 respectively. The dataset for the ImageNet Challenge is about 1,2 millions of images for training, 50.000 images for validation and 100.000 images for testing. A technique called *fine-tuning* let us train over a network previously trained, this way the network does not need to *learn* everything from zero. Therefore, fine-tuning a network does not requires millions of pictures.

Deep learning and neural networks have made Machine learning evolve a step beyond anything had ever been seen before. It was on February 10, 1996 when the IBM Deep Blue Machine [20] defeated the World Champion Garry Kasparov in a Chess Match. But there was still a game where computers seemed unable to outperform the human brain. This changed recently when, in October 2015, AlphaGo [20] became the first Computer Go program to beat a professional human Go player.

As Nielsen [16] states, neural networks and deep learning provide the most accurate results to some problems such as image recognition and language processing.

Some previous work has been done in order to classify activities from images. For instance Castro et al. [12] showed that convolutional neural networks are able to classify such type of information.

Goals and motivations

The purpose in this work is to develop the tools needed to address the problem of classifying images obtained through lifelogging. In this case images are egocentrics. To fulfill the purpose a classical algorithm of classification will be use as well as *state-of-the-art* algorithms based in neural networks and deep learning techniques.

In fact, as such algorithms need data previously classified and the neural networks specifically need thousands of data to start showing correct predictions as well, in order of accelerating the process of manual classification, a tool for the easy labelling of those images will also be developed.

The goal of this work is to have a quick manual classification method and a method of automatic classification with a high success rate as well. In addition to those tools, the images used and classified will work for future works looking for going deeper in that area.

Structure of the project

The project will be structured in two main blocks, one more practical in which an Annotation tool will be designed and developed in order to help us labeling thousands of pictures. The second block will be more experimental. In this block we will try to study and understand how Convolutional Neural Networks work. We will also *fine tune* a convolutional network and analyze its accuracy. Furthermore, we will combine the neural network with another classifcation mechanism as Random Decision Forests in order to improve our results. A discussion section will follow the experiments of the second chapter of the project. Finally a conclusion will sum up the experiences and thoughts about this work.

Chapter 1

Annotation Tool

In order to train a Convolutional Neural Network, the network needs to be fed with thousand of pictures. Nowadays, egocentric pictures are easy to obtain thanks to wearable cameras (see Figure 1.1), which allow us to gather as many pictures a day as we want. The camera is set up to shoot two pictures per minute and assuming that the camera is being used around 12 hours per day, it will provide approximately 1500 pictures every day.

Figure 1.1: GetNarrative model 2 camera[a].

[a]Image Source [http://getnarrative.com/]

Labeling pictures one by one does not sound as a sensible idea. Dealing with thousands of pictures and its labels is a hard task to do. Using naive ways to store picture i.e. XML would not work as soon as reaching hundreds of pictures. Moreover, having 2 pictures per minute means that bunch of pictures would fall in the same category[1]. This means that a tool which allows users to select multiple pictures at a time and add a category to all of them just with a couple of clicks would decrease the time spent labeling

[1]Pictures close in time space usually will have the same label

pictures.

1.1 Specifications

Since this is a tool which has to be used by regular users [2] and developers [3], makes sense to provide a centralized access in which developers can add pictures to the system, users and developers can perform the process of labeling pictures, and developers can generate and use the different datasets.

In this context, the easiest way of provide a centralized tool would be developing a web application and hosting it in a server with a public *IP*.

Even though a small web application may be faster to develop, there are potential problems to avoid:

- Privacy and security
- Speed and performance
- Interaction

1.1.1 Privacy and security

The application should be able to provide access to admin users and *non-admins*. Admin users will control the application from the inside, creating users, labels, generating datasets, deleting pictures and labels... Non-admin users will only be able to see and label their pictures. The application will only be accessed through a **Login Form** with *username* and *password*.

Users Users will be divided between admin and non-admin and they will have to use a login page in order to access to the application. There will be differences regarding the possibilities of both roles of users. While non admin users will only be able to browse and label their pictures, admin users will have access to the whole catalog. Admin users will also be able to define new labels, generate datasets, and perform advanced tasks such as delete pictures.

1.1.2 Speed and performance

Even though web browsers are becoming more and more powerful, there is still a huge difference between web and native applications, regarding interaction. One cannot expect a web application to be as *easy-to-interact* as a native one. In web browsers we cannot select content to interact with it. In order to achieve that, the functionality should be developed in *Javascript*, which is the natural language of the web browsers. In order to store information the data needs to be sent to the server, which means sending forms and refreshing the page, unless we develop the functionality using techniques such as *ajax* [4].

[2] Users not related with computer science

[3] End users of the datasets

[4] Ajax is an acronym of Asynchronous Javascript and XML, and let us send information to the browser in a transparent way to the user[`https://en.wikipedia.org/wiki/Ajax_(programming)`]

As we are dealing with thousands of pictures per day, if we want to present pictures to the user in a web application, we need to find a way to load the pictures in the web browser without compromising the usability and without saturating the web browser. Having 1500 pictures in a folder, and taking a look to them is an easy task but trying to load all of them in a web browser at once makes no sense.

Importing pictures Some tasks will not be performed through the web application due the complexity. It would not make sense to upload thousands of pictures to the platform. For that reason, scripts will be provided to make tasks easier. These scripts will be run using a terminal.

There should be a script to perform the task of importing pictures. This script will work recursively from a given folder. It will list all the incoming pictures, move them to a defined folder, get the path and perform tasks such as inserting it into the database, assign a default label and id. The pictures should be in a particular folder. If there is no physical access to the computer where the application is running, pictures could be uploaded using an *FTP* or similar application. Being a relatively lightweight web application, it could be run from a desktop computer. In such case pictures could be just copied to the related folder using a USB Pen Drive or something similar. Other scripts to modify pictures will be provided too, such as an automation script to modify the size of all the pictures in a given folder etc...

Labeling pictures The procedure of labeling pictures should be as much smooth as possible. Pictures must be presented in a catalog, ordered by year, month and day. This means that one should be able to browse them easily. There should be the possibility to select pictures in matches, and apply a category to all at once. While pictures are selected, one must be able to know which of them are currently selected, in case we want to deselect one or many of them. Also it must be possible to add more pictures to the current batch.

Furthermore, there must be a way to know if a picture has been labeled previously as we do not want to label pictures that were previously labeled unless we want to change the current label.

Through the catalog, there should be a way to access only to pictures assigned to a any particular label. As pictures will be accessed by days, there should be a way to see all the pictures of a particular day assigned to any given label. This is really useful when we want to be sure that pictures have been labeled correctly.

As said previously, pictures must not be loaded all at once, or it will collapse our browser, a way to load them dynamically without altering the flow of the current action, should be provided too.

1.1.3 Other things to take into account

Datasets Admin users should be able to generate and download datasets. Datasets should contain a file called *labels.txt*, which will have a list of all the labels used and, at least, one file called *train.txt* file which will contain a two columns: Picture path and Label Id.

Picture path The *path* will correspond to a particular picture. The end user may use this data in a different computer and not where the annotation tool is being executed, so there should be the possibility of defining relative or global paths to be added automatically to the pictures.

Label id The *Label Id* must fit with the labels provided in the file *labels.txt*. Labels will be indexed starting from 0.

Pictures A dataset would not be useful without the pictures that are going to be used, so a mechanism to download pictures should be provided. There should also be the possibility to download only pictures associated to certain labels in case we want to shrink our dataset for testing purposes.

1.2 Methodology

In order to chose a programming language to develop the application, and a framework to avoid *reinventing the wheel*, a set of requisites were defined:

- The language should provide an api to interact with the O.S. to remove or generate files easily.
- Generate zip files.
- The framework should provide a clean way to define a model-view-controller pattern.
- A routing engine [5].
- A template engine.
- *Third-party* plugins friendly

This section will cover the basics of the annotation tool and the reasons of choosing some options against others.

Python There are many options nowadays to develop a web application. On the server side we find languages such as PHP, Java, ASP, ASP.Net, Python and Node.js. A lot could be said about PHP. It is a well known language for web programmers with an active community which has provided powerful frameworks such as Laravel and Symfony. We also have Java and its Spring MVC. But on my personal opinion both are not pretty fun to use due its verbosity. PHP is double-edged because, as I remember, is easy to fall into bad practices. And Java tends to pile-up constraint after constraint and requires you time to get comfortable enough to enjoy the programming experience.

On the other side we have Python and Node.js. Node is seen as a revolution, which pulled together server and client sides and skyrocketed the use of Javascript along the web development community and *back-end* developers. Having previously used Node, Express[6] and Handlebars[7] for personal purposes, it was a strongly considered option.

[5] Routing engine, also known as URL dispatcher is a way to link a url with a functionality.
[6] Express is one of the most used web frameworks for Node [`http://expresssjs.com/`]
[7] Handlebars is a template engine commonly used in conjunction with Express [`http://handlebarsjs.com/`]

Empowering Python for web development we find Django[8], a professional open-source framework used in webs like Instagram, Pinterest, Mozilla. Django is a complete framework full of extensions which make our labor as a developer easier. It provides an intranet, a form designer, a neat template engine and a really powerful routing system[9].

Finally, python was chosen over Node due to it simplicity and ease of use, but instead of Django we would use Flask.

1.2.1 Flask

Flask [6, 13] is not called a framework but a microframework. Flask core is simple but extensible. This means that you are able to develop a small web page with a bunch of lines of codes, or develop more complex systems that grow easily if you keep things organized. Flask uses Werkzeug for the routing engine [10], in Listing 1.1 we see how a simple page in flask works. The url $'/'$ is pointing to the controller *index()* in this case the view is a plain text with $'HelloWorld'$ as a response. The second function maps the url $'/user/ < username >'$ to the controller *show_user_profile(**username**)*. The url parameter $< username >$ will be parsed and given as an input parameter to the controller. Note that *@app.route*$('/')$ is just a decorator, and we can stack as many route decorators as we want.

Listing 1.1: Example of minimum application using Flask

```
1  from flask import Flask
2  app = Flask(__name__)
3
4  @app.route('/')
5  def index():
6      return 'Hello World!'
7
8  @app.route('/user/<username>')
9  def show_user_profile(username):
10     # show the user profile for that user
11     return 'User %s' % username
```

As a template engine we are going to use Jinja[11] which is pretty intuitive to use. One of its features is template inheritance. In Listing 1.2 we can see a basic template in which we define a block called *body* which may be filled by a template that inherits from this one. This template extends a template called *layout.html* with the command *{% extends "layout.html" %}*. It also contains a python loop *{% for user in users %}*.

[8]Django is the most famous web framework for Python [https://www.djangoproject.com/]
[9]Django URL dispatcher [https://docs.djangoproject.com/en/1.9/topics/http/urls/]
[10]Werkzeug is an HTTP and WSGI utility library for Python [http://werkzeug.pocoo.org/]
[11]Jinja is also known as Jinja2 [http://jinja.pocoo.org/]

Listing 1.2: Example of a minimum template using Jinja2

```
1 {% extends "layout.html" %}
2 {% block body %}
3   <ul>
4   {% for user in users %}
5     <li><a href="{{ user.url }}">{{ user.username }}</a></li>
6   {% endfor %}
7   </ul>
8 {% endblock %}
```

In Listing 1.3 an example of how to render a template from a controller method is shown. We expose parameters to the template. This is an easy and a powerful way to pass all the information needed to the template because we can use arrays, dictionaries etc.

Listing 1.3: Example of controller rendering a template

```
1 from flask import render_template
2
3 @app.route('/hello/')
4 @app.route('/hello/<name>')
5 def hello(name=None):
6     return render_template('hello.html', name=name)
```

Flask and WTForms WTForms will help us to define web forms with a couple of lines of code, but not only that, it will also provide a way to validate the forms on the server, in conjunction with Flask-WTF[4], and check if the required input parameters have been filled correctly.

1.2.2 SQLite

In order to store all the information regarding the application: users, pictures, labels... a SQLite[9] database is going to be used. It has been chosen over other options like MySQL[12] or PostgresSQL[13]. The reason of using SQLite is because it is lightweight and small, and it is *self-contained* which means that does not need a server to run it. The idea is to store thousands of rows of information in the database, but not millions.

Flask and SQL Alchemy In order to interact with the database system a python toolkit called SQLAlchemy[5] is going to be used. SQLAlchemy maps the database in order to interact with the data as if it was a collection. It means that we do not need to write SQL statements to gather the data from its tables. Other of the benefits of using SQLAlchemy is that we do not need to rely on an specific Database engine, we only need to define our current service and link it with the ip of the server, or in our case, with the path of our

[12]One of the most commonly used Database servers in conjuntion with PHP [`https://www.mysql.com/`]

[13]Postrgres is maybe, as its web page says "The world's most advanced open source database" [`https://www.postgresql.org`]

SQLite file. If in the future we want to move from SQLite to another system we would only need to migrate our data[14] but none of code would need to be modified.

As we are using Flask, an extension called Flask-SQLAlchemy[3] is going to be used to get the functionality of SQLAlchemy inside our Flask environment.

1.2.3 AngularJS

AngularJS[1] will also be used at some points in the application. There is no need to build the entire application with as a *single-page-application* which is what Angular is used for, but it will be really useful to browse the picture catalog without compromising the usability and the user experience. More about angular properties and challenges found will be said in Section A.1.

1.2.4 Bootstrap and jQuery

Bootstrap[2] will be used as a *front-end* framework. Bootstrap will not only help us to keep things simpler but also to maintain a concordance along the web application pages. It will provide us *ready-to-use css styles* to have a responsive web page and some javascript functionalities.

We will also use jQuery[7] along the website, and some small plugins will be developed and attached to the library. More about jQuery plugins will also be said in Section A.1.

1.3 Development

This section will provide general information about the development process. Some parts will be explained and others will not, as being equals to others previously mentioned. The critical points will be focused as well as some tricks and ideas.

1.3.1 Setting up the environment

Prior to start any work a development environment must be set up. We have used Ubuntu 14.04 LTS 64 bits, Python 2.7.11 provided by the Anaconda 2.3.0 distribution and Python Virtual Environments to keep track of all the python packages needed.

Python Virtual Environments Python Virtual Environments [11] is an essential tool for any python developer. It helps to keep track of the packages needed for your application. The way to use it is pretty simple. Just create a virtual environment inside a folder, activate it and keep it active while you are installing packages or running your application. Each package you install will remain in a particular folder of the current environment activated. So yo can create multiple environments for multiple applications. The benefit is being able to install packages without colliding with your general python distribution, or with different versions of the same package. If a package is required but it is not installed in

[14] SQLAlchemy migrate [`https://sqlalchemy-migrate.readthedocs.io/en/latest/`]

your environment, it will be automatically linked from your python system distribution. In Listing 1.4 a basic set of commands for virtualenv is shown.

Listing 1.4: Basic commands for virutalenv

```
# install virtualenv if it is not available in your system
$ pip install virtualenv

# this command creates the environment inside the venv folder
$ virtualenv venv

# this commands activates the environment
$ source venv/bin/activate

# now your terminal input will look this way
(venv)$

# every python package you install will be inside the venv environment
(venv)$ pip install numpy

# show the packages  or add them into the file
(venv)$ pip freeze > requirements.txt

# installing all the packages from the specified file
(venv)$ pip install -r requirements.txt

# deactivating the environment
(venv)$ deactivate
```

Using virtualenv, you can generate a list of all the current packages of your environment in case you need to install them in another environment. This means that you will be able to clone environments, with its particular package versions, with a couple of commands.

Git and Github Not much to say about Git and Github. Git is your right-hand man as a developer. Git has been used in conjunction with Github due that the application has been developed and tested in about four different computers during the development process.

Development Server Flask includes a development server. It is not reliable for production but it is good enough to not to worry about generic webservers as Apache or Nginx during the development phase. Even though the application is finally using Nginx, it was not until the application was ready for production use where we decided to deploy and move from the internal server to Nginx.

WSGI and uWSGI WSGI[21] is the acronym of Web Server Gateway Interface. It is an specification which defines what a *middleware* should provide. We are going to use

uWSGI[10] as a middleware. It will be the one doing the communication between the web server and our application.

1.3.2 Use cases and Database model

Now that we have sharpened our tools it is time to design and develop the application. Firstly we will define the general use cases of the application, then we will move to the data model, and then we will try to implement the many sections of our application.

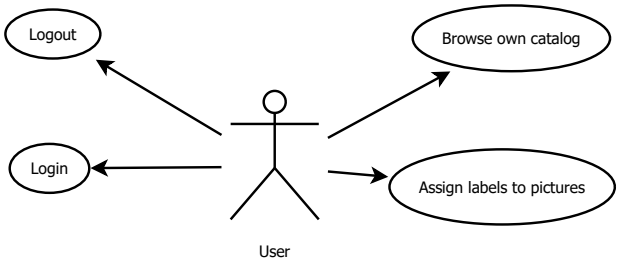

Figure 1.2: Use cases for a user

Even though being a relatively small application, some use cases have been described in order to keep a clear scope about what is going to be implemented.

User and admin use cases The regular user will have limited capabilities, as we see in Figure 1.2 they will only be able to perform four basic operations. In Figure 1.3 we see the additional options the admin will perform.

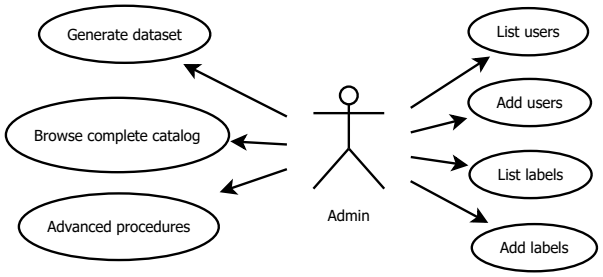

Figure 1.3: Additional use cases for an admin

A user will only be able to see its catalog and label its pictures. Login and logout is also considered. An admin will be able to browse and label the complete catalog as well as listing and createing labels and users. As advanced procedures we contemplate actions as delete pictures which contain certain labels or empty the database. Those procedures were not clear from the beginning and have been changing along the development[15] of the application.

[15]These procedures were changing by the time the application was in use, and while the experimental phase of training the neural networks was in process.

Database model Firstly we define the model and its peculiarities and then we will proceed adding the tables to the database. The system will have have four tables.

- Role
- Label
- User
- Picture

It was decided that the table which contains the information of the pictures will contain also the id of the label associated instead of using a third table to keep relations. This means that a picture will only be associated with one label. The decision was taken due that our classifier will not be a multilabel classifier. In Figure 1.4 we see a representation of the data model. It can be seen in the Picture table that a picture will pertain to a certain user and also to a certain label

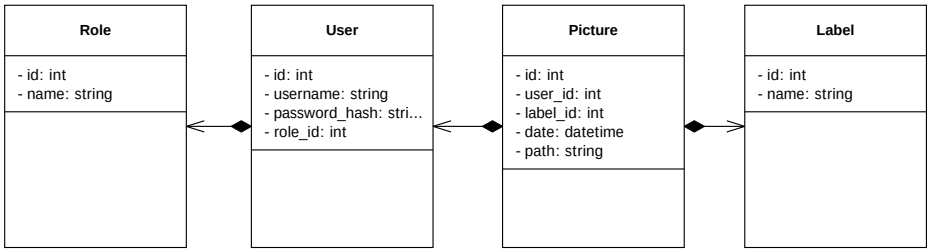

Figure 1.4: A representation of the data

Models with Flask-SQLAlchemy As said in Section 1.2.2, SQLAlchemy will do a tedious work for us. To get the database working we define our model as a python objects with parameters and functions. All the models are stored in the file called `models.py` of the application. We see an example in Listing 1.5. Note that in line 11 we are defining the relation with the table *Role* and *User*. We are also adding a *back-reference* with the keyword *backref='role'*. It will allow us to get the role of a given user. The keyword *lazy='dynamic'* means that the data regarding the relation will only be loaded if it is requested, in order to avoid overhead just by selecting a *Role*.

Listing 1.5: Example of the Role table definition

```
1 class Role(db.Model):
2     # this will be the real name on the database
3     __tablename__ = 'roles'
4
5     # two columns: id and name
6     id = db.Column(db.Integer, primary_key=True)
7     name = db.Column(db.String(64), unique=True)
8
9     # this is a relation, we will be able to gather the users
```

```
10      # associated to a particular role
11      users = db.relationship('User', backref='role', lazy='dynamic')
12
13      # overriding the repr method
14      def __repr__(self):
15          return '<id: %i, name: %r>' % (self.id, self.name)
```

Listing 1.6 shows a piece of the *User* table definition. Note that we have added some properties to keep passwords as a hash representation inside the database. This will be transparent to the user, they will provide their password and the system will hash it and store it. When the user is logging into the application, the provided password will be hashed and compared with the hash stored inside the database. We have also added a foreign key and a naive way to check whether the user is an admin or not.

Listing 1.6: Piece of the User table definition

```
1  class User(UserMixin, db.Model):
2      __tablename__ = 'users'
3      id = db.Column(db.Integer, primary_key=True)
4      username = db.Column(db.String(64), index=True)
5      password_hash = db.Column(db.String(128), index=True)
6
7      # foreign key
8      role_id = db.Column(db.Integer, db.ForeignKey('roles.id'))
9
10     # relationship
11     pictures = db.relationship('Picture', backref='user', lazy='dynamic')
12
13     @property
14     def password(self):
15         raise AttributeError('password is not a readable attribute')
16
17     @property
18     def is_admin(self):
19         return self.role_id == 1
20
21     @password.setter
22     def password(self, password):
23         self.password_hash = generate_password_hash(password)
24
25     def verify_password(self, password):
26         return check_password_hash(self.password_hash, password)
```

The rest of the tables keep the same rules.

1.3.3 Views, Forms and Templates

In order to exemplify deeply the implementation, we are going to go through the flow of viewing the current users and creating a new one. For this task we are going to define a view, a form and a template. But we will also show some examples outside the scope of listing and creating users in order to have a bigger picture of the capabilities provided by the tools used and the framework.

View and create users We need to start somewhere, so let us start defining the views. Views are placed in a file called `views.py`. In this case there are views for logged users and views for not logged users. A methodology called Blueprints[16] is used, and it is a way to split the application in modules, to keep views related in the same module. Even though views for not logged users are just the login and logout pages that may change in the future, that is why this disposition was chosen.

Note that views are closely related to templates, due that a view returns a rendered template. But that does not mean that there are as many templates as views. For example, for the catalog there will only be one template but up to four different views.

Listing 1.7 is a piece of the view for the $'/login'$ page which accepts $methods = ['GET','POST']$ calls. We also see how if a user is correctly logged in it is redirected inside the application with a welcome message, if it is not, it is redirected to the login page with another message.

Listing 1.7: Piece of login view

```
1  @auth.route('/login', methods=['GET', 'POST'])
2  def login():
3      form = LoginForm()
4      if form.validate_on_submit():
5          username = form.username.data.lower()
6          # Gathers the users from the database filtering by name
7          users = User.query.filter_by(username=username).all()
8          ##### hidden code  for simplicity #####
9              # redirect user to inside
10             flash('Bienvenido/a %s' %username , 'success')
11             return redirect(request.args.get('next') \
12                 or url_for('main.index'))
13         # user not valid adds message and redirect
14         flash('Usuario y/o password incorrectos.', 'danger')
15         return redirect(url_for('.login'))
16
17     return render_template('auth/login.html', form=form)
```

In Listing 1.8 we see two more decorators before the controller definition, in this case the functionality they perform is pretty intuitive, *@login_required* checks the user is logged into the application, *@admin_required* checks the user is an Admin. If those cases are not

[16] See flask blueprints [`http://flask.pocoo.org/docs/0.11/blueprints/`]

true, an *error 404*[17] is thrown. Note that in this case, only *GET* methods are allowed and the response view will have the *mimetype 'text/txt'*. We may return any other type of data as zip or json.

Listing 1.8: A view to export the labels in a txt

```
@main.route('/export/labels.txt', methods=['GET'])
@login_required # login required decorator
@admin_required # admin required decorator
def export_labels():
    labels = Label.query.all()
    response = ""
    labels.pop(0) # first label is never used
    for label in labels:
        response += '%s\n' % (label.name)
    return Response(response, mimetype='text/txt')
```

Users view In Listing 1.9 we see a complete view which contains a form to add users. To work with forms, the form needs to be created and exposed into the template adding it as a parameter to the function *render_template*(). Then, if the view is invoked by *POST* the form may be validated and a new potential user may be created. At the time to add a new user it is checked that there is not already a user with the same name in order to avoid conflicts, then the user is created, added to the database session and changes in the data model are committed.

Listing 1.9: The Users view

```
@main.route('/users', methods=['GET', 'POST'])
@login_required
@admin_required
def users():
    form = NewUserForm()
    if form.validate_on_submit():
        user = User.query.filter_by(
            username = form.username.data).first()
        if user is not None:
            flash('There is already a user with that name %s' \
            % user.username, 'danger')
            return redirect(url_for('.users'))

        newuser = User(username=form.username.data.lower(),
            password=form.password.data,
            role_id=form.role.data)

        db.session.add(newuser)
```

[17] See Error 404 [https://en.wikipedia.org/wiki/HTTP_404]

```
19          db.session.commit()
20
21          flash('Usuario %s registrado correctamente' \
22              % user.username, 'success')
23          return redirect(url_for('.users'))
24      user_list = User.query.order_by(User.id).all()
25      return render_template('users.html', form=form, users=user_list)
```

Note that there is no way to delete a user from the system using the web application, this was a design decision. A way to delete users will described in Section A.1.

Users form In order to add users to the database a form is needed. As said in Section 1.2.1 Flask-WTF is used. Working with Flask-WTF is pretty straight forward, we only need to define our form like any other Python class. All the forms related to a module[18] will be defined in a file called `forms.py`. To define a user a name and a password is needed. Also, we need to double check that the user has inserted the password correctly, so we will ask two times for the password. We will also ask for the role to be assigned to the new user.. The current implementation of the form is seen in Listing 1.10, note that there we specify the *validators*, which are the conditions that the user must accomplish in order to register the new user. Those validators are functionalities that will run on the server, here we do not rely in Javascript form validators due that Javascript may be disabled on the client side. The check procedure is triggered in line 6 of Listing 1.9.

Listing 1.10: The form used to create a new user

```
1  class NewUserForm(Form):
2      # the user input
3      username = StringField(u"User",
4          validators = [
5              DataRequired(), Length(1, 64),
6              Regexp('^[A-Za-z][A-Za-z0-9]*$', 0,
7                  'User name must only contain letters and numbers')
8              ])
9      # the password input
10     password = PasswordField(u"Passsword",
11         validators = [
12             DataRequired(),
13             EqualTo('password2', message='Passwords must be the same')
14             ])
15
16     # password again
17     password2 = PasswordField(u'Confirm password',
18         validators = [ DataRequired()]
19         )
20
```

[18] Remember that we are using two modules, for authenticated and not authenticated users.

```
21      # A naive way to create the role SelectField
22      role = SelectField(u'Role',
23          choices=[('1', 'Admin'), ('2', 'User')])
24
25      # the submit button
26      submit = SubmitField(u"Create")
```

Users template The template will be the last step to have defined the whole flow of creating a user. Templates are stored in a folder called *templates/*. We may define a template folder for each module, but that will not be our case. So, a template called `users.html` is defined which inherits from `base.html` where are the main css and javascript files among other html markup. In Listing 1.11 we see a piece of the final template. Note that in this case the form is just processed using the call $\{\{wtf.quick\_form(form)\}\}$ where form is an object given to the template in line 25 in Listing 1.9

Listing 1.11: Piece of users template

```
1  {% extends "base.html" %}
2  {% import "bootstrap/wtf.html" as wtf %}
3  {% block title %}
4   {{ super() }} - Register new user
5  {% endblock %}
6  {% block content_title %}
7      <h1 class="page-header">Register new user</h1>
8  {% endblock %}
9
10 {% block content %}
11  ##### hidden html stuff for simplicity #####
12     {% for user in users %}
13     <div class="list-group-item">
14         {{ user.username}}
15         <span class="pull-right text-muted small">
16         <em>
17         {% if user.is_admin %}
18             Admin
19         {% else %}
20             User
21         {% endif %}
22         id: {{ user.id }}
23         </em>
24         </span>
25     </div>
26     {% endfor %}
27
28  ##### hidden html stuff for simplicity #####
```

```
29
30      ### the form
31          <form class="form" method="post" role="form">
32              {{ wtf.quick_form(form) }}
33          </form>
34
35  ##### hidden html stuff for simplicity #####
36 </div>
37 {% endblock %}
```

View and create labels The procedures explained for the users apply the same for the labels. An admin will be able to see and create new labels within the application.

Catalog

The catalog is the core of the application. Its purpose is to be easy and intuitive. Only a few clicks should be needed to assign labels to a pictures. The catalog is divided in six views and two templates.

- The visible catalog
  - Browse catalog by *user*
  - Browse catalog by *user* + *year*
  - Browse catalog by *user* + *year* + *month*
  - Browse catalog by *user* + *date*[19] + *labelid*
- The catalog API
  - Get pictures by by *user* + *date* + *labelid*
  - Set labels

Easy to use An intuitive way to interact with the catalog to label a picture is shown in Figure 1.5, just click as many pictures as you want and then click a label. The initial state is when there is not any picture selected, so when a label is clicked, the selected pictures will be assigned to that label and they will be deselected in order to start the procedure again.

In order to improve the interaction with the application, there has been developed the functionality shown Figure 1.6. The idea here is to click a picture, and then having the shift key pressed, click another one. Then all the pictures in between will be selected. And the same procedure can be followed to deselect pictures. The state of the last picture clicked will be the state assigned to the rest of pictures in between. This way we can select and deselect multiple pictures at a time. This procedure is common to the one that any operative system provide in their file explorer applications. More about how this has been implemented will be shown in Section A.1.

[19]Date is formed with $year - month - day$ in ISO 8601 format `YYYY-MM-DD` [`https://www.w3.org/TR/NOTE-datetime`]

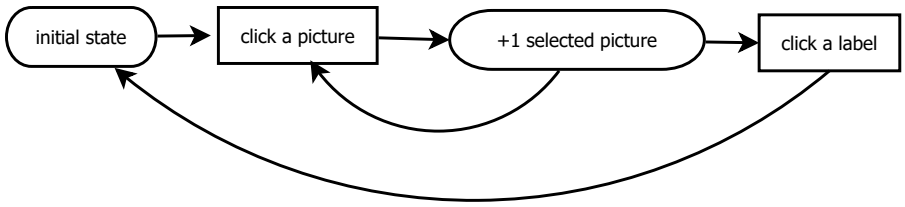

Figure 1.5: Flow to assign a label to a picture

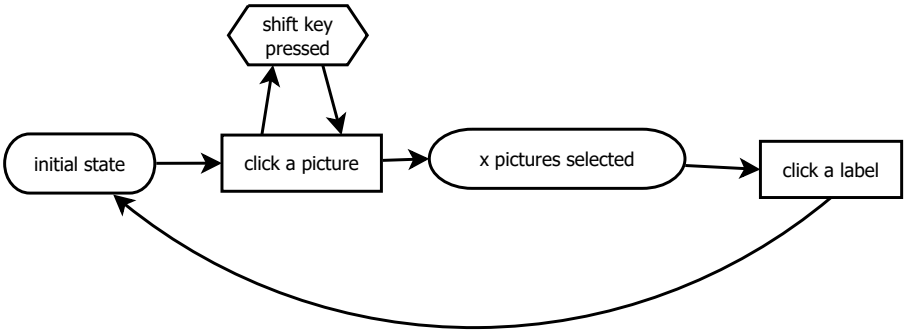

Figure 1.6: Flow to select multiple pictures and assign a label

Easy to understand But how do we represent selected pictures, pictures which currently have a label assigned?. The idea behind the tool is to provide as much information as possible. This means with one sight we need to know which pictures are selected, which pictures have a label, which labels have those pictures...

Figure 1.7: Different states of a picture from not having label to having one

The solution to the problem is shown in Figure 1.7 where is seen how a picture goes from not having a label to having one assigned. Obviously labels will have a color which will be unique, and for the pictures without label, color white will be used, so this way is easy to spot such not labeled pictures. The procedure to remove the label of a picture is shown in Figure 1.8.

Furthermore, showing the pictures as a mosaic, led us to a nice visualization of the cat-

Figure 1.8: Different states of a picture from having label to not having one

alog where is easy to see groups of pictures assigned to a particular labels. An example of how this is visualized is shown in Figure 1.9. More about how this has been implemented will be shown in Section A.1.

Figure 1.9: The current layout of the catalog

Lightweight Now is the time to solve a major problem. How do we load thousands of pictures in a browser without overload it?. The idea here is to use a pagination system mixed with an *infinite scroll*[20]. So, from the beginning there will only be loaded a set of pictures, let us say 75 pictures[21]. As long as the user is scrolling down, when the bottom of the page is reached, another set of pictures will be loaded. But, if the user does not scroll down, no pictures will be loaded.

With this methodology we will not only save bandwidth but will also save time and RAM memory. This process is totally transparent for the user, there will not be any button saying *next page*. In order to achieve this functionality we have mixed some Javascript tools

[20] An infinite scroll is the technique of loading data dynamically as long as the user is scrolling down a webpage.

[21] This will be our pagesize

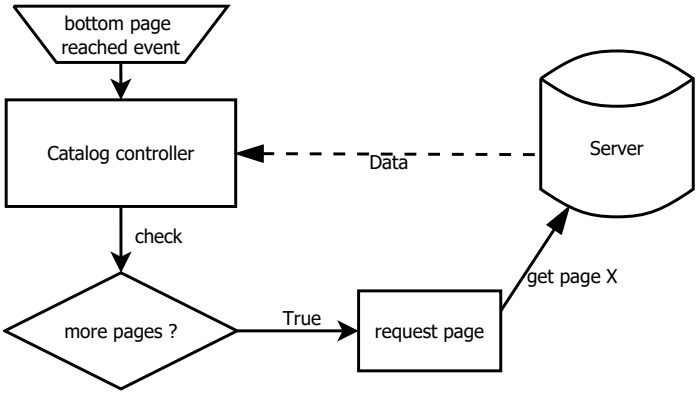

Figure 1.10: How a new set of pictures is loaded

and Ajax requests. There is an illustration in Figure 1.10. When the bottom of the page is reached, an event is fired once, so our catalog controller checks if there are more pages[22] to be loaded. That information was provided with the last set of pictures. If there are more pages to be loaded, an ajax request is performed to the server. The server then generates a response in JSON[23] format with all the information needed.

Listing 1.12: Piece of a JSON response

```
{
    "label-id": null,
    "more-pages": true,
    "next-page": 3,
    "pictures": {
        "77": {
            "datetime": "2015-02-18T17:48:41",
            "id": 77,
            "label": 1,
            "path": "1/2015-02-18/072427e.jpg"
        }
    },
    "user-id": 1,
    "working": true
}
```

Listing 1.12 shows an example of a server response. In this case *"label − id"* : *null* means that pictures in the response would pertain to any possible label. *"more − pages"* : *true* means that there are more blocks of pictures to be loaded and *"next − page"* : 3 is the number of the next page of pictures. This number is important because is the only way to know within the server, which bunch of pictures should be returned. In other words,

[22] In this context a new page means a new set of pictures
[23] See Javascript Object Notation [http://www.json.org/]

this number will be the parameter we will provide when performing the request call. "*pictures*" is a dictionary where each key is a picture id and each value is the information of the picture we need to perform the rest of operations, which would be generate a $< img >$ object with Javascript, assign the path to load that particular picture, assign the css class associated to that picture from the "*label*" value, and insert the picture in the page as any other html chunk.

Dataset

The dataset is the final step of our group of functionalities. A dataset is generally a group of data with a particular structure. If we were working in a machine learning problem trying to predict the final price of a house based on the size, age and place where the house is, the dataset would be a list, let us say a csv[24] file, containing all those properties.

As we are dealing with a classification problem with a convolutional neural network, our dataset contains a file, generally called `train.txt`, which contains the path of a picture and its given label id. The dataset also contains a file called `labels.txt` with the name of each label, where the first line indicates that such label has the id 0.

This is all the information we need to train a convolutional neural network and this is why a Dataset generation section is needed. However, having thousands of pictures in the application does not mean that one requires all of them to train the neural network. The user may want to use part of the pictures to train, and also another part to test the net. The Dataset will be a zip file called `dataset.zip` based on the following available options.

Train, validate and test Any machine learning algorithm has two mandatory phases, the train and the test. Information used in the train phase must not be used in the test phase, otherwise the results will not be real due the algorithm would be predicting over data seen previously. That is why a `test.txt` file is needed. A third file called `val.txt` could also be generated. Usually the validation file is used to test the network in the training phase with pictures that are not in the training set, nor in the test set. This way we see an approximation about the performance we should expect when running the test phase.

The `train.txt` file is always generated, in order to generate the other two files, the user will only need to select the percentage of data required for each file. Selecting 0% for the `val.txt` means that that file will not be generated. Same apply for the `test.txt` file.

Randomize the output The randomize output option is given to avoid generating the dataset in the same order the data is stored in the database. This is optional and is provided due that, generally, randomized data is always used. Thus, if you select to randomize the output while generating the files, you will not need to do the randomization later on.

Extend the relative path The path given in `train.txt, test.txt, val.txt` for each picture is relative to the application[25], so by the time you are feeding your network with

[24] See comma-separated values [`https://en.wikipedia.org/wiki/Comma-separated_values`]

[25] The folder structure of how pictures are stored is explained in Section 1.3.4

pictures, the path might not be the same. Here one will find two options, set the absolute path of where pictures are stored[26], or append a path to the pictures. This second option will append the given path to each picture. In Listing 1.13 there is an example of how this works.

Listing 1.13: Example how path definition works

```
# relative path
user/date/filename.jpg

# append absolute path
/media/storage/application/dataset/user/date/filename.jpg

# append given path: /home/name/fancy/path
/home/name/fancy/path/user/date/filename.jpg
```

Select the labels you want Let us say you have a lot of labels and some of them have a only a few pictures. Perhaps you would want to train the network without those labels. The application will show the list of current labels and the number of pictures associated to that label, so the user will be able to select the labels he want to download. This means that only those labels will appear in the `labels.txt` file and also pictures associated to those labels will be inserted in the `train.txt` and so on.

Downloading the pictures There is no point in appending a hundred thousand pictures in the `dataset.zip` file. The dataset will contain a small python script which will do the work. This file will have been previously configured based on the options specified by the user and also based on the computer where the annotation tool is installed. It will not need any additional parameter to download all the pictures even though optional parameters are accepted. This script will loop through our `train.txt`, `val.txt`... files and download all the pictures with the same structure they are in the application. Listing 1.14 shows how the script could be used.

[26]This is useful if you are training the network in the same computer where the annotation tool is installed

Listing 1.14: Example of downloading pictures

```
# downloading pictures
$ python download_pictures.py
Reading pictures ...
Creating folder structure ...
699 pictures are going to be downloaded using 1 worker(s)
Percent: [###-----------------------------------------] 7% Downloading...

# if you have a good bandwith, try
# downloading pictures with more threads
$ python download_pictures.py --threads 4
Reading pictures ...
Creating folder structure ...
699 pictures are going to be downloaded using 4 worker(s)
Percent: [########################----------------] 63% Downloading...
```

1.3.4 Utilities

The annotation tool is shipped with some useful utilities. The most important, indeed, is the script to import pictures. But there are some others that will do our life easier.

Importing pictures

This script will insert all the information needed inside the database and will move the pictures into a particular folder. There are some previous steps to do before importing pictures:

1. Pictures must be 256 x 256 pixels
2. Pictures must be in a particular folder
3. Pictures must have a particular folder structure

First condition is not mandatory but recommended. The layout of the catalog works with pictures with that size. The script works the next way: It first checks the main folder, and then recursively search for all the pictures inside, adds the paths one by one into the database, and finally, moves all the pictures to a different folder. The folder structure of the group of pictures we want to import must obey one of the two following structures:

- user-id/year/month/day/time.jpg
- user-id/year-month-day/time.jpg

This way, when we loop through the folder, we are able to insert the pictures in the database with their date and time.

Manipulating pictures

Two scripts have been included to help us apply changes to the pictures before or after importing them.

Matlab script This script[27] was made to work in conjunction with the metadata of thpictures. That metadata provide values as the rotation the camera had when took the picture. The script gathers that information and apply a series of changes to the pictures

1. Applies a rotation to the image to get the content parallel to the floor.
2. Crops the picture to get the maximum area after the rotation process.
3. Reduces the size of the picture to 256 x 256 pixels.

Figure 1.11: The process applies a rotation and then crops the picture to get the maximum squared area

Figure 1.11 shows the process of apply such changes to a picture. This script requires you to have Matlab in your system. Moreover, a bash extension has been added to get the script working recursively.

Bash script If you do not have complaints about your pictures being upside down and so on, another script has been added to help you the task to resize them in order to accomplish of the conditions to import pictures. This bash script uses a ImageMagick[28] tool called Mogrify. It searches recursively in a given folder and resizes the images up to 256 x 256 pixels. The script does this without taking into account the proportion of the image.

Deploy Deploying an application is a hard task to do, that is why a deploy script has been included. This script will ask you some questions like the ones in Listing 1.15. Once the deployment process have been completed, the application will be ready to be used

[27] The script was provided by the department of computer vision from Universitat de Barcelona
[28] See ImageMagick and Mogrify [`http://www.imagemagick.org/script/mogrify.php`]

Listing 1.15: Example of how the deploy script works

```
$ python manage.py deploy
Creating database and default tables ...
Insert a name for the Administrator Role (default: Administrator):
$ Superman
Insert a name for the User Role (default: User):
$ Human
Insert a name for the Administrator User (default: admin):
$ Clark Kent
Insert a password for the Administrator User (default: admin):
$
```

Shell The application also provides an interactive Python shell powered by Flask. Being a shell means that you can run whatever python code you want with the incentive that you can interact with the application in real time without having to use the browser. One of the uses would be getting or deleting data from the database.

1.3.5 The docs

The application also contains a documentation about how to install the application and how to use some of the scripts provided. The documentation is made with Sphinx [8], a Python Documentation Generator, and processed in html and pdf. The documentation goes through the following steps:

1. Downloading and installing the application
2. Deploying and executing the application
3. Importing pictures to the application

The documentation is not only made to be as clear as possible but it will also provide answers to some general problems that could appear during the installation process.

1.4 Results

This section will show the Annotation tool, how an administrator would see the tool and how to use it to get the best of it.

1.4.1 Main Annotation tool

Login page

First we see is the login page. The login page only contains a form and a title. Figure 1.12 shows how the login page is shown in a desktop web browser. After inserting our credentials, and being them correct, the system will lead us to the dashboard which includes a welcome message, as seen in Figure 1.13

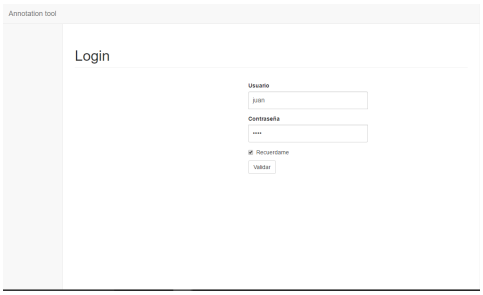

Figure 1.12: Login page

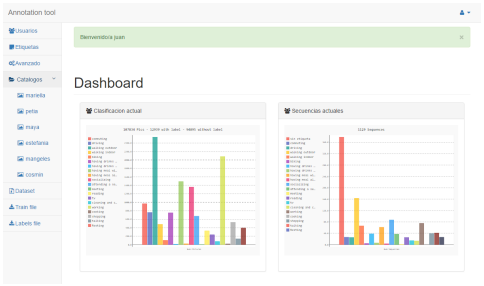

Figure 1.13: Welcome page

Main page

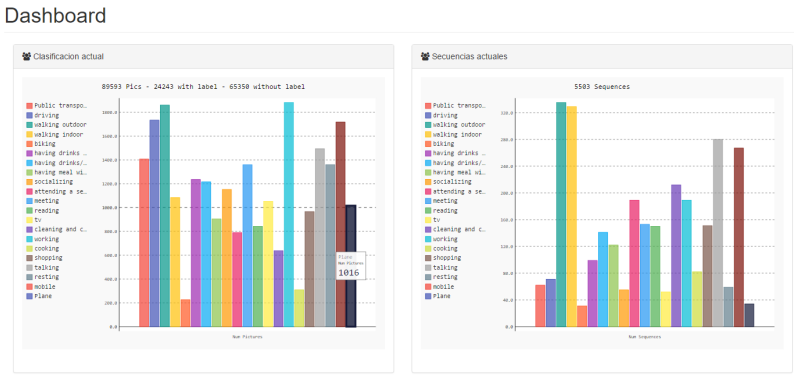

Figure 1.14: Two main charts. The left chart shows the number of pictures per category. The right chart shows the number of sequences per category. A sequence is a group of pictures in a row.

The annotation tool is built as a dashboard and in order to allow users to browse it easily the left main menu will always be shown. The main page page contains useful information on the two charts shown in Figure 1.14. The left chart shows the number of pictures per category. On the bar to the right we see a legend that appears when a mouse is over a bar.

The right chart shows the number of sequences per category. A sequence is a group of pictures in a row. If we assign 50 pictures in a row to a category, that group of pictures will be a sequence. If we assign a picture to a category and such picture is alone among others that pertain to another category, that picture will pertain to a sequence, formed with only that picture. That is why categories such as *Plane* or *Driving* have a low number of sequences, because being in a plane is not a *short time* activity.

Users page

The user page, as shown in Figure 1.15, contains two main blocks, one in which we can see the users registered in the application with their id. The second block contains a form to insert new users.

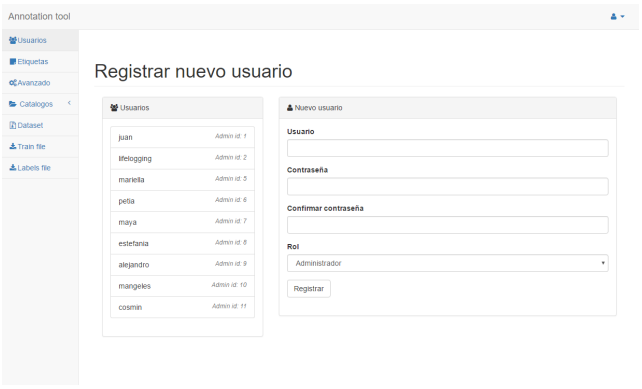

Figure 1.15: The users page shows the list of users and a form to register new users.

Labels page

The labels page also contains two main blocks. On the left the current list of labels is shown. On the right there is a small form to insert new labels. In Figure 1.16 its seen that each label has a unique color assigned.

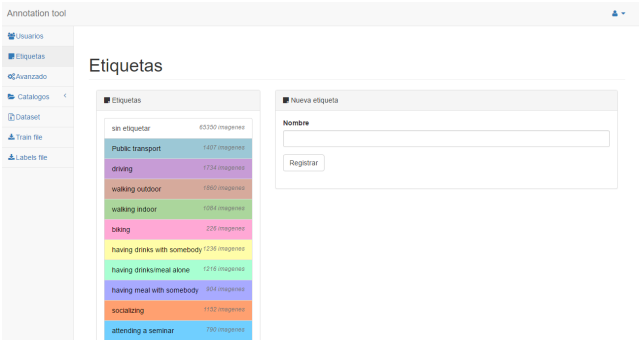

Figure 1.16: The labels shows the list of labels and a form to insert new labels.

Advanced options page

The advanced options page, shown in Figure 1.17 contains two *dangerous* functionalities, that is why form buttons are in red color. In order to run those functionalities a keyword must be supplied by the user.

The Catalog

The Catalog is the center of the application. In Figure 1.18 it is shown the main page of the catalog. Here the user can choose between two options. At the top there is the list of years with pictures. At the bottom list the days with more pictures pending to be labeled is shown.

Figure 1.17: Advanced options page. It contains two dangerous functionalities.

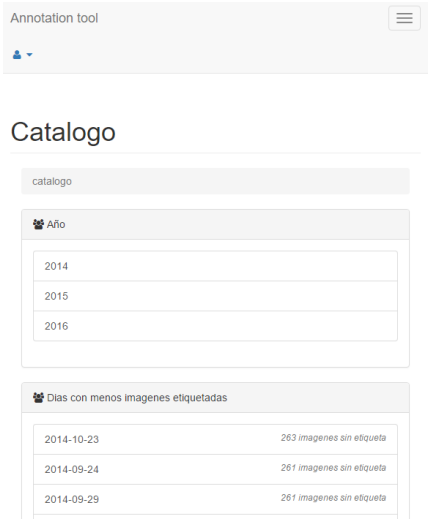

Figure 1.18: Catalog page where can be chosen to browse per year or go straight to a day with pictures to get labeled.

Figure 1.19 shows a list of months after having selected a year in the previous page. Figure 1.20 shows a list of days and labels in order to browse the pictures by day, or by day and label.

Figure 1.21 shows the list of pictures. Here is where pictures are assigned to categories. There is a list of pictures on the right side of the page. Thus, a user only need to select the pictures and then click on a label.

Dataset page

The dataset page is the one allows to generate the dataset containing all the information about pictures and labels. As seen in Figure 1.22, it has a list of categories and a form with options in order to customize the dataset file.

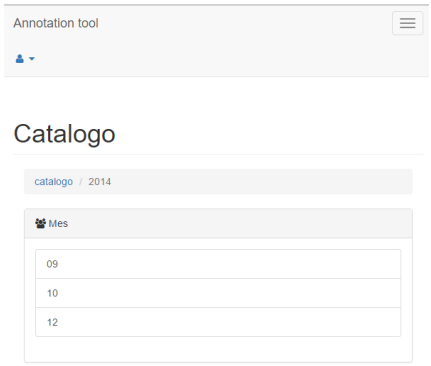

Figure 1.19: Catalog page when a year has been chosen.

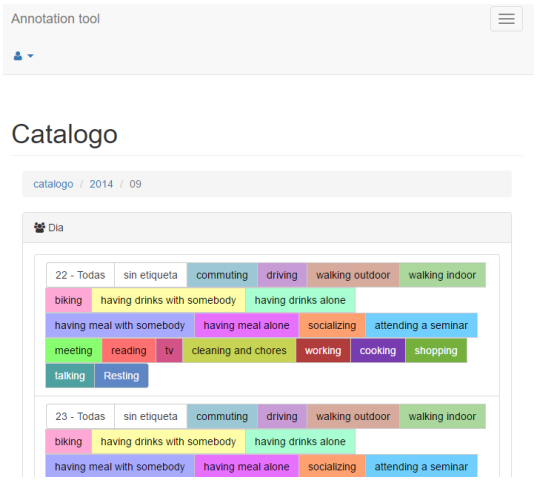

Figure 1.20: Catalog page when a month has been chosen. Here there is a list of the different days that contain pictures and also the different labels abailable in order to see only pictures related to that day and label

The Catalog is the center of the application. In Figure 1.18 it is shown the main page of the catalog. Here the user can choose between two options. At the top there is the list of years with pictures. At the bottom list the days with more pictures pending to be labeled is shown.

1.4.2 The docs

As stated previously, the annotation tool provides a documentation it *html* format. This makes it easy to browse and includes a search tool to help users finding *key words*. Figure 1.23 shows one of the pages of the documentation.

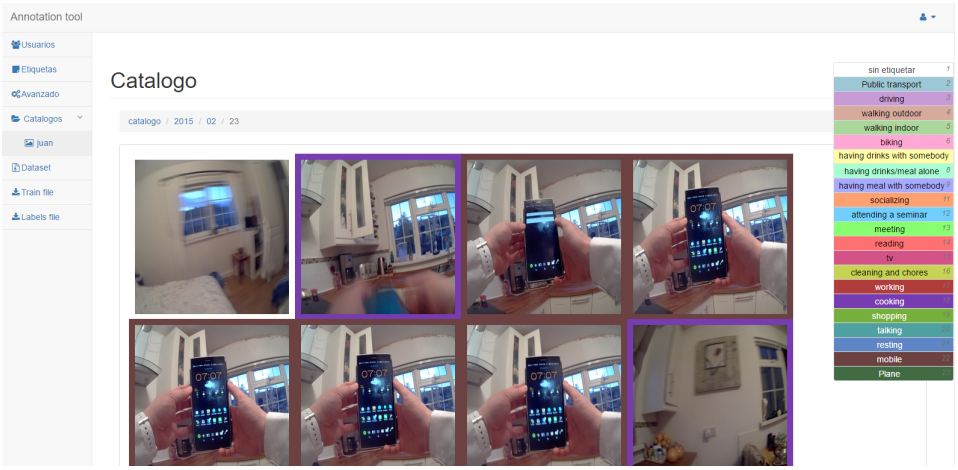

Figure 1.21: Catalog showing pictures to be assigned to a label. The list of labels is shown on the right side of the page.

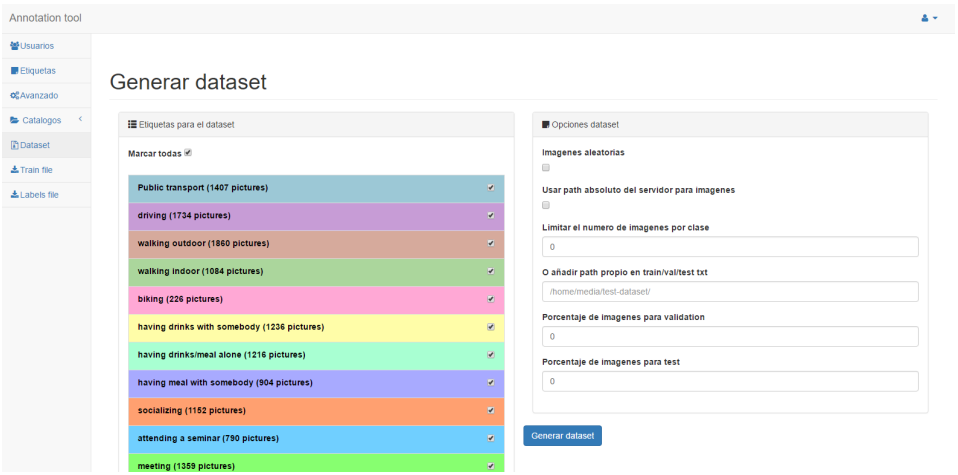

Figure 1.22: Dataset page. On the left there is the list of categories to be download. On the right a form with some options to customize the dataset.

Figure 1.23: The documentation is provided in html.

Chapter 2

Activity recognition

2.1 Specifications

Our classifier is built to infer activities from images. Nowadays, wearable cameras are cheap items and people are more willing to use in a daily basis. Cameras like GoPro[1] or NarrativeClip [2] are powerful enough to obtain hundreds of pictures per day. Also the quality of the pictures has increased enough to make those pictures good enough to work with them in computer vision applications such as object recognition. Even though algorithms eventually work with small picture size, picture transformation like image rotation, is sometimes mandatory to work with such pictures. Image rotation is usually followed by a reduction of the image size in order to have squared pictures (see Figure 2.1). Thus, having better cameras providing great pixel definition is empowering computer vision and image classification algorithms.

Figure 2.1: After applying a rotation of an image a crop is needed in order to keep a rectangular picture. During this process, the size of the image is reduced proportionaly to the rotation applyed

Once the image is classified, the systems returns an activity. The activity is a word or a sentence which provides a general and a simple description of what is visualized in the picture. Example activities are for instance *mobile*, *reading* or *having meal with somebody*. During the trial and error process, the activities list varied due the necessity to be

[1] `https://gopro.com/`
[2] `http://getnarrative.com/`

more specific or more lax with the information we were working with. Eventually a final activities list containing 21 different activities was defined which is sown in Table 2.1.

Activities
Public transport
Driving
Walking outdoors
Walking indoors
Biking
Having drinks with somebody
Having drinks\meal alone
Having meal with somebody
Socializing
Attending a seminar
Meeting
Reading
TV
Cleaning and chores
Working
Cooking
Shopping
Talking
Resting
Mobile
Plane

Table 2.1: Final activities list

2.2 Methodology

Image classification is a problem which can be solved using machine learning techniques. Our classifier is built using Random Decision Forests which is a classic machine learning algorithm and *state-of-the-art* techniques such as Deep Learning and Neural Net-

works. In our case, the network used is a Convolutional neural network which slightly differs from the *non-convolutional* ones.

2.2.1 Supervised classification

Deep learning and Neural Newtorks are based in supervised classification techniques. The way it works is training the network with multiple data as an input which has been already labeled. After the training process the network should be able to *predict* or classify pictures which have not been used during the training process.

The most simple representation of a classifier is shown in Figure 2.2 which is the basic function of a linear *or logistic* classifier.

$$Wx + b = y$$

Figure 2.2: Linear classifier function

Here x represents the input data, W represents the weights applied to the input, b represents a bias factor and y the output value.

Linear classification In order to be able to classify from an input we must find the values (weights and bias) that are good giving the results we want[18]. As we already know the output value we want, because we previously classified our input data for the training process, we may be able to tweak the weights and biases to produce the expected output given the input data. The result of the linear classifier function is sometimes called **Logit** or **Score function**. An overview of a classification procedure is seen in Figure 2.3.

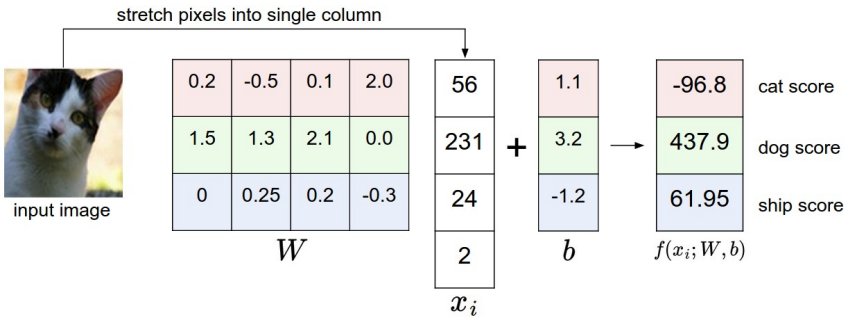

Figure 2.3: Example of classifying from an input image to a class[a]

[a]Image Source [http://cs231n.github.io/linear-classify/]

Softmax A classifier function usually does not output a single value but as many as categories we are working with. The output array is made by numbers and the higher the number in a cell, the higher the probability of the input data to pertain to a certain category. But those numbers are hard to compare if they are not normalized. In order to

obtain the numbers between 0 and 1 we use a function called softmax. Softmax is shown in Figure 2.4. This function will normalize our resultant vector to values between 0 and 1 and the more close to 1 is a particular number, the bigger the probability of the input to pertain to such label or category. Cells with a value close to 0 represent that the input data is not related with such category.

$$f(v_i) = \frac{e^{v_i}}{\sum_j e^{v_j}}$$

Figure 2.4: Softmax function

Learning process The learning process is made by a combination of techniques. First of all we need to compare our output category with the category we have already given to the input data. The way a category is assigned to an input data is also with a vector full of zeros and a one in a format called *1-hot labels*. This vector has a 1 for the category assigned and a 0 for the rest of categories. Thus, the output of the softmax function has to be processed to have the same format as the input category and then we compare the output vector with the input vector in a process called cross entropy function. This is the way we compare the distance between two probability vectors.

$$H(p,q) = -\sum_x p(x) \log q(x)$$

Figure 2.5: Cross entropy function between p and q, where p is the *true* category and q is the predicted

The distance between those vector represents the cost of our function, which is the difference between the real value and the predicted value. This cost is the value we want to minimize. Here Stochastic Gradient Descent is used, which is an approximation, or simplification, of a Gradient Descent, the usually used algorithm to find the local minimum of a given function. Stochastic gradient descent is computationally less demanding providing faster solutions. Each iteration of our algorithm our function is a step closer to the local minimum. Eventually, when comparing the cost function of iteration i and iteration $i-1$ the difference is less than d where d is a value we defined to agree that the function has converged, we can stop our algorithm.

Validation and Test In order to check if our classifier is getting the expected results we need to test it. But we cannot use the same data we have been using to train because we would not be able to know that the classifier is generalizing properly. This is why we need a validation set. A validation set is a subset which contains data which is not present in the training dataset this way we can use our classifier to predict the results and compute the loss function the same way we use the train dataset. So after n iterations using training data we perform m iterations using validation data.

Once we have decided to stop the training process we need to perform the final test. As we have been validating the classifier with validation data, our classifier has been also memorizing information of the validation dataset, so we cannot use that data to perform this final test. That is why we need the Test dataset. The Test dataset allows us to measure the real performance of our classifier.

2.2.2 Neural Networks

ReLU The description above is common on machine learning algorithms but it represents a linear classifier. In Figure 2.6 we see an example of a linear classifier with two input parameters.

$$w_1 x_1 + w_2 x_2 + b = y$$

Figure 2.6: Linear classifier function with two variables

Even though this function allows us to work with two input parameters, and it is extensible to n input parameters, the resultant will always be a linear function. To extend our classifier with *non-linearity* we use an activation function called Rectifier, so once we apply the rectifier to our function we have a *rectified linear unit*. The output of a unit *or neuron* after applying the rectifier is shown in Figure 2.7. This function will always output 0 or a value greater than 0. In order to have a deeper network, we can stack different layers of linear operations followed by a rectifier unit to have what is called *neural network*

$$f(x) = max(0, x)$$

Figure 2.7: Rectifier

Backpropagation Backpropagation is used to update all the weights we have in our different layers. This is made computing the derivatives of our function. As our network is made by simple operational blocks we can perform the derivative of the function by using the *chain rule* and obtain the partial derivative for each of the weights we have in our function. This partial derivative shows the sensitivity the function has respect changes on that given variable. Modifying the weight applied to a given variable we expect to alter the overall outcome of the global function and, as a result, reduce the value of the loss function.

Dropout A problem that may occur during training is called overfitting, this means that our classifier will not be able to generalize well due to getting specialized to the data it has already been training with. One technique to reduce this problem is called **Dropout** [14]. While we are training the network, similar inputs tend to produce similar values in our activation functions and as a result, similar output in our loss function. This prevents our

network from learning. Dropout disables activations randomly, which means that some of our activation functions will return 0 whatever was the value it has as an input. This way our network cannot rely on any given activation and is forced to learn redundant representations for similar inputs. This makes the learning process slower but gives a stronger network. In Figure 2.8 a representation of how dropout alters the behaviour of the network is shown.

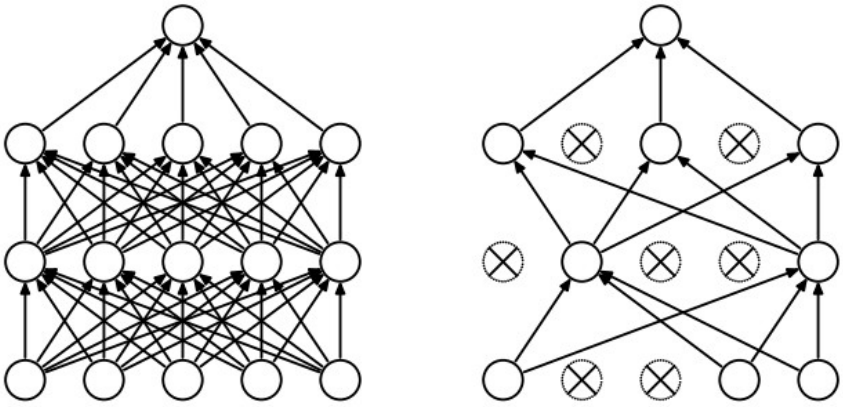

Figure 2.8: An example of applying dropout to a network[a]

[a]Image Source [14]

2.2.3 Convolutional Neural Networks

Convolutional neural networks are a type of neural networks specialized to work with images. This network shares parameters accross space, which means that the input data is evaluated as a whole and not pixel per pixel. A color image is represented as a 3 layer matrix, each layer represents a color Red, Green and Blue, and each cell contains a value between 0 and 254 which represents the intensity of the color. In the R layer, 0 means white and 254 means totally red.

Taking a small path *or kernel* from the original picture with width w height h and depth 3, and performing a convolution at some point of the picture, it gives us a column with depth k. k is the number of filters we are applying.. Getting all those columns we can build a block with a reduced w and h unless the kernel size was equal to 1. This block only connects cells *or neurons* from one layer to another, not within them, this is what is called *depth column*. An example of an output volume is shown in Figure 2.9.

Stacking different convolutions we end up having at the top of our network a column with $w = h = 1$ and a depth equal to the number of labels we are working with.

Convolutional layer In a convolution each pixel es multiplied by its neighbors using the filter or *kernel* as weights. As we said previously a convolution produces a reduction of the height and weight of an input. In this case we increase the depth k. k is the number

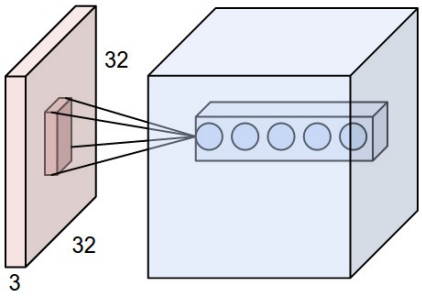

Figure 2.9: Output volume and depth column representation[a]

[a]Image Source [http://cs231n.github.io/convolutional-networks/]

of filters we want to apply to the input and also the depth of the convolutional layer. So after a convolution we will have k layers[3] *or feature maps*. Thus in our convolution volume we would have different activations in different layers. As each layer shares weights, one particular layer could be being activated by horizontal lines, another one by vertical lines etc. The size of the kernel defines the size of w and h for the output convolution volume. Sometimes a convolution with a kernel size of 1x1 is used in order to get a non linear and deeper model from a given input.

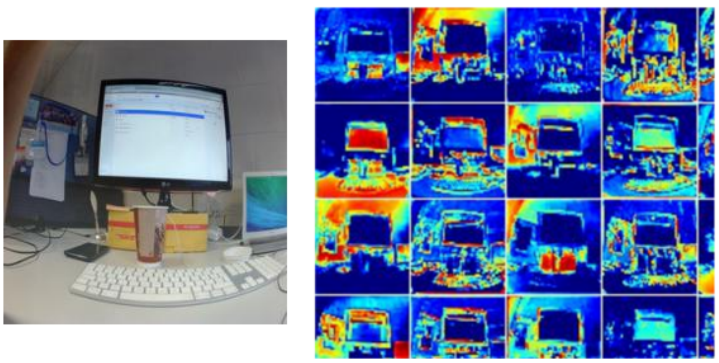

Figure 2.10: Visualization of some layers or *feature maps* after applying a convolution

Pooling layer Pooling layers are often used after convolutions to reduce the spacial size of that given convolution, and as a result, the number of parameters. Pooling layers work the same as convolutional ones but the convolution operation is replaced by another one. At every point of our feature map instead of multiplying values we keep the max for the *max pooling* or the average in a *average pooling*.

Fully connected layer Fully connected layers connect neurons of a particular layer with all the neurons in the previous layer.

[3]A convolutional layer produces a 3 dimensional block, the third dimension is usually called the number of layers

Inception Module Inception module was presented in [19] and is a combination of different convolutions. Instead of performing different convolution at different points of the network we compose a convolution module from a given input running different convolutions and an average pooling. Then, all the outputs are stacked one after another. Figure 2.11 shows the architecture of an Inception Module.

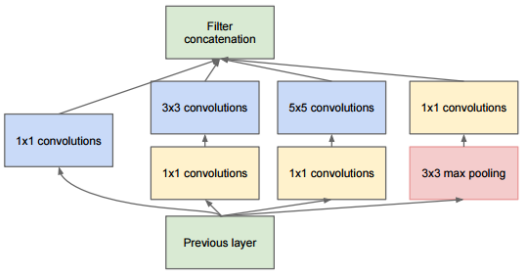

Figure 2.11: Inception module with dimensionality reduction[a]

[a]Image source[19]

2.2.4 Random Decision Forests

Our convolutional neural network is going to be complemented with a Random Decision Forest **RDF**. The RDF is going to be attached at the top of the neural network, thus, instead of training the RDF with pictures it is going to be trained with a set of features given by the network. In this case, we will use the neural network to collect a set of features before the last layer, which is the layer which classifies the input data. Eventually we will not have a *1-hot label* vector but a set of features which depends on the network we are using. This means that the final classification step will be performed by the RDF. Once the system is trained, in order to test the overall performance of the classifier, a similar process will be performed for the test Dataset. A test picture will be the input of the convolutional neural network and the outputted set of features will be the input for the RDF. This procedure has been used in [12] with good results.

2.3 Development

In order to train a convolutional neural network, a deep learning framework is required. Nowadays, there are many deep learning frameworks available on internet. As many of them are ready to be used in regular neural networks and convolutional ones, the only requisites to choose one were the following:

- Python interface
- Good documentation

A good documentation is basic for any newcomer. Dealing with a new framework is a hard task and a good documentation is needed if you want to be able to cope with the

learning curve. Python interface may not be mandatory but it is useful if you want to perform several tests and automate them.

The computer used had the following specifications:

- Ubuntu 14.04 LTS 64 bits
- 16 GB Ram
- Nvidia GTX 960 4GB GPU.

Some tests were done to decide between Tensorflow and Caffe. Finally Caffe was chosen due it was easier to perform fine-tuning over networks trained with ImageNet, and its memory efficiency [4]. All the tests were performed using the GPU.

For the RDF a python implementation from scikit-learn[17] library was used.

The dataset provided contains around 80,000, but finally only 18644 had been used. 13991 have been used for training the neural network, 1857 for validation and 2796 for testing. Even though most of the pictures were concentrated in categories such as working, driving or walking outdoors, there were only two categories below 600 pictures, and only 4 categories below 800 hundred. 12 categories were over 1000 pictures. It would have been easy to increase the dataset in, let us say, 10,000 pictures, but that would have only been for pictures of 3 or 4 categories. In order to get the dataset as much balanced as possible, the number of pictures per category was limited to 1000 pictures. In Table 2.2 the final number of pictures for training, validation and testing is shown.

Having the dataset ready we proceeded developing the environment needed to perform all the experiments. As we are using Python, we decided to use iPython notebooks to create all the code and run the experiments. This was chosen over other IDE's because the environment was set as an external server, through ssh and with the graphical interface disabled when running the experiments, so we only needed to run iPython notebook as a server and browse to the ip from any other computer.

2.4 Results

2.4.1 Training the networks

Two neural networks were used for the fine-tuning process, AlexNet and GoogLeNet. The base weights were both extracted from the Caffe Github Repository.

A file called `fine-tuning` has been created to perform the training process. Here we load the `train.txt`, `val.txt` and `labels.txt` which pertain to our dataset. In order to train with Caffe we need to define 2 main files *train_val.prototxt* and `solver.prototxt`. The *train_val.prototxt* file specifies the topology of our network while training and validating. The content is similar to a JSON file, and it is usually defined by hand even though Caffe provides a python interface to define it.

For AlexNet we opted to keep separated the `train.prototxt` specifications and the `val.prototxt` specifications. In both the last layer was renamed to $fc8\_activities$ and the parameter num_output set to 21.

[4] There were memory problems running an AlexNet adaptation with a batch of bigger than 32 pictures with Tensorflow. These problems might be produced by a bad implementation of our algorithms

class	Training set	Validation Set	Test set
Plane	764	92	143
Mobile	733	116	149
Resting	740	104	154
Talking	761	90	149
Shopping	750	81	134
Cooking	230	34	45
Working	748	94	158
Cleaning and chores	455	67	114
Tv	743	106	151
Reading	643	64	134
Meeting	751	107	140
Attending a seminar	595	74	118
Socializing	754	98	147
Having meal with somebody	687	86	127
Having drinks/meal alone	752	88	158
Having drinks with somebody	770	98	131
Biking	163	25	38
Walking indoor	704	134	160
Walking outdoor	750	81	168
Driving	746	103	150
Public Transport	752	115	128
Total	13991	1857	2796

Table 2.2: Number of pictures per class and category

For GoogLeNet was a bit different because GoogLeNet network provides 3 outputs at 3 different levels. The names applied were $loss1/classifier\_mod$, $loss2/classifier\_mod$ and $loss3/classifier\_mod$ respectively, and the parameter num_output set to 21 as in AlexNet. In this case, train and val are defined in the file $train\_val.prototxt$.

The rest of the parameters for both networks, remain as in the default specifications.

Both nets were trained for 10 epochs. In the AlexNet adaptation, with a batch size of 64,

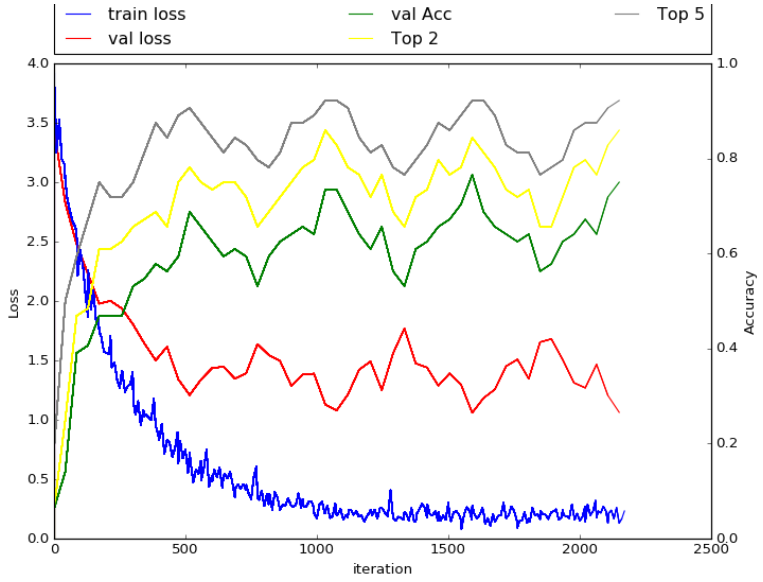

Figure 2.12: AlexNet Train loss, Validation loss and Validation accuracy Top 1, 2 and 5

the total number of iterations was around 2180. GoogLeNet took around 4000 iterations to perform the 10 epochs with a batch size of 32. The batch size for GoogLeNet was 32 due that it took around 4GB of GRAM. At the beginning a batch size of 24 was used until we tried disabling the graphical interface to freed around 70MB of GRAM from the graphical unit. Using the command `nvidia-smi` we check the memory usage as shown in Figure 2.13.

```
+-----------------------------------------------------------------------------+
Thu May 26 18:53:43 2016
+------------------------------------------------------+
| NVIDIA-SMI 352.79     Driver Version: 352.79         |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce GTX 960     Off  | 0000:01:00.0     Off |                  N/A |
| 22%   56C    P2   110W / 160W |   4023MiB /  4095MiB |     98%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID  Type  Process name                               Usage      |
|=============================================================================|
|    0     26799    C   /home/hermetico/anaconda/bin/python           4008MiB |
+-----------------------------------------------------------------------------+
```

Figure 2.13: GoogLeNet memory usage with a batch size of 32

The validation phase has been performed 5 times per epoch. As we are finetuning the base learning rate has to be small due that the network is already trained to classify different objects. The base learning rate was set to 0.00003 for the AlexNet and 0.000037 for GoogLeNet. The learning rate has been drop each 5 epochs and a snapshot has been

produced each 10 epochs. In Table 2.3 the final solver used for each network is shown.

	AlexNet	GoogLeNet
test iterations	20	11
test interval	43	109
base lr	0.00003	0.000037
display	10	27
max iterations	2180	4370
lr policy	"step"	"step"
gamma	0.1	0.1
momentum	0.9	0.9
weight decay	0.005	0.005
step size	1090	2185
snapshot	2180	2185
solver mode	GPU	GPU
solver type	SGD	SGD

Table 2.3: Solver definition for AlexNet and GoogLeNet fine-tuning

Both network used the same `train.txt` and `val.txt`. After running the solvers for both networks both resultant weights were stored in `.prototxt` files. In Figures 2.12 and 2.14 the process of training both networks is shown.

The notebooks in which the networks have been fine-tuned are called `finetuning-alexnet.ipynb` and `finetuning-googlenet.ipynb`.

2.4.2 Testing the networks

Once the network is trained a test to evaluate the final performance is needed. A `deploy.prototxt` file has been created for both networks. For AlexNet the layer with the soft-max computation has been renamed to `probs` and linked to the $fc8\_activities$ layer. For GoogLeNet, the layer has been called `probs` too, and, in this case, linked to the $loss3/classifier\_mod$ layer. Two different tests were performed over both trained networks:

- Class accuracy
- Total accuracy

The class accuracy is performed dividing the `test.txt` classes, and taking the average of all the pictures that have been classified correctly. The global accuracy test is performed with all the pictures without doing the class distinction.

Class	AlexNet Accuracy	GoogLeNet Accuracy
Plane	60%	74%
Mobile	75%	75%
Resting	90%	86%
Talking	75%	59%
Shopping	54%	61%
Cooking	7%	44%
Working	72%	73%
Cleaning and chores	26%	43%
Tv	91%	86%
Reading	78%	79%
Meeting	12%	27%
Attending a seminar	58%	69%
Socializing	39%	69%
Having meal with somebody	52%	44%
Having drinks/meal alone	53%	60%
Having drinks with somebody	55%	53%
Biking	5%	18%
Walking indoor	18%	33%
Walking outdoor	79%	82%
Driving	87%	88%
Public Transport	38%	41%
Global Accuracy	58%	62%

Table 2.4: Class and global accuracies for AlexNet and GoogLeNet after finetuning

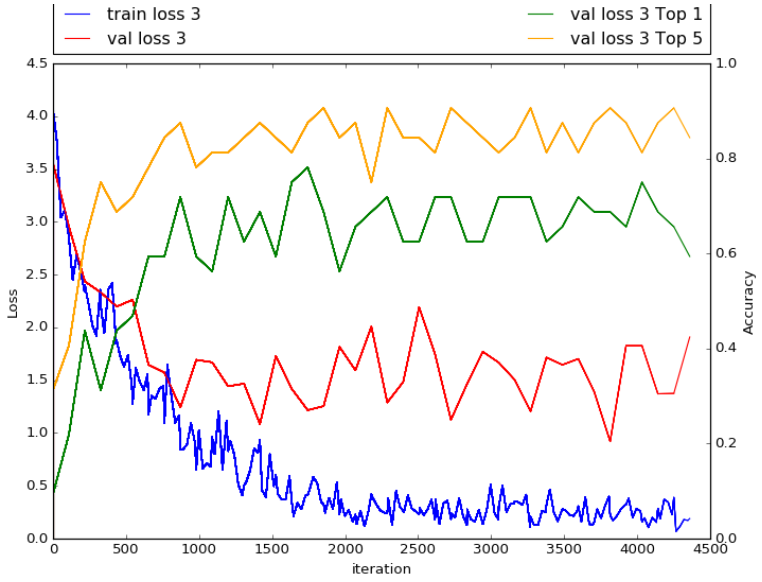

Figure 2.14: GoogLeNet Train and Validation loss, Validation Top 1 and Top5

The Table 2.4 shows the correlation between the classes and it accuracies for both networks and also the global accuracy. In Figures B.1 and B.2 confusion matrices for AlexNet and GoogLeNet are shown.

The notebooks in which networks have been tested are called `accuracy-activitiesnet-alexnet.ipynb` and `accuracy-activitiesnet-googlenet.ipynb`.

2.4.3 Training and testing the RDF

In order to improve the classifier another step has been added to the pipeline. We have combined the output of the cnn with a Random Decision Forest. As in [see 12], we have trained the RDF with the probabilities of the network but not pixel information or histogram. The random forest classifier has been defined with 500 estimators. The training process for the RDF was the same for both systems.

The cnn's were loaded as a deploy network and the input data consisted in both `train.txt` and `val.txt`. The validation set was used because there is no validation phase when training the RDF so it made sense to increase the traning dataset.

After training the RDF an accuracy test has been performed the same way we did previously. In Table 2.5 we see the correlation of classes and accuracies for both systems, AlexNet + RDF and GoogLeNet + RDF. As previously, in Figures B.3 and B.4 confusion matrices for AlexNet + RDF and GoogLeNet + RDF are shown.

Notebooks used to train and test the CNN + RDF are `accuracy-activitiesnet-alexnet-randomforest.ipynb` and `accuracy-activitiesnet-googlenet-randomforest.ipynb`.

class	AlexNet+RDF Accuracy	GoogLeNet+RDF Accuracy
Plane	86%	90%
Mobile	82%	76%
Resting	96%	98%
Talking	73%	71%
Shopping	58%	67%
Cooking	31%	37%
Working	89%	86%
Cleaning and chores	54%	64%
Tv	92%	92%
Reading	72%	71%
Meeting	75%	75%
Attending a seminar	85%	84%
Socializing	69%	65%
Having meal with somebody	67%	67%
Having drinks/meal alone	72%	65%
Having drinks with somebody	75%	67%
Biking	68%	68%
Walking indoor	53%	55%
Walking outdoor	82%	74%
Driving	97%	98%
Public Transport	75%	72%
Global Accuracy	76%	76%

Table 2.5: Correlation between classes and accuracies of both CNN+RDF systems.

2.4.4 Training and testing the RDF with more features

In order to add more contextual information, we have been opted to incorporate features from previous layers to the RDF.

For AlexNet we have run experiments with the following layers:

- soft-max probabilities + fully connected layer $'fc6'$ with a total of 4117 features

- soft-max probabilities + fully connected layer $'fc7'$ with a total of 4117 features

And for GoogLeNet the experiments were the following:

- soft-max probabilities + fully connected layer $'pool5/7x7\_s1'$ with a total of 1045 features

In Table B.1 a sum up of the different combinations is shown. Also, in Figures B.5, B.6 and B.7 we show the confusion matrices for each different combination of features.

Notebooks used to train and test the CNN + RDF with extended features are `accuracy-activitiesnet-alexnet-randomforest-extended-features.ipynb` and `accuracy-activitiesnet-googlenet-randomforest-extended-features.ipynb`.

2.5 Discussion

From the beginning, the classification has not been a regular process. Labeling 20.000 pictures is not a task that can be done in a couple of days even with the annotation tool. This pushes us to make some errors, for instance, a type of picture that at some point made sense to be labeled as *Socializing*, another day could make sense to be labeled as *Having drinks with somebody*. This leads us to produce some overlapping of classes along the dataset. That could be one reason for having low accuracies at some classes.

Along the experiments we see an improvement in the accuracy reaching a 85% of global accuracy for both systems, based in AlexNet and GoogLeNet. In Table 2.6 we can see how with a classifier based only in a convolutional neural network global accuracy reaches only a 58% which, taking into account that there are 21 labels, is not a bad result. GoogLeNet gets a slightly better 62% of global accuracy.

AlexNet	**CNN**	**CNN (probs) + RDF**	**CNN (FC7, probs)**	**CNN (FC6, probs)**
Global accuracy	58%	76%	84%	85%

Table 2.6: AlexNet based classifiers and accuracies

GoogLeNet	**CNN**	**CNN (probs) + RDF**	**CNN (pool5/7x7, probs)**
Global accuracy	62%	76%	85%

Table 2.7: GoogLeNet based classifiers and accuracies

For the CNN based system, we see a small improvement of GoogLeNet over AlexNet of 4%. However, we see better improvements in accuracy for some classes in Table 2.4 like *Cooking* which goes from 7% in AlexNet up to 44% in GoogLeNet. The same for other classes as *Cleaning and Chores*, *Meeting*, *Biking* or *Socializing*.

If we take a look to the confusion matrices, for instance in Figure B.1, which shows the

result for AlexNet, we see that pictures that pertain to the class *Meeting* are usally categorized as *Talking*, or even *Reading*. This does not happen in GoogLeNet. Figure B.2 shows that up to 27% of the pictures are correctly categorized. This phenomenon is even worse for the *Cooking* category, where only a 6% is correctly categorized in AlexNet and most of the pictures are classified as *Cleaning and Chores*, *Reading* or even *Mobile*.

It seems that GoogLeNet achieves better results on classes that need more complexity at the time to be described and AlexNet gets stronger among classes which relies on central objects. For instance AlexNet gets better results for classes such as *TV* or *Mobile*. It is also better on the *Resting* class, whose pictures are usually a ceiling. On the other hand, GoogLeNet does not shine specially in any class but does not fail in any particular class either, as AlexNet does for example in *Biking* with a low 5%.

When modifying the pipeline and adding the RDF to take the final step of the classification, results increased nicely. We see that in AlexNet the global accuracy increases a 18% to get a global accuracy of 76%. We reason that this is a result of freeing the CNN of taking the final decision. In the CNN the last decision is made getting the higher result for the Top 1. But what happens when a second class is almost as high as the one with the highest result? In the CNN with a Top 1 accuracy that does not matter due that the CNN will only take the best one. This phenomenon is mitigated when using Top 2 or even Top 3 solutions. As can be seen in Figure 2.12 Top 2 results increases the accuracy around a 10%. But a Top 2 is not one answer but two. Here is where adding a second system to work with in that *shaded* region makes sense. We suspect that the RDF is able to decide that a class with a lower probability, but close to the best one, could be the correct one. We see in confusion matrices B.3 and B.4 that results are more regular and only *Biking* and *Cooking* classes have high rates of miss-classifications. And we can also see in Table 2.5 that results are now more regular along classes. When we had a 5% of accuracy for *Biking* in AlexNet now we have a 68% of accuracy.

Once we took this path, we tried, and indeed we achieved, to get even better results. This time rather than training just with the probabilities, we added more information. This is why we finally used our trained CNN's as a feature extractor to feed the RDF which is the one taking the responsibility of making the final decision. Mixing the features allow us to reach the highest results even though some classes still present some miss-classification tendencies as represented in confusion matrices B.6, B.5 and B.7. In Table B.1 we see that AlexNet + RDF of (fc6, probs) features gets a 100% accuracy for *Driving* and a 99% of accuracy for *Resting*. GoogLeNet does not get 100% in any class but 99% in *Driving* and *Resting*.

As expected, we got the better results in classes where we have more examples like *Driving, Working, Mobile, Plane* but also when they do not overlap each other as in *Meeting, Socializing, Talking* or *Walking indoor, Shopping*. Even though the overlapping problem seems to be mitigated mixing CNN and RDF's it is not for the classes with a low number of examples like *Cooking* and *Biking*.

Conclusions

This project has walked through from a practical work of building the Annotation tool to a more *research related* work in which a lot of experiments were done.

In the first part of this project, different aspects of software development were put in practice. The Annotation Tool uses and mixes some libraries, such as Flask and Angular, that are known to be the best solutions of its areas. But working with new technologies beyond the general scope, is always a long process, first studying such technologies and, and after that, a strenuous phase of *trial and error*. That is why getting the best of different tools, and combine them, is always a hard task to do. The Annotation tool, eventually, emerges as a powerful tool that fulfills its main purpose. Thus, labeling pictures with this tool can be done in an easy and fast manner. It also provides extra tools that can be used intuitively. But the most important is that it has proven to be a good foundation for projects that require similar characteristics regarding working with pictures.

On the other hand, the second part required long hours of experimental work. The automatic classifier built only with a CNN showed that such *state-of-the-art* algorithms are powerful tools that can achieve great results to problems that did not have solutions years ago. However, during this project we have seen that combining new solutions with other well known approaches, as RDF's are, those results are capable of being improved. Thus, the combination of a RDF and CNN features outperforms a classical CNN approach with a global accuracy rate of 85%.

This project has ended with a manual classification tool and an automatic classifier algorithm. It also provides a big dataset of around 25.000 classified images which allows people to go deeper on this area. Thus, we could say that the goals have been fulfilled.

As far as I am concerned, in hindsight, the first part of this project helped me gaining confidence providing solutions to problems with limited tools. Moreover, the second part, showed me what a research project is about: sometimes your are not sure if you are doing it right or doing it wrong, and there is no much information out to light up the path you are walking through.

References

[1] Angularjs - superheroic javascript mvw framework. URL https://angularjs.org/.

[2] Bootstrap - the world's most popular mobile-first and responsive front-end framework. URL http://getbootstrap.com/.

[3] Flask-sqlalchemy, . URL http://flask-sqlalchemy.pocoo.org/.

[4] Flask-wtf, . URL https://flask-wtf.readthedocs.io/en/latest/.

[5] Sqlalchemy. URL http://www.sqlalchemy.org/.

[6] Flask web development, one drop at a time. URL http://flask.pocoo.org/.

[7] jquery: The write less, do more, javascript library. URL https://jquery.com/.

[8] Sphinx, a python documentation generator. URL http://www.sphinx-doc.org/en/stable/.

[9] Cross-platform c library that implements a self-contained, embeddable, zero-configuration sql database engine. URL https://www.sqlite.org/.

[10] The uwsgi project. URL http://uwsgi-docs.readthedocs.io/en/latest/.

[11] Virtualenv is a tool to create isolated python environments. URL http://docs.python-guide.org/en/latest/dev/virtualenvs//.

[12] D. Castro, S. Hickson, V. Bettadapura, E. Thomaz, G. Abowd, H. Christensen, and I. Essa. Predicting daily activities from egocentric images using deep learning. *ISWC*, 2015.

[13] M. Grinberg. Flask web development:developing web applications with python, 2014. URL http://flaskbook.com/.

[14] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0580, 2012. URL http://arxiv.org/abs/1207.0580.

[15] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. 2012.

[16] M. A. Nielsen. Neural networks and deep learning. 2015.

[17] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[18] Standord. Convolutional neural networks for visual recognition, 2016. URL `http://cs231n.github.io/`. [Online; accessed 1-June-2016].

[19] C. Szegedy, G. Inc, W. Liu, Y. Jia, G. Inc, P. Sermanet, G. Inc, S. Reed, D. Anguelov, G. Inc, D. Erhan, G. Inc, V. Vanhoucke, G. Inc, A. Rabinovich, and G. Inc. Going deeper with convolutions.

[20] Wikipedia. Deep blue (chess computer) — wikipedia, the free encyclopedia, 2016. URL `https://en.wikipedia.org/w/index.php?title=Deep_Blue_(chess_computer)&oldid=721053580`. [Online; accessed 2-June-2016].

[21] Wikipedia. Web server gateway interface — wikipedia, the free encyclopedia, 2016. URL `https://en.wikipedia.org/w/index.php?title=Web_Server_Gateway_Interface&oldid=713921269`. [Online; accessed 1-June-2016].

Appendices

Annotation tool

A.1 Detailed functionalities

In this section we will cover some aspects of the development that are worth to be mentioned.

Multiple selection

Select multiple pictures at a time is achieved using the *shift* key as mentioned in Section 1.3.3. The procedure is the following: When a picture is clicked, we check if the shift key is pressed, if not, we store the index of the clicked picture. When the picture is clicked and the shift key is pressed, we get the new index and substract from the the array of pictures, all the pictures in between those indexes. The next step is to change the state of those pictures based on the state of the last picture clicked.

Listing A.1: Piece of code for multiple selection

```
1  $boxes.click(function(evt) {
2      if(!_lastChecked) {
3          _lastChecked = this;
4          return;
5      }
6      if(evt.shiftKey) { // check if
7          var start = $boxes.index(this),
8              end = $boxes.index(_lastChecked);
9          var boxes = $boxes.slice(
10             Math.min(start, end),
11             Math.max(start, end) + 1);
12         // for each element
13         $.each(boxes, function(key, value){
14             // changes the state
15             $(value).prop('checked', _lastChecked.checked)
16             // call to angular
17             angular.element(value).triggerHandler('click');
18         })
19     }
```

```
20      _lastChecked = this;
21  });
```

In Listing A.1 we see a piece of Javascript code which does the work. This code is wrapped into a jQuery plugin. Note that in the code we are dealing with *checkboxes*[5] instead of pictures, this is the way we internally work in order to be able to associate states to pictures and send that information to the server when is needed as well as changing how they are shown.

Visualizing selected pictures

As said in the previous section, pictures are associated to a checkboxes, internally, when a picture is clicked, the checkbox associated is *checked* or *unchecked*. But we do not see checkboxes, we only see the pictures. This is achieved using CSS techniques.

A checkbox is a type of *input element*[6], which is used to let the user interact with the application. As every html element, we can apply some style, which is the way how elements are shown in a web page, with css.

Listing A.2: CSS piece of code

```
1  /* hides the checkbox */
2  .pictures-list input{display:none;}
3
4  /* applies a style when is checked */
5  .pictures-list input[type=checkbox]:checked + img{
6      border-color:#444;
7      opacity:.8;
8  }
9
10 /* applies a style to the picture when is associated
11 to the label with id 2 */
12 .pictures-list label.picture-label-2{
13     border-color:#9CC8D6;
14     background-color:#9CC8D6;
15 }
```

In this case we have made checkboxes invisible, and added some borders to the pictures, so when a checkbox is in the state checked, the current picture will have a grey border and an opacity effect.

Also, when a picture has a label associated, we modify the style given to the picture applying other border, a wider one, with an specified color. In Listing A.2 we see a piece of the code that makes this visualization possible.

[5]The HTML input element $<$ *inputtype* $=$ *"checkbox"* $>$ [https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input/checkbox]

[6]The HTML element $<$ *input* $>$ [https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input]

Angular and Flask

Angular is a Javascript model-view-controller framework in itself, to build single page applications. In our case we have limited the use of angular to the catalog section.

They way angular work is inserting chunks of snippets inside the html. When the page is loaded, the angular parses those chunks of code into something with some meaning. You can also bind some elements in order to get an event as soon as an element has changed.

Listing A.3: CSS piece of code

```
1 What is your name?
2 <input type="text" ng-model="your_name" />
3
4 <h1>Hello {{ your_name }}</h1>
```

In Listing A.3 we see a simple example, the variable *your_name* is linked in the model using the directive $ng-model = "your\_name"$. Thus, as soon as we interact with the input we will see how the text changes.

Updating pictures

In our application, the model will contain the checkboxes associated to the pictures. When we click a label, another event is fired and a message is sent to the server with the new information. The content of the message is based on the information we have inside the angular model, instead of going through all the pictures of the page and checking its states, the model has already all the information needed. The message is a pretty lighweight JSON due it only contains a list of picture indexes and a label id, as seen in Listing A.4. Once the message is sent to the server, and the server responses OK, we change the state of those checkboxes and update another parameter associated to each picture, called label. Then the picture gets the color of the given label.

Listing A.4: Piece of JSON sent to update the information in the server

```
{
    "keys": ["22899","22900","22901","22902","22903","22904"],
    "label":"1"
}
```

Loading pictures

We also take advantage of angular at the moment we want to load pictures. We said previously that we are using an *infinite scroll* mechanism to load pictures. This mechanism fires an event when the bottom of the page is reached. When we get the new bunch of pictures to be inserted in the page, we do not actually insert them manually. This task is done in conjunction with angularjs. We actually update the model, angular internally detects a change in the model and updates the page consequently. Angular is smart enough to not reload information which was already loaded

Putting everything together

There is a last step to be done. Angular uses {{ *pattern* }} to detect variables inside the html code, but, our template engine uses the same type of symbols. Luckily, angular provide us a directive to change its default behavior and associate a different pattern to detect the variables inside the html code. Thus, we have defined our controller to look for the pattern {*a pattern a*}. The functionality is seen in Listing A.5. If we did not do this, our template engine would think that those chunks must be parsed too.

Listing A.5: Example of how to change Angular symbols

```
1 annonApp.config(['$interpolateProvider',
2     function($interpolateProvider) {
3         $interpolateProvider.startSymbol('{a');
4         $interpolateProvider.endSymbol('a}');
5 }]);
```

In Listing A.6 we see a piece of the catalog template. The directive $ng-repeat =$ "*id in annonLoader.orderedPictures*()" makes that piece of html to be repeated as many times as pictures are in the model. The directive {*a picture.id a*} replaces that code with the current picture id. We also see a {{ *url_for*('*static*', *filename* =' *media*/') }}, but this, instead of being an angular directive, is a piece of the template engine which will be evaluated in the server, not in the client, as the rest of the angular code does.

Listing A.6: Example of the template with angular code

```
1 <li ng-repeat="id in annonLoader.orderedPictures()"
2     ng-init="picture=annonLoader.pictures[id]">
3
4     <label for="check-{a picture.id a}"
5         class="picture-label-{a picture.label a}"
6         title="label {a picture.label a}">
7
8         <input id="check-{a picture.id a}"
9             type="checkbox" ng-change="elemChange()"
10            ng-model="selectedCheckboxes[picture.id]"
11            value="">
12
13        <img ng-src="{{ url_for('static', filename='media/')}}
14            {a picture.path a}" imageonload border="0">
15    </label>
16 </li>
```

A.2 Extra functionalities

After finishing the annotation tool some changes were applied to the annotation tool in order allow the user to insert comments to the pictures. The tool now would show

the catalog the same way but after pointing to a picture, an icon will appear next to the picture. Once the user clicks to the icon *modal-box* will appear as shown in Figure A.1. There, the user will be able to insert a comment regarding to the picture and its label.

This comment will appear also when generating the dataset, inside the `train.txt` file, next to the picture path and its label id.

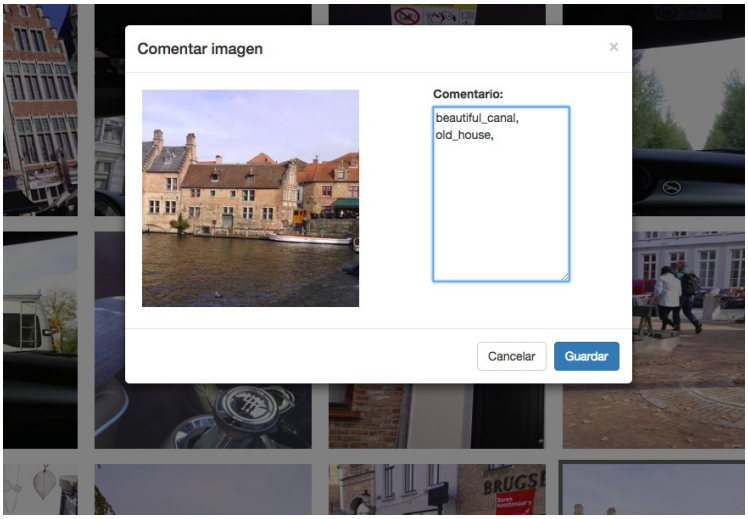

Figure A.1: Second version of the catalog to allow the user inserting comments to pictures.

Activity recognition

A.3 Additional figures and tables

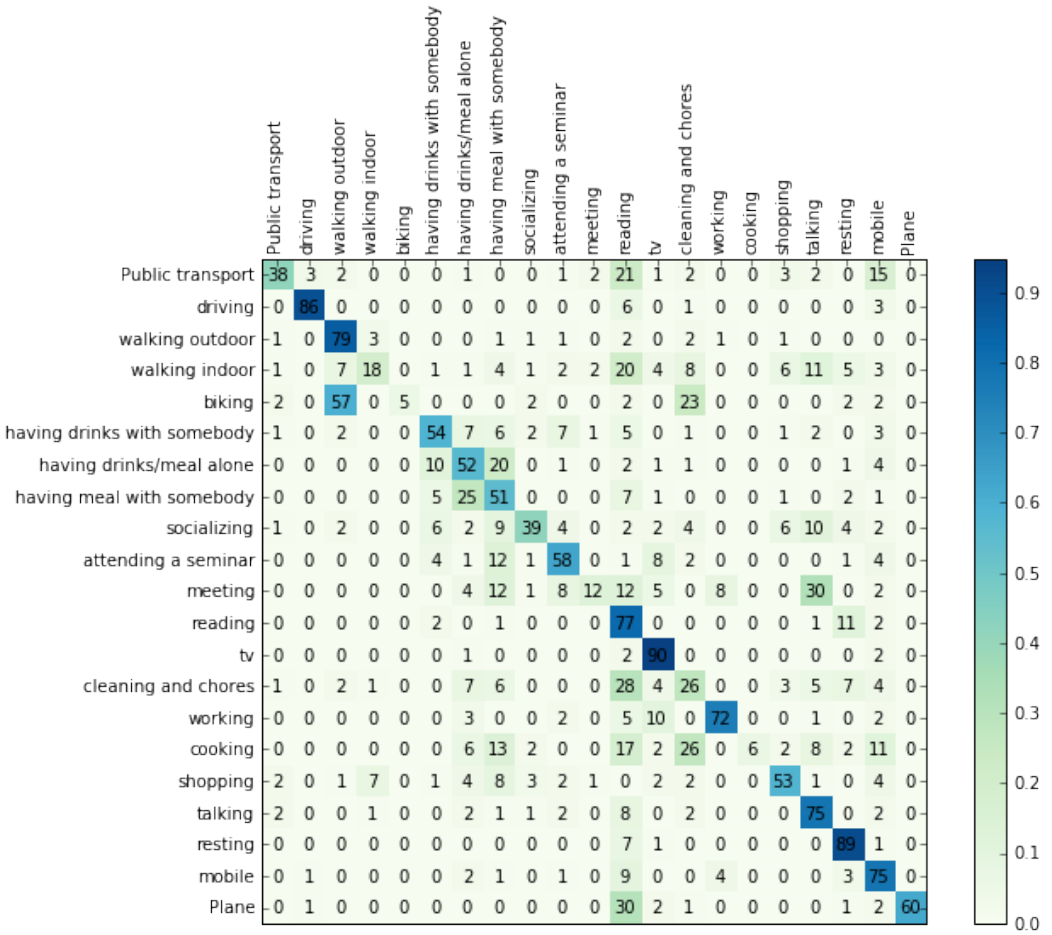

Figure B.1: Confusion matrix for finetuned AlexNet. Columns are the predicted labels and rows the real labels

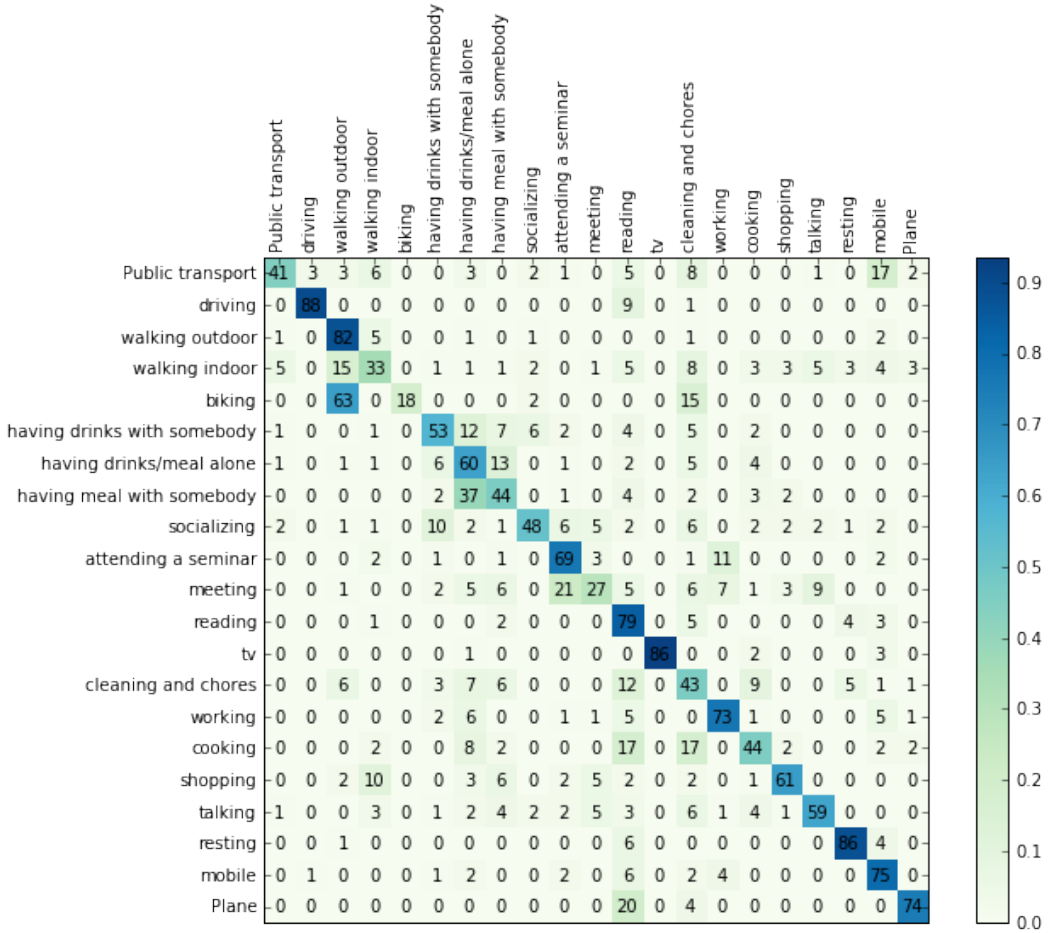

Figure B.2: Confusion matrix for finetuned GoogLeNet. Columns are the predicted labels and rows the real labels

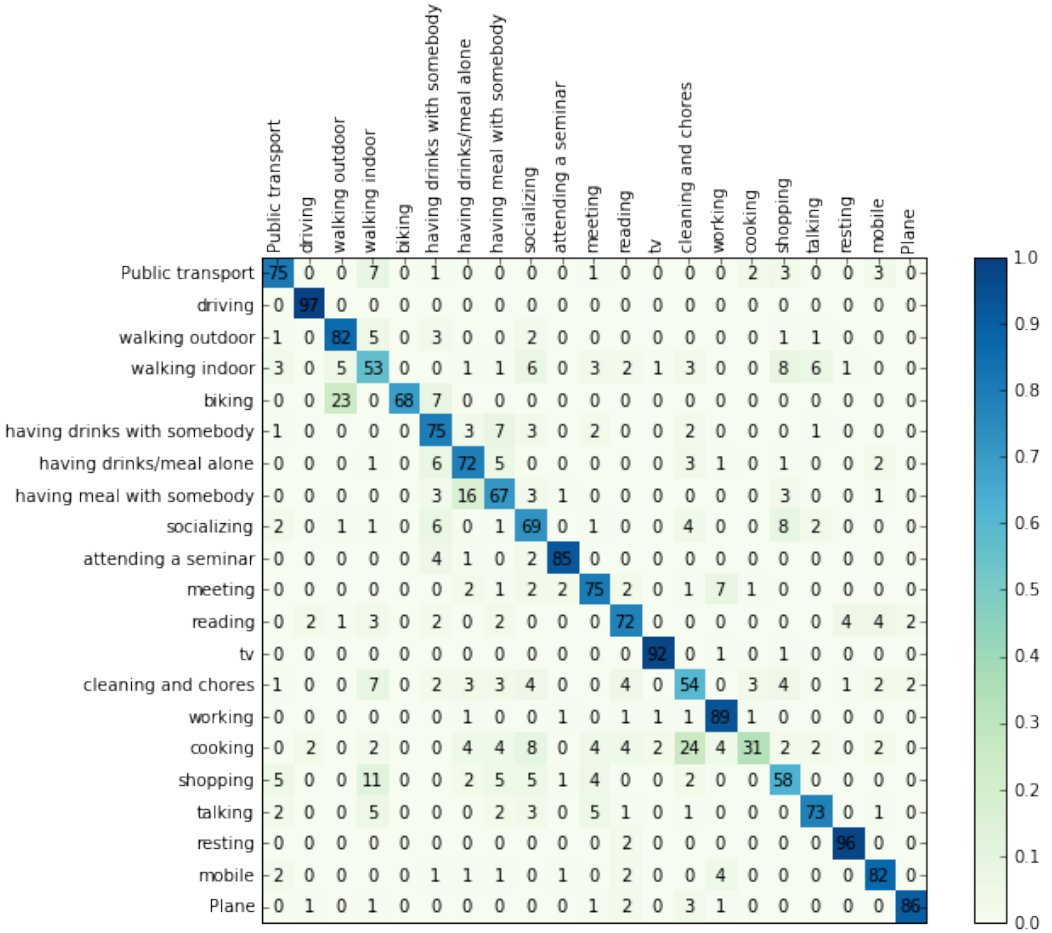

Figure B.3: Confusion matrix for finetuned AlexNet + RDF. Columns are the predicted labels and rows the real labels

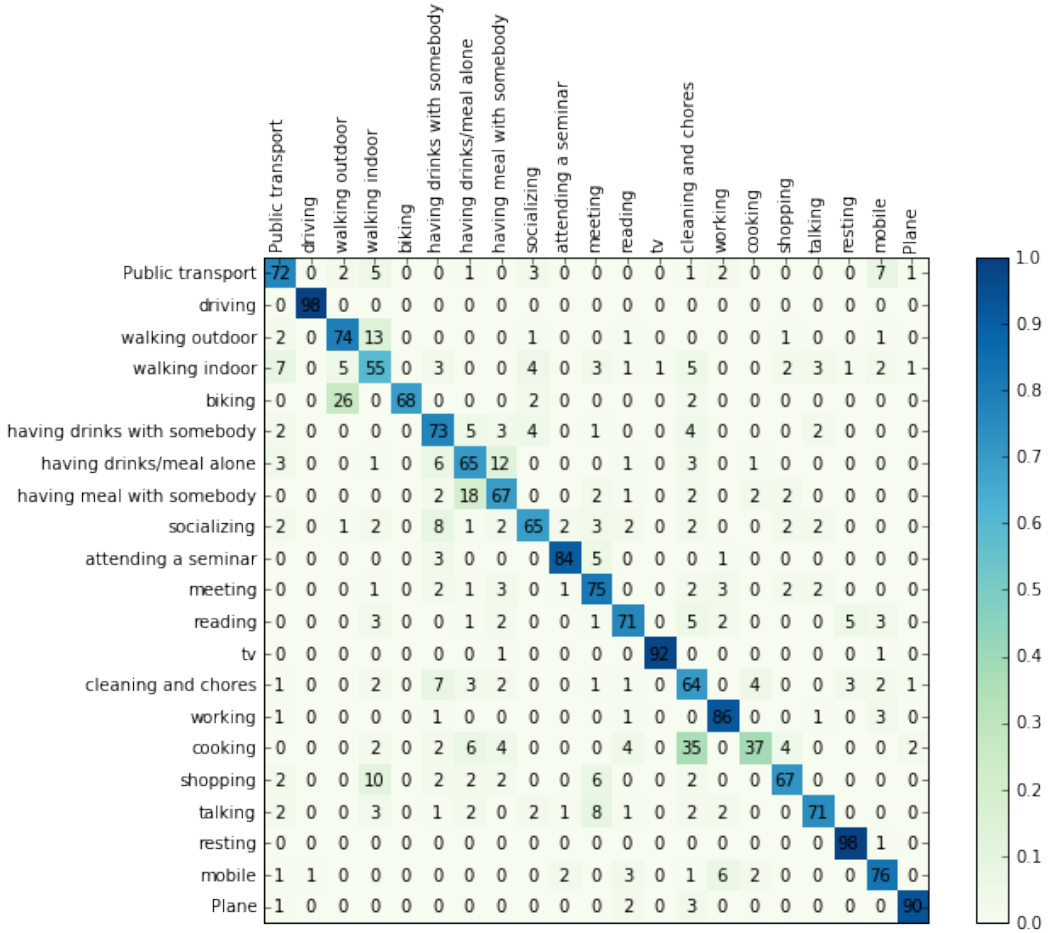

Figure B.4: Confusion matrix for finetuned GoogLeNet + RDF. Columns are the predicted labels and rows the real labels

class	AlexNet+RDF (FC6, probs)	AlexNet+RDF (FC7, probs)	GoogLeNet+RDF (pool5/7x7_s1, probs)
Plane	92%	91%	94%
Mobile	87%	90%	86%
Resting	99%	99%	99%
Talking	83%	87%	84%
Shopping	73%	76%	78%
Cooking	35%	37%	40%
Working	95%	93%	93%
Cleaning and chores	60%	55%	72%
Tv	94%	94%	96%
Reading	82%	83%	83%
Meeting	85%	83%	88%
Attending a seminar	92%	90%	91%
Socializing	85%	82%	78%
Having meal with somebody	86%	84%	83%
Having drinks/meal alone	83%	82%	73%
Having drinks with somebody	90%	88%	86%
Biking	73%	73%	76%
Walking indoor	65%	65%	70%
Walking outdoor	85%	84%	83%
Driving	100%	98%	99%
Public Transport	83%	83%	85%
Global Accuracy	85%	84%	85%

Table B.1: Correlation between classes and accuracies for different configurations.

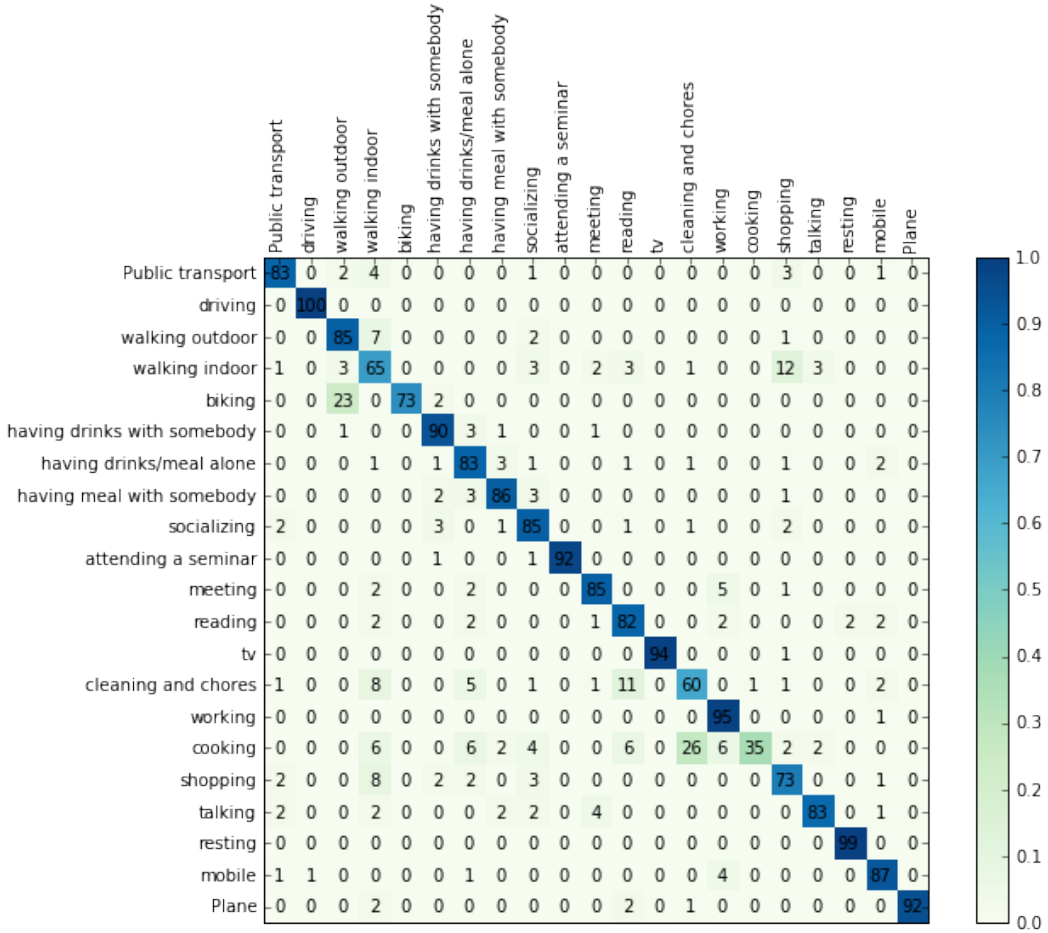

Figure B.5: Confusion matrix for finetuned AlexNet + RDF for layers FC6 and probs. Columns are the predicted labels and rows the real labels

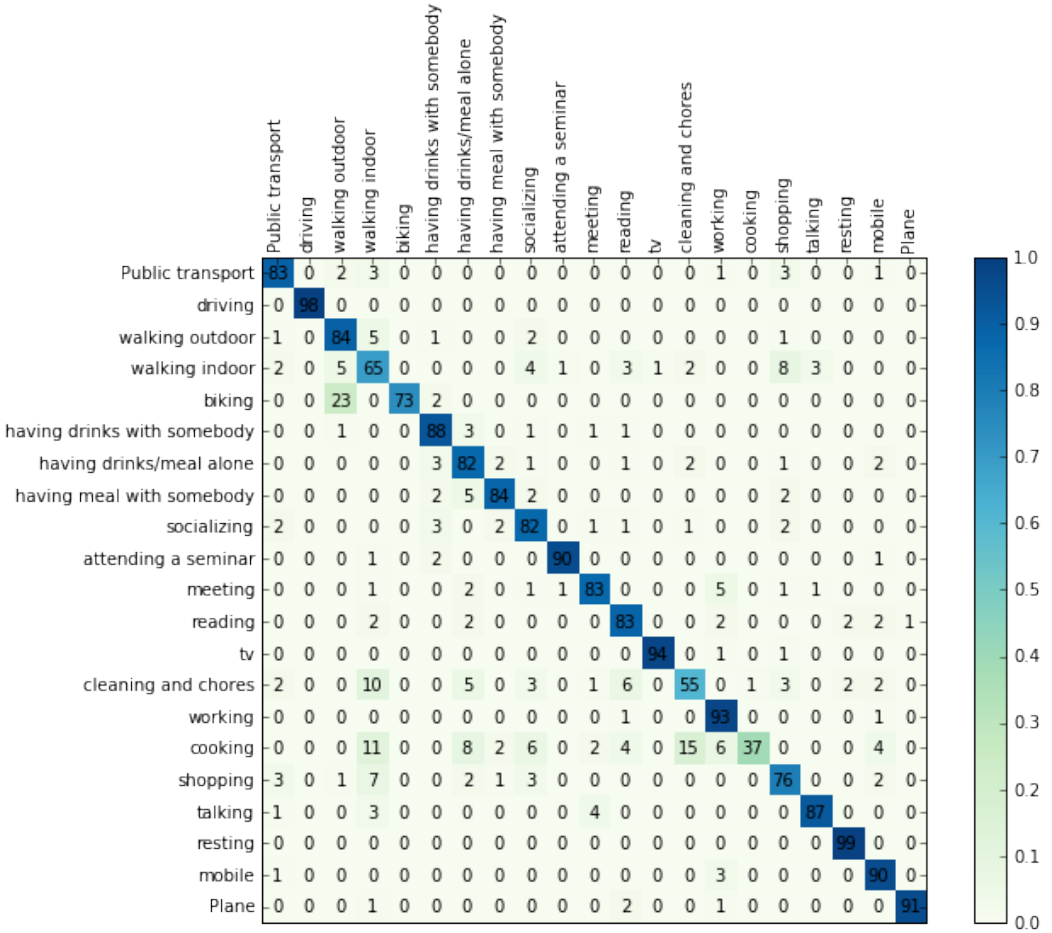

Figure B.6: Confusion matrix for finetuned AlexNet + RDF for layers FC7 and probs. Columns are the predicted labels and rows the real labels

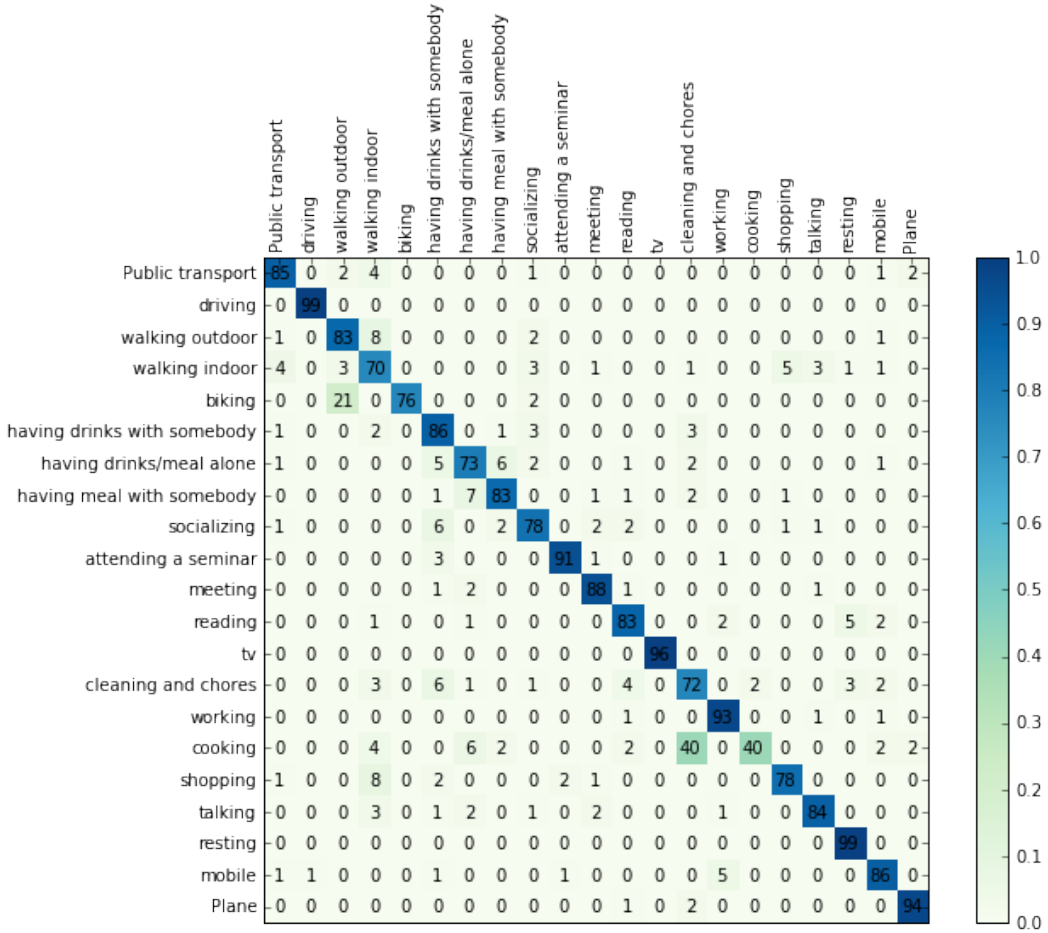

Figure B.7: Confusion matrix for finetuned GoogLeNet + RDF for layers Pool5 and probs. Columns are the predicted labels and rows the real labels