UNIVERSITÀ DI PISA

University of Pisa

Laurea Magistrale (MSc) in Artificial Intelligence and Data Engineering

Project

Multimedia Information Retrieval and Computer Vision

LSH + MobileNet + Mushrooms Dataset

Academic year 2021-2022

Valerio Giannini, Alessio Serra, Giulio Federico, Leo Maltese, Marco Simoni

Google Drive:
https://drive.google.com/drive/folders/1YQwJXaiCoQlOFRw3RxYDGZFGI7bpItl_?usp=sharing

Introduction

The objective of this project is to verify that a particular pre-trained neural network, in our case the MobileNet, is able to classify a set of images for which was not trained before. To achieve our goal, we fine-tuned the network, re-trained some layer of the pre-trained model, and we test its robustness with the introduction of a distractor, a set of images which are not related with our dataset. Moreover, to perform the matching between the feature extracted with our network, we implemented the LSH index in both versions, projection and binary version, in order to speed up the matching process.
After training the network, having chosen the best model after a series of tests, and creating different LSH index with different parameters, we did a several test in order to compare different results using different configurations of the LSH index and the MobileNet fine-tuned/not fine-tuned.
At the end, we implemented a very simple web application to see the behaviour of the entire system in friendly way.
To summarize, following there are some details about the principal components that we used in our project:

- **MobileNet neural network**: MobileNet is a class of CNN that was open sourced by Google and it is designed to be used in mobile applications. There are 3 versions of the MobileNet: MobileNet, MobileNet_v2, MobileNet_v3 (Small and Large), with respectively 86, 154, 234 (Small) and 269 (Large) layers.

- **Mushrooms Dataset**: this dataset is composed by 6714 images divided in 9 classes and each class represents a specific typology of mushroom. There is not balancing among the classes (we will analyse in the proper way this problem)

- **Distractor**: it is a dataset, named “mirflickr25k”, containing 25,000 images related to different concepts and subjects (animal, people, objects, etc.)

- **LSH index**: locality-sensitive hashing (LSH) is an algorithmic technique that hashes similar input items into the same "buckets" with high probability (the number of buckets is much smaller than the universe of possible input items). Since similar items end up in the same buckets, this technique can be used for nearest neighbour search and for others purpose.

Feature Extraction

To extract image features were used networks obtained applying transfer learning technique starting from pre-trained networks that belongs to the MobileNet-family, namely *Mobilenet, Mobilenet_v2* and *Mobilenet_v3_large*.

These three networks have different characteristics and architectures, which differ mainly in the number of layers and parameters, as can be seen in *Table 1.*

Models	Parameters	Layers
Mobilenet	3.2M	86
Mobilenet_v2	2.3M	154
Mobilenet_v3	4.2M	269

Table 1: Statistics of mobilenet-family model used.

Features have been extracted using three different approaches:

- From *freezed* pre-trained model, just using the output of the pre-trained network.
- From the penultimate layer of a model with a MLP on top of the *freezed* pre-trained model.
- From the penultimate layer of a model with a MLP on top of the *fine-tuned* pre-trained model.

Network Training

All the base networks of the Mobilenet family have been pretrained on *ImageNet*, that contains 1.4 million labeled images and 1000 different classes, and are taken from Tensorflow Hub, which also provides a preprocessing function to transform image into a suitable form for the model, which supports images of any input size greater than 32 x 32, even if with larger sizes better performances are obtained.

Different configuration of each network were considered in order to improve the network's ability to distinguish classes of mushrooms, since in this way the extracted features represent in a more meaningful way input images.

In particular were tested the effects of *Data Augmentation* to counteract the imbalance of the training set, different Multilayer perceptron, modifying the number of layers and of parameters, and also different techniques to avoid overfitting.

MobileNet Performance and Improvement during the Training Phase

In this section we analyse, in a specific manner, the performance of the MobileNet V1, the base version of the net, which is composed by 86 layers. In particular, we analysed which is the behaviour of the network when we try to train it with a dataset composed by mushrooms which have 9 different classes. Moreover, we tried to achieve the best configuration for the training of the MobileNet.

About the Dataset assigned, we can see the distribution of the classes in the following image (Figure 1):

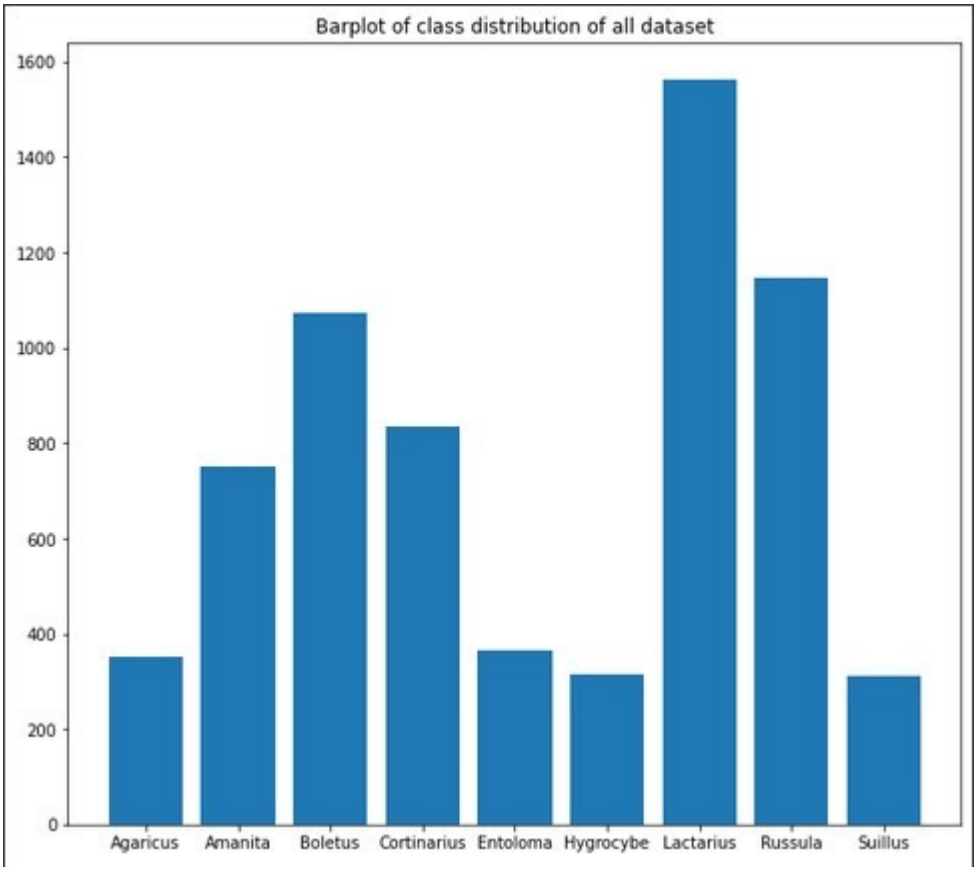

Figure 2: Distribution of the classes

How we can see, the dataset is unbalanced, in particular there are 4 classes that have very few samples respect the majority class, and, moreover, the dimension of the dataset is quite small. Starting form this first analysis, we tried different configuration in order to train the MobileNet in a better way. Following are described the main configuration that we used to train the MobileNet, in particular we report the results of the training with **a MLP Classifier** on top of the freezed model and, if the results of the previous model are reasonable, also with the **Fine Tuning** with MLP Classifier. We will report also other parameters when the results of the training are acceptable.

Configuration 1

In this first configuration we tried to train the MobileNet without any particular modification, so we trained the net with a simple MLP classifier on top of the pre-trained model, below the results:

Principal parameters MLP:

- Layer 1: 512 neurons
- Layer 2: 9 neurons (number of classes)
- Learning rate: 0.0001
- Dropout: 0.2

Results MLP Classifier				
Loss	**Validation loss**	**Accuracy training**	**Accuracy validation**	**F1-Macro**
0.2995	0.6361	0.9117	0.7913	0.7803

With this configuration, as was to be expected, the net tends to overfit. This is because the dataset, as we have seen in previous paragraph, is unbalanced and we have quite few images in our dataset. In conclusion this configuration was **discarded**.

Configuration 2

In second configuration, considering the previous results, we added a data augmentation during the training phase. In this way tried to solve the overfitting problem and, at the same time, "create" a sort of new training data from existing training set. Below the results:

Principal parameters MLP:

- Layer 1: 512 neurons
- Layer 2: 9 neurons (number of classes)
- Learning rate: 0.0001
- Dropout: 0.5

Result MLP Classifier				
Loss	**Validation loss**	**Accuracy training**	**Accuracy validation**	**F1-Macro**
0.7646	0.7447	0.7328	0.7410	0.7481

Principal parameters MLP Keras Tuner:

- Layer 1: 2048 neurons
- Layer 2: 9 neurons (number of classes)
- Learning rate: 0.001
- Dropout: 0.5

Result MLP Classifier Keras Tuner				
Loss	**Validation loss**	**Accuracy training**	**Accuracy validation**	**F1-Macro**
0.6566	0.7018	0.7656	0.7703	0.7713

Principal parameters Fine tuning:

- Layers not trainable: 50
- Learning rate: 0.0001
- Other parameters are the same of MPL Keras Tuner

Result Fine Tuning				
Loss	**Validation loss**	**Accuracy training**	**Accuracy validation**	**F1-Macro**
0.6492	0.6432	0.7700	0.7958	0.7721

This solution eliminates the problem of the overfitting but, how can see from the results obtained, we don't achieve very high result in terms of accuracy. So this configuration can be acceptable but we don't have very high performance.

Configuration 3

Since in the previous configuration we did not obtain very high performance, we balanced the dataset in order to test the performance of the MobileNet with a balanced dataset. We applied data augmentation on the training set in a static way, so we augment it increasing the minority classes applying some transformation on the image (rotation, zoom, etc.). In this way we have a perfect balanced dataset, Figure 2, and with this dataset we trained the MobileNet removing the block of data augmentation during the training phase inserted in previous configuration, below the results:

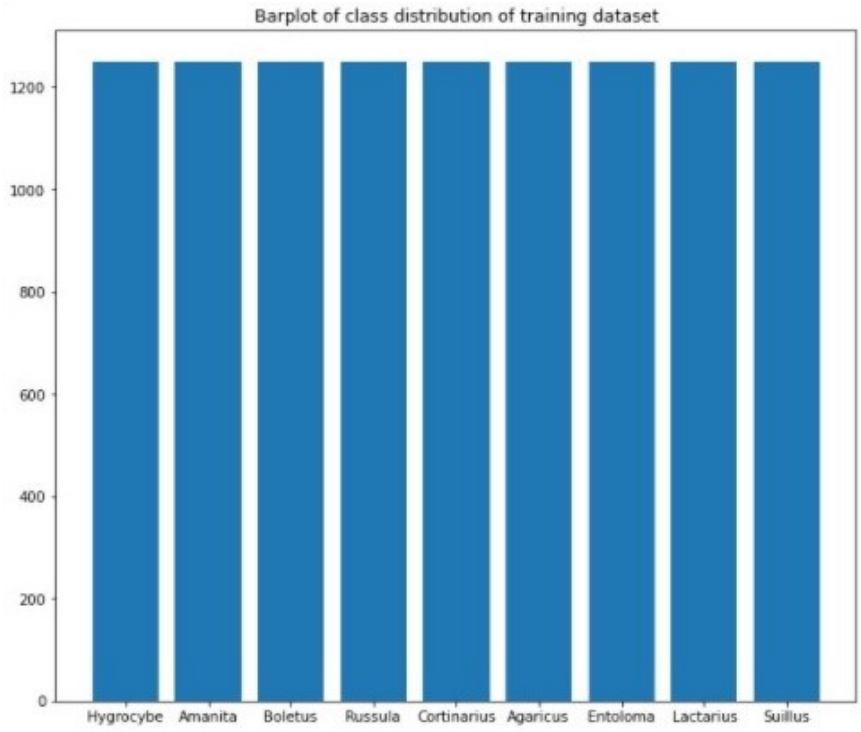

Figure 3: Balanced Training set

Principal parameters MLP:

- Layer 1: 512 neurons

- Layer 2: 9 neurons (number of classes)
- Learning rate: 0.0001
- Dropout: 0.5

Result MLP Classifier				
Loss	**Validation loss**	**Accuracy training**	**Accuracy validation**	**F1-Macro**
0.2046	0.6640	0.9528	0.7718	0.7691

Even in this case we had the problem of the overfitting, probably because, respect the minority classes, the images of the majority classes had very few transformations during the balancing and so net learn always on the same images. Even this model is **discarded**.

Configuration 4

In this final configuration, in order to remove the problem that we had in the configuration 3, we reintroduce again the data augmentation block during the training phase, in order to transform the images of the majority classes in the dynamic way, so the network at each epoch see different images of the same class. Below the result:

Principal parameters MLP:

- Layer 1: 512 neurons
- Layer 2: 9 neurons (number of classes)
- Learning rate: 0.0001
- Dropout: 0.5

Result MLP Classifier				
Loss	**Validation loss**	**Accuracy training**	**Accuracy validation**	**F1-Macro**
0.6111	0.6942	0.7820	0.7568	0.7267

Principal parameters MLP Keras Tuner:

- Layer 1: 2048 neurons
- Layer 2: 9 neurons (number of classes)
- Learning rate: 0.0001
- Dropout: 0.5

Result MLP Classifier Keras Tuner				
Loss	**Validation loss**	**Accuracy training**	**Accuracy validation**	**F1-Macro**
0.5006	0.6227	0.8260	0.7853	0.7581

Principal parameters Fine Tuning:

- Layers not trainable: 50
- Learning rate: 0.0001
- Other parameters are the same of MPL Keras Tuner

Result Fine Tuning				
Loss	**Validation loss**	**Accuracy training**	**Accuracy validation**	**F1-Macro**
0.3711	0.4539	0.8705	0.8634	0.8447

This configuration mitigates the overfitting problem and at the same time improve the performance of the network respect the second configuration (it can be visible in the test with Fine Tuning). So this result to be the best configuration for train the MobileNet.

At the end we achieve our objective, so we obtain a good configuration both for the network and with s balanced training set. One consideration that we must be done is that we applied 2 times data augmentation, one time in a static way, increasing the images of training set for the minority classes, and another time during the training of the network. This can bring us to think that the images could be so much transformed when they are in input of the MobileNet. To avoid this fact, we set the parameters for the transformation of the images, both during the balancing and the training phase, in a proper way to do not change completely the images when they are in input of the network.

HYPERPARAMETER TUNING WITH HYPERBAND KERAS TUNER

To find the best hyperparameters for our network we used the HyperBand Keras Tuner, introduced by (Li et al, 2018) in order to carry out random search optimizing the budget at our disposal, i.e. the training time.

First it is defined a search space, which depends on the hyperparameter to be tested and their possible values, in particular we have a 3D search space to tune:

- *Number of MLP layers,* searching in the range [1, 3].
- *Number of neurons for Dense MLP layers,* searching in the range [512, 2048] with a step of 384.
- *Learning rate,* searching among [1e-3, 1e-4].

After the space definition, the Hyperband runs the *SuccessiveHalving* algorithm considering a budget B of 50 and a reduction factor η equal to 3.

With this algorithm at each iteration are tried a number of configuration, sampled from the search space, equal to the budget chosen divided by η. Then the latter is reduced by η, just keeping the best network configurations and further training them for a longer number of epochs, the sum of which is always equal to the budget.

This process is repeated until just one configuration survives, thus obtaining an optimal configuration, exploring many possibilities through a good compromise between *exploration and exploitation.*

All the models were trained with a batch size equal to 32 and for at most 50 epochs, unless the validation loss did not improve for five consecutive epochs, in fact in that case early stopping interrupts the training to avoid overfitting. Moreover, as optimizer was used Adam and as loss the sparse categorical cross entropy.

To evaluate the network performance has not been used only the accuracy, in fact, even if we suppose to have the same misclassification cost for all the classes, we want to verify the performance also on minority ones. For this reason we have chose also the F1 on single classes, that corresponds to the harmonic mean of precision and recall, and the F1-macro to have a more compact index of performance.
The F1-micro in our case would be useless since that in a single label multi-class classification it is equal to the accuracy.

The best hyperparameter found are reported in *Table 2.*

MODEL	#Layers	#Neurons	learning rate
v1	2	2048	1e-4
v2	2	2048	1e-3
v3	1	512	1e-4

Table 4: Optimal Parameters found with Keras Tuner.

The corresponding results obtained are reported in *Table 3* and *Table 4*, where can be noticed that the model which can extract the most effective features is *mobilenet_v3_large,* that can reach the 80% of accuracy, even if those features are given in input to the shallowest MLP.

MODEL	accuracy	F1-macro
v1_freezed.h5	0.765766	0.733563
v2_freezed.h5	0.695195	0.671047
v3_freezed.h5	**0.803303**	**0.780891**

Table 5: Summary statistics on freezed model with optimal parameters.

MODEL	Agaricus	Suillus	Amanita	Russula	Cortinarius	Boletus	Entoloma	Hygrocybe	Lactarius
v1_freezed.h5	**0.705882**	**0.782609**	0.910714	0.671329	0.548387	0.857143	0.769697	0.756303	0.600000
v2_freezed.h5	0.592593	0.770270	0.871795	0.625000	0.515152	0.821429	0.638158	0.671698	0.533333
v3_freezed.h5	0.686567	0.764331	**0.927273**	**0.766234**	**0.620690**	**0.931034**	**0.818750**	**0.782979**	**0.730159**

Table 6: Statistics on single classes of freezed model with optimal parameters.

FINE TUNING THE PRE-TRAINED MODEL

To increase performance we have also fine tuned part of the pre-trained networks to adapt their weights to our mushrooms classification task.

Starting from the model trained with the optimal hyperparameters found with the Keras Tuner, we have further trained the base networks considering different "*cutting*" levels, i.e. the point below which the layers are still freezed.

To determine the optimal cutting level, where considered four possibilities for each network, that corresponds to the 4/8, 5/8, 6/8 and 7/8 of the total number of layers.

The learning rate used has been reduced by a factor of ten, as we want to limit the magnitude of the modifications we make to the representations of the layers that we are fine-tuning, since that we could harm these representation.

In *Table 6* and *Table 7* are presented the results obtained using the *mobilenet_v2*, in which we can notice that the best number of layers is the lowest with which we can reach 87% of accuracy, that is much larger than the 70% we have achieved with the freezed model.

Also in the other models we have obtained as the best number of layers the lowest ones, thus demonstrating that higher level representation can be modelled worse using the pre-trained weights.

Another point in common with all the networks is that there is not a monotonic decrease in performance when increasing the number of freezed layers, in fact as shown in *Table 5,* the *fine_tuned_115* performs better than *fine_tuned_96*.

MODEL	accuracy	F1-macro
fine_tuned_115.h5	0.851351	0.837905
fine_tuned_135.h5	0.839339	0.821062
fine_tuned_96.h5	0.834835	0.831162
v2_fine_tuned_77.h5	**0.872372**	**0.854069**

Table 7: Summary statistics of Mobilenet_v2 fine tuned with different cutting levels.

MODEL	Agaricus	Suillus	Amanita	Russula	Cortinarius	Boletus	Entoloma	Hygrocybe	Lactarius
fine_tuned_115.h5	**0.837838**	0.865672	0.953704	0.813953	0.677419	0.918033	0.841121	0.839286	0.794118
fine_tuned_135.h5	0.727273	0.828571	0.962264	0.819277	0.709677	0.900000	0.834356	0.830357	0.777778
fine_tuned_96.h5	0.757576	0.840764	**0.971698**	**0.845238**	**0.757576**	0.909091	0.785714	0.787402	**0.825397**
v2_fine_tuned_77.h5	0.818182	**0.870748**	0.942222	0.828402	0.746269	**0.950820**	**0.883117**	**0.884956**	0.761905

Table 6: Statistics on single classes of Mobilenet_v2 fine tuned with different cutting levels.

We have also tried to fine tune lower layers, but performances get worse, in fact earlier layers in the convolutional base encode more generic, reusable features, while layers higher up encode more specialized features. And it is more useful to fine-tune the more specialized features, as these are the ones that need to be repurposed on our new problem.

FINAL RESULTS ON TEST SET OF ALL MODELS

We have selected six best models according to the results obtained on the validation set and finally we have tested their generalization capabilities, obtaining as best overall model the *mobilenet_3_fine_tuned* with almost the 90% of accuracy, that has also the best performances on single classes with the 88% of F1-macro.

MODEL	accuracy	F1-macro
v1_fine_tuned_43.h5	0.835052	0.827178
v1_freezed.h5	0.759941	0.742643
v2_fine_tuned_77.h5	0.873343	0.859440
v2_freezed.h5	0.706922	0.662103
v3_fine_tuned_135.h5	**0.896907**	**0.883415**
v3_freezed.h5	0.848306	0.833715

Table 8: Summary tatistics of best models using test set.

MODEL	Agaricus	Suillus	Amanita	Russula	Cortinarius	Boletus	Entoloma	Hygrocybe	Lactarius
v1_fine_tuned_43.h5	0.753623	0.859060	0.927273	0.836364	0.750000	**0.898551**	0.811688	0.801587	**0.806452**
v1_freezed.h5	0.646154	0.797386	0.858333	0.720000	0.718750	0.870968	0.743202	0.736402	0.592593
v2_fine_tuned_77.h5	**0.835821**	0.885906	0.937500	0.864198	0.835821	0.866667	0.873418	0.864198	0.771429
v2_freezed.h5	0.464286	0.818792	0.854701	0.625000	0.571429	0.727273	0.668874	0.728571	0.500000
v3_fine_tuned_135.h5	0.828571	**0.932432**	**0.959641**	**0.892473**	**0.888889**	0.885246	**0.885135**	**0.885246**	0.793103
v3_freezed.h5	0.794118	0.860927	0.915556	0.797297	0.769231	0.866667	0.841791	0.864198	0.793651

Table 9: Statistics on single classes of best model using test set.

LSH

Locality Sensitive Hashing has been implemented endowing a python object of 2 peculiar modalities:

- random projection (projection)

- random position (binary)

The complete implementation of the index and the consecutive test on queries, could be carried out performing the following steps:

1. Parameters Initialization

First of all, the index is initialized setting a variable number of parameters which allow us to specify, among the other things, the modality of the index and the total amount of features that are present in the descritptors. The complete list of parameters that can be set is shown in the following snippet.

```
lsh = LSH(descriptor_dim=dataset.shape[1], mode='projection', num_g=10, num_h=1,
          root='/content/drive/MyDrive/MIRCV_Group/indexes', sub_root='index_1')
lsh.add(dataset)

lsh = LSH(descriptor_dim=dataset.shape[1], mode='binary', num_g=8, num_h=9,
          root='/content/drive/MyDrive/MIRCV_Group/indexes', sub_root='index_1')
lsh.add(dataset)
```

Parameters

- *descriptor_dim:* int --> The dimension of the object descriptors to be indexes
- *mode* : {'projection', 'binary'}, default='projection'
  The type of hash functions to be built.
  - If 'projection', random projections are used
  - If 'binary', random positions are used
- *num_g:* int, default=1 --> The number of g hash functions used
- *num_h*: int, default=1 --> The number of h hash functions used for each g
- *w*: int, default=4 --> If mode == 'projection', it is the size of the segments in the projection vectors *
- *root*: str, default='' --> Root for the directory where to save the index
- *random_state* : int or None, default=None --> Random seed initializer
- *verbose* : bool, default=False --> Controls the verbosity
- overwrite_hash:bool, default=True --> Re-Use of an already set of generated hash functions
- overwrite_index:bool, default=True --> Re-Use of an already built index
- in_memory:bool, default=False --> Query Operation carried out in memory

* The parameter 'w' concerns just the Projection Mode.

Once each parameter has been defined, the following phase consists in the generation of the hash functions 'g'; this operation has been implemented through the usage of the function shown below. The main purpose

of the following method is to arrange and stacking the whole set of g functions that are necessary for the construction of the index.

```
def _init_hash_functions(self):
  """ Initialize g hash functions used to calculate the hashes

  if 'self.overwrite_hash' is True, save the functions to the specified file.

  if 'self.overwrite_hash' is False, load the func from the specified file.
  """
  # # # file_exists = os.path.exists(self.matrices_filename)
  if not self.overwrite_hash:
    # load from file
    if self.binary_mode:
      npzfile = np.load(self.hash_filename)
      self.g_functions = npzfile['gs']
      self.num_g = self.g_functions.shape[0]
      self.num_h = self.g_functions.shape[1]
    else:
      npzfile = np.load(self.hash_filename)
      Xs = npzfile['Xs']
      bs = npzfile['bs']
      self.g_functions = (Xs, bs)
      self.num_g = self.g_functions[0].shape[0]
      self.num_h = self.g_functions[0].shape[1]

  else:
    if self.binary_mode:
      h_functions = [self._generate_g_function() for i in range(0, self.num_g)]
      self.g_functions = np.stack(h_functions)
```

```
    # save to file
    np.savez(self.hash_filename, gs=self.g_functions)
  else:
    Xs = []
    bs = []
    for i in range(0, self.num_g):
      X, b = self._generate_g_function()
      Xs.append(X)
      bs.append(b)

    self.g_functions = (np.stack(Xs), np.stack(bs))

    # save to file
    np.savez(self.hash_filename, Xs=self.g_functions[0],
             bs=self.g_functions[1])
```

As we can notice, both the two available modes leverage an external method called '_generate_g_function', which is needed for filling each single g function with a defined number of h functions.

```
def _generate_g_function(self):
  """
  Generate a g hash function as a composition of num_h h functions

  Return
  ----------
    h_functions: numpy.ndarray
      array of h hash functions generated based on operating mode

  """

  if self.binary_mode:
    # Sample num_h indices between 0 and descriptor_dim without replacement
    h_functions = self.rnd.choice(range(0, self.descriptor_dim),
                                  self.num_h, replace=False)
    h_functions.sort()
  else:
    # Generate X = (x-1, ..., x-num_h) matrix composed of random
    # vectors from a standard normal distribution. Each vector represents
    # a random projection plane
    X = self.rnd.randn(self.num_h, self.descriptor_dim)

    # Generate b = (b-1, ..., b-num_h) vector of random scalars
    # from a uniform distribution
    b = self.rnd.uniform(0, self.w, self.num_h)

    h_functions = (X, b)

  return h_functions
```

2. Construction of the index

Once the hash functions are ready to be used, it is time to build the index, associating objects to the buckets that has been produced by the generation of the hash functions.
Thus, the object has been endowed of a method called 'Add', through which the insertion of the elements of the dataset is guaranteed.
The following is an illustrative figure that anticipates the internal structure of the Add method; for a better explanability, the code has been partitioned in two different parts.

Add

```
def add(self, dataset):
  """
  Hash num_points descriptors of a dataset and insert them into num_g buckets.

  Parameters
  ----------
  dataset: numpy.ndarray
    Array of object descriptors of shape (num_points, descriptor_dim)

  """
  self.dataset = dataset
  if self.binary_mode:
    self.mean = self.dataset.mean(axis = 0)

  if self.overwrite_index:
    # Empty index directory
    shutil.rmtree(self.index_dir)
    self.index_dir.mkdir(parents=True, exist_ok=False)
  else:
    return

  index = [dict() for i in range(self.num_g)]
  for obj_id in tqdm(range(self.dataset.shape[0])):
    obj_descriptor = self.dataset[obj_id]
    buckets = self._hash(obj_descriptor)
    for i, bucket in enumerate(buckets):
      index[i].setdefault(bucket, []).append(obj_id)
```

The dataset is passed as argument of the method and in the case in which the mode is 'Binary', then we compute the features' mean all over the whole dataset in order to make possible the future binarization of elements' descriptors.
The main goal of the function is to assign each component of the dataset to the corresponding bucket through the usage of the function _hash, which will be analyzed later.

HASH

The following method is the one charged for hashing descriptors; in fact, in the parameters section, we can spot the argument 'obj_descriptor'.

```
def _hash(self, obj_descriptor):
  """
  Parameters
  ----------
  obj_descriptor: numpy.ndarray of shape (descriptor_dim,)
    Point p to be inserted in the index

  Return
  ----------
  string_buckets: numpy.ndarray of shape (num_g,), each element is a str
    - IF mode == 'projection':
        Formatted buckets as g-i(p) = '[h-i-1(p). ... .h-i-num_h(p).]'
    - IF mode == 'binary':
        Formatted buckets as g-i(p) = '[P{h-i-1(p)}. ... .P{h-i-num_h(p)}.]'

        P --> Powers of 2

  """
```

Obviously, in this case, we need to carefully separate the operations that have to be carried out for the binary mode with respect to the ones that have to be implemented for the standard version.
The Binary version is handled leveraging the function called 'Binarizer', whose main purpose is to binarize the descriptor argument against the mean value of the complete dataset. After that, the components of the descriptor defined by the h functions are selected.
The final step consists in the transformation of the binary value, obtained in the previous phase, in the corresponding two base version and save the analyzed element into the bucket whose id will be the string with a name equal to the computed base two version.

```
if self.binary_mode:

  def binarizer(descriptor):
    upper, lower = 1, 0
    binary = np.where(descriptor > self.mean, upper, lower)
    return binary

  powers_of_two = 1 << np.arange(self.num_h - 1, -1, step=-1)
  binarized_object = binarizer(obj_descriptor)
  binary_buckets = []
  for function in self.g_functions:
    binary_buckets.append(binarized_object[function])

  binary_buckets = np.array(binary_buckets)
  bin_indices = binary_buckets.dot(powers_of_two)
  string_buckets = bin_indices.astype(str)
```

Regarding the Projection mode, the first necessary operation consists in retrieving the value of the X vector and the scalar value b. A consequent 'Broadcasting' operation will be implemeted to compute the correct bucket for the passed descriptor.

```
else:
  # Hash the point
  X = self.g_functions[0]
  b = self.g_functions[1]

  # use numpy broadcasting
  buckets = np.floor((X.dot(obj_descriptor)+b)/self.w).astype(np.int64)

  def stringfy(x):
    stringfied=""
    for i in x:
      stringfied=stringfied+str(i)
      stringfied=stringfied+"-"
    stringfied=stringfied[0:len(stringfied)-1]
    return stringfied

  string_buckets = np.apply_along_axis(stringfy, 1, buckets)
return string_buckets
```

3 Query

At this point, the index is ready to be used and the last thing that has to be implemented concerns the execution of the query formulation. The following snippet shows the whole set of required parameters; firstly, the method needs the descriptor of the query the user wants to process, wheter the second argument corresponds to the number of descriptors that need to be retrieved.
The last argument corresponds to the distance measure function that the user wants to apply in order to discover the k-most similar elements; as we can see, it concerns just the projection mode because the binary version only uses the Hamming distance.
The results that we can obtain performing this method are the identifiers of k-elements with their corresponding distances to the given query and also th total amount of distance computation that are performed to complete the task.

```
def query(self, q, k=1, distance_func="euclidean"):
  """
  Given a query q retrieve all point from the num_g buckets g-1(q),..., g-num_g(q)

  Parameters
  ----------
  q: numpy.ndarray of shape (descriptor_dim,)
    Descriptor of the query

  k: int, default=1
    Number of descriptors to be retrieved

  distance_func: str, default='euclidean'
    The distance function to be used when mode=="projection".
    Currently it needs to be one of ("euclidean", "cosine")

  Return
  ----------
  neigh_dists: numpy.ndarray of shape (k,)
    Distances of the k found neighbors at most. None if no neighbor is found

  neigh_ids: numpy.ndarray of shape (k,)
    Identifiers of the k found neighbors at most. None if no neighbor is found

  computed_distances: int
    Number of computed distances
  """
```

In order to be processed, the query has to be 'hashed', leveraging also this time the method '_hash' that we have seen before. Through this operation we can update the list of neighbour of the query, that is the list of elements of the dataset that ended up in the same buckets of the query.

```
if k < 1:
  k = 1

buckets = self._hash(q)
neighbors = set()
for i, bucket in enumerate(buckets):
  if not self.overwrite_index and self.in_memory:
    ids = self.index[i].get(str(bucket), [])
    neighbors.update(ids)
  else:
    bucket_filename = os.path.join(str(self.index_dir),f'bucket_{str(i)}')
    with dbm.open(bucket_filename, 'c') as db:
      if bucket in db:
        ids = orjson.loads(db[bucket])
        neighbors.update(ids)

# If no point has been retrieved
if len(neighbors) == 0:
  return np.array([]), np.array([]), 0
```

Once we have the whole list of potential candidates, we can proceed with the computation of distances. For the computations of the cosine similarity and the euclidean distance the sklearn versions have been implemented, whilst for the a specific function have been deployed.

```
if self.binary_mode:
  d_func = self._hamming_distance

else:
  if distance_func == "euclidean":
    d_func = euclidean_distances
  elif distance_func == "cosine":
    d_func = cosine_similarity

# Compute Distances
neighbors = list(neighbors)
distances = d_func(q.reshape(1, q.shape[0]), self.dataset[neighbors])[0]
computed_distances = distances.shape[0]
```

```
def _hamming_distance(self, x, y):
  ''' Compute Hamming Distance

  Parameters
  ----------
  x: numpy.ndarray of shape (1, descriptor_dim)

  y: numpy.ndarray of shape (n, descriptor_dim)

  Return
  ----------
  dist: numpy.ndarray of shape (n,)
  '''
  upper, lower = 1, 0
  query = np.where(x > self.mean, upper, lower)
  neigh = np.where(y > self.mean, upper, lower)
  dist = np.logical_xor(query, np.array(neigh))
  dist = np.count_nonzero(dist, axis=1)
  dist = dist.reshape((1, dist.shape[0]))
  return dist
```

The final operations that need to be done are the ones concerning the sorting of the retrieved elements based on the computed distances and the consecutive skimming and selectionof the k-most relevant elements.

```
# Sort
dtype = [('id', int), ('distance', float)]
dist_array = np.rec.fromarrays([neighbors, distances], dtype=dtype)
dist_array = np.sort(dist_array, order='distance')

# Select only the K-NN
if distance_func == "cosine":
  knn_array = dist_array[-k:][::-1]
else:
  knn_array = dist_array[:k]

knn_ids = []
knn_dists = []
for neigh in knn_array:
  neigh_id = neigh[0]
  neigh_dist = neigh[1]
  knn_ids.append(neigh_id)
  knn_dists.append(neigh_dist)

knn_ids = np.asarray(knn_ids)
knn_dists = np.asarray(knn_dists)

# Return the k nearest
return knn_dists, knn_ids, computed_distances
```

LSH Persistent Storage Benchmark

Another question we asked ourselves was "what kind of persistent storage are we going to use to store our indexes?". The need for a persistent storage is mainly due to the fact that, without it, it would be necessary to recompute the indexes every time. furthermore, we cannot be sure that in a real application environment the index will be able to fit in RAM memory.

We therefore decided to run a benchmark to test which storage mode among the 6 proposed by us was the best for our task. In particular, the modes we tested were:

- **CSV 1**: Create a different csv file for each bucket of each hash function g. Incrementally insert new points by appending them at the end of each bucket they fall into. It requires minimal RAM usage since each point is immediately added to its bucket files.
- **CSV 2**: Create a different csv file for each hash function g. At index creation time, a dictionary is mantained in-memory for each hash function; each dictionary has bucket names as keys and a list of points as value. At the end of the index generation process, each dictionary is saved in a different csv file.
- **CSV 3**: Create a different csv file for each bucket of each hash function g. At index creation time, a dictionary is mantained in-memory for each hash function; each dictionary has bucket names as keys and a list of points as value. At the end of the index generation process, each bucket of each dictionary is saved in a different csv file.
- **DBM 1**: DBM is a Python built-in persistent key-value store. We create a different DBM store for each hash function g. At index creation time, a dictionary is mantained in-memory for each hash function; each dictionary has bucket names as keys and a list of points as value. At the end of the index generation process, each dictionary is saved in a different dbm store. Since DBM only stores strings or bytes, we decided to serialize each bucket's content as a JSON formatted string and then store it as a value, using the bucket name as key.
- **DBM 2**: Same as DBM 1 but makes us of a non-built-in JSON library called orjson, which is stated to be faster on average wrt the Python built-in JSON library.
- **LevelDB**: LevelDB is a fast key-value storage. We create a different LevelDB store for each hash function g. At index creation time, a dictionary is mantained in-memory for each hash function; each dictionary has bucket names as keys and a list of points as value. At the end of the index generation process, each dictionary is saved in a different levelDB store. As serialization library, we decided to use orjson.

Once the storage modes have been identified, we decided to compute 3 different statistics on each of them:

- **Total Index Creation Time**
- **Total Storage Size**
- **Average Query time**: averege time required to compute a query on the index saved on-disk.

We tested a range of hyperparameters combinations. On a first try, we decided to test values of *num_g* in the range [1, 3] and *num_h* in [1, 10], obtaining the subsequent results:

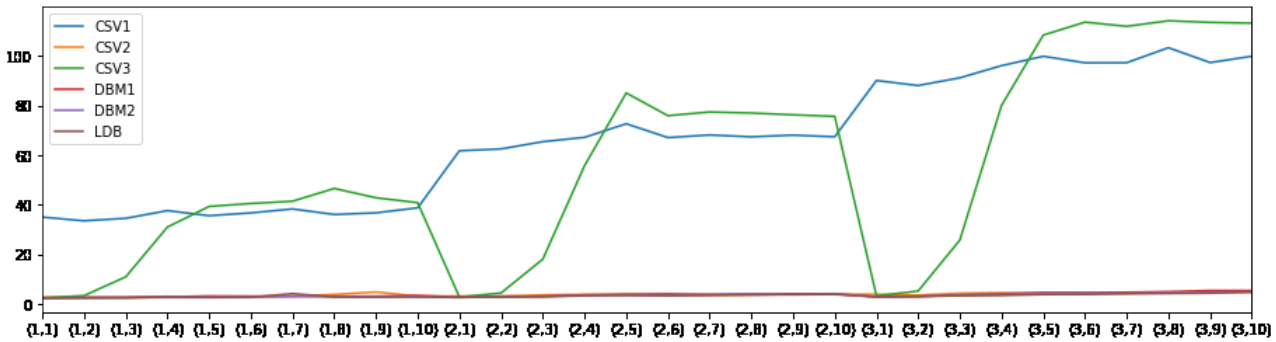

Figure 10: Total Index Creation Time. The y axis is in seconds [s]

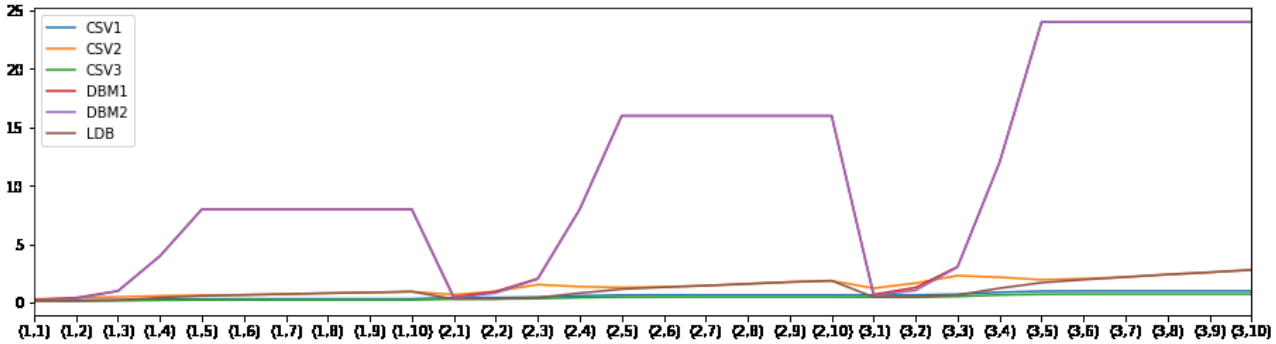

Figure 11: Total Storage Size. The y axis is in MegaBytes [MB]

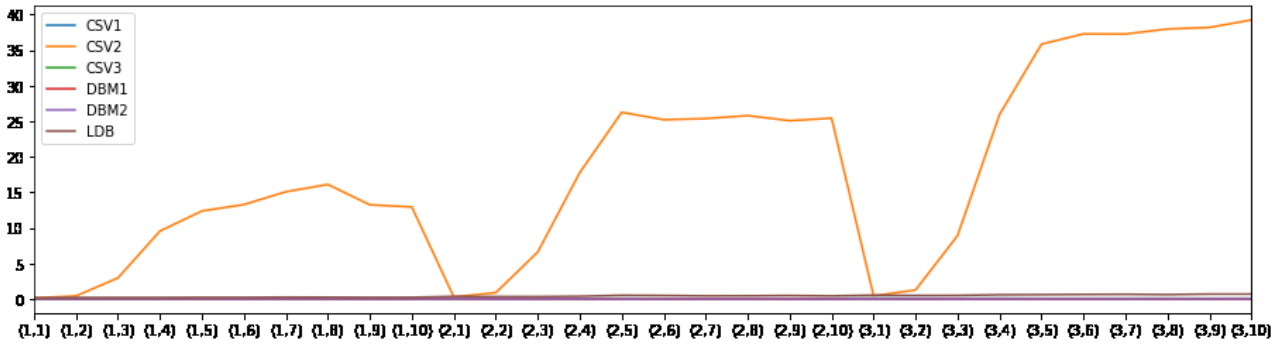

Figure 12: Average Query Time. The y axis is in seconds [s]

As can be seen from the 3 figures, each index has a different behaviour:

- CSV 1 and CSV 3 have very high Total Index Creation Time
- DBM 1 and DBM 2 have very high Total Storage Size
- CSV 2 has very high Average Query Time

In order to better evaluate the differences, for each statistic we plot only those modes which have smaller values.

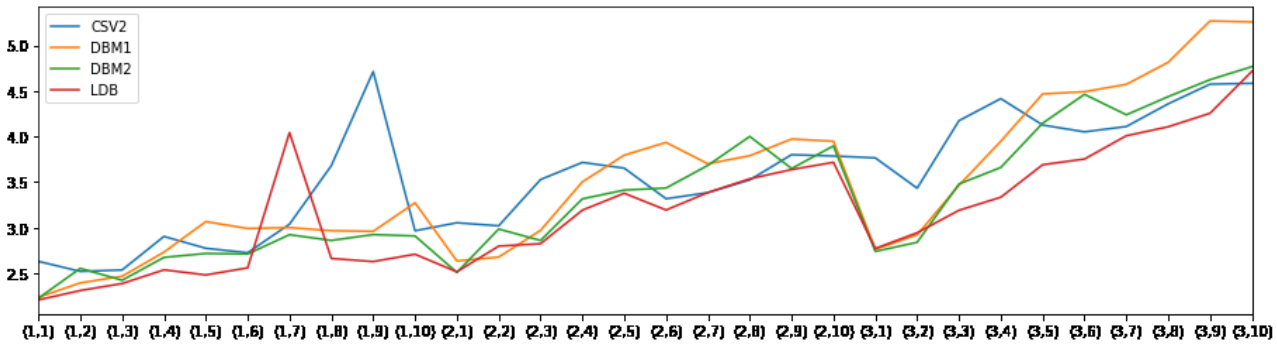

Figure 13: Total Index Creation Time. The y axis is in seconds [s]

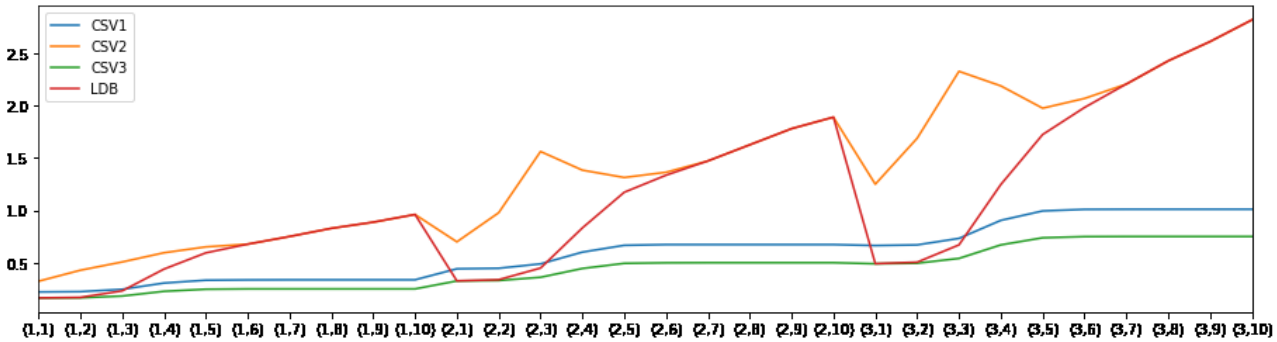

Figure 14: Total Storage Size. The y axis is in MegaBytes [MB]

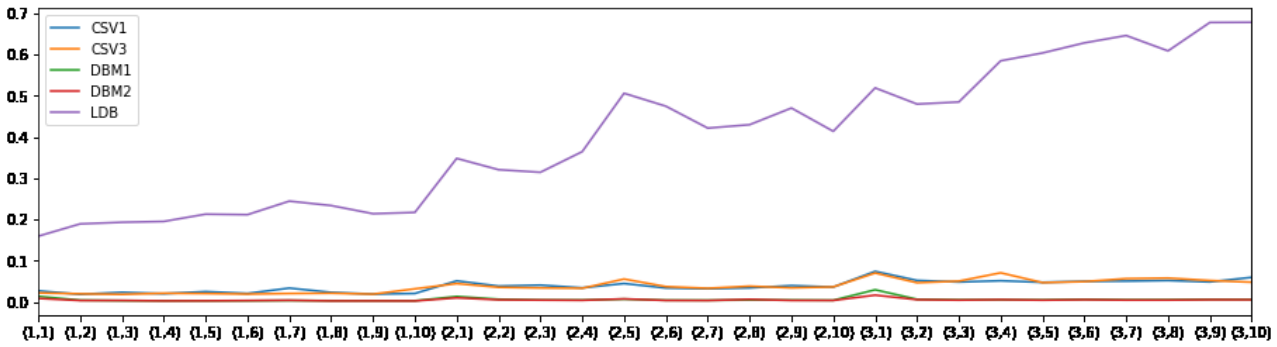

Figure 15: Average Query Time. The y axis is in seconds [s]

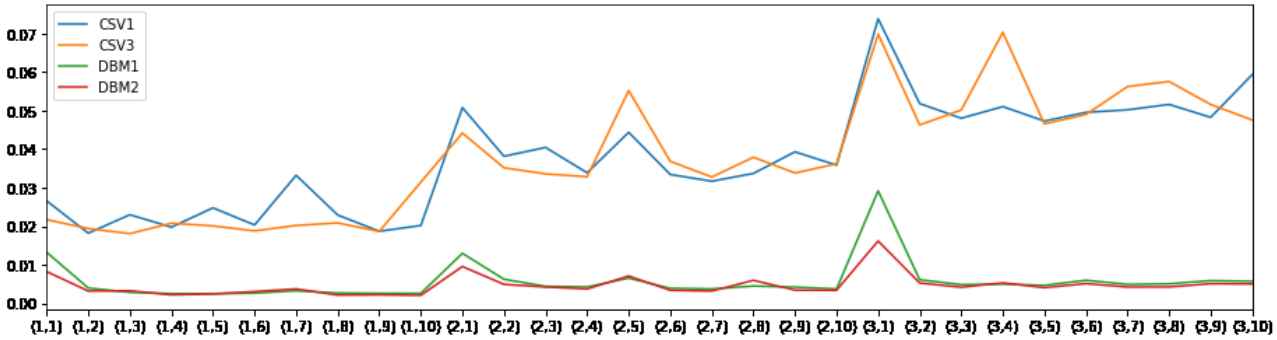

Figure 16: Average Query Time. The y axis is in seconds [s]

Based on this benchmark we were able to make some observations:

1) We can observe some periodic incremental schemes, from which we are able to intuitively derive possible behaviors of the various modes for ranges of hyperparameters other than those tested. In particular, we can observe regularities depending on the variation of *num_h*.
2) The best mode in terms of *Total Index Creation Time* is mainly **LDB**, although **DBM 2** has quite similar performance.
3) The best mode in terms of *Total Storage Size* is **CSV 3**, on pair with **LDB** just for *num_h = 1 or 2*.
4) The best mode in terms of *Average Query Time* is always **DBM 2**.

Since for our project we needed an index that was fast both in terms of creation time (to be able to create many indexes in a feasible time) and average query time, without any constraints in terms of memory usage, we identified **DBM 2** and **LDB** as the best alternatives.

Possible Future Improvements

During the testing phase, we saw that many indexes were not generally able to find the requested *k* quantity of nearest neighboors for a query. This mainly happened in the *LSH projection mode*, which for high values of *num_h* has a tendency to generate sparse buckets.

A possible improvement could be the following: in case, after calling *query(q, k)* to retrieve all the individual points from the same buckets where a query *q* falls into, we find out that there are not at least *k* points, we can then iterate over those retrieved points, starting from the one with the smallest distance from *q*, to try retrieve other points.

Let's call the point with the smallest distance from *q* at iteration *i* as *p_i*, we can then call *query(p_i, k)* and retrieve *p_i* nearest neighbors. We can then do the union between *q'*s nearest neighbors set and *p_i'*s nearest neighbors set, obtaining a new set which contains at least all *q'*s nearest neighbors + any points which are neighbors of *p_i* and not of *q*. We then sort this new set based on the distance from *q*, check if there are enough points and, if not, we can iterate the same process with a different point *p_i* which is the one with the smallest distance from *q* which has not yet been utilized for querying.

In this way we may be able to relieve the sparsity problem at the cost of an increased average query time.

Performance Evaluation

The following tests were carried out using the mobilenet_v3 (large) because from the analyzes conducted in the previous chapters it was found to be the best network obtained. Three macro tests were conducted:

1. Overall Map, Map per class and average query time calculation using the basic model as features extractor.
2. Overall Map, Map per class and average query time calculation using the basic model with the addition of the MLP as features extractor.
3. Overall Map, Map per class and average query time calculation using the basic model with the addition of the MLP followed by fine tuning as features extractor.

In each of the three macro-tests, the Map and the average query time were calculated using the **standard LSH** and its **binary version** as an index, and finally a **brute force** approach was also performed on the features to have a benchmark on the MAP and average query times obtainable. Furthermore, everything was conducted considering two possible scenarios, namely the one in which the **distractor database** (mirflickr25k) does not exist and the one in which it does exist.

The distractor database contains very different images from the target (mushrooms) images that are the subject of our analysis. Figure 1 shows a batch of sample images from the distractor database.

Fig.1: mirflickr25k (the distractor database)

The database (mushrooms) consists of 5367 images, while the distractor consists of 25.000 images. Once the splitting into training, validation and test set was carried out, 270 queries were taken from this last data set to perform the entire analysis. 270 samples were chosen to have an estimate of the best mAp and to then be able to perform also a mAp-per-class. Precisely, looking at the following distribution in figure 2 it is clear that the minimum number of samples in the test set is 32 and belongs to the Suillus class. Therefore, having a total of 9 classes, 30 representative samples were chosen for each class.

Agaricus	36
Amanita	75
Boletus	108
Cortinarius	85
Entoloma	37
Hygrocybe	33
Lactarius	157
Russula	116
Suillus	32

	ImageName	Class
333	0139_4Slu9ELtokU.jpg	Boletus
298	0343_bk4U5tF-Po4.jpg	Boletus
215	0262_bE-cUUNaFnw.jpg	Boletus
153	0726_wNJGG_syL80.jpg	Boletus
334	0513_M5Q-u5HDgBc.jpg	Boletus
...	...	...
189	058_x2Ekv3UNbYo.jpg	Hygrocybe
141	195_WpZovdY4Avg.jpg	Hygrocybe
253	219_TNCiivIfrJo.jpg	Hygrocybe
485	221_vtRk8qlMEA4.jpg	Hygrocybe
448	092_pLnT-Cgm2OA.jpg	Hygrocybe

Fig.2: number of samples per class (left) and query set (right).

The mAp stands for *Mean Average Precision* and for a set of queries is defined as follows:

$$mAp = \frac{\sum_{q=1}^{Q} AveP(q)}{Q} \qquad (1.1)$$

where AveP(q) would be the *average precision* for a given query q, while Q would be the total number of queries on which to calculate the mean of the average precision.

The function used to calculate the AveP is the following:

```
def compute_AveP(ranks, groundtruth, k):
        n_queries = len(ranks)
        ranked_relevance = groundtruth[np.arange(n_queries).reshape(-1,1), ranks]
        prec_at_i = ranked_relevance[:, :k].cumsum(axis=1) / np.arange(1, k+1)
        aps = (ranked_relevance[:, :k] * prec_at_i).sum(axis=1) / k
        return aps
```

The variable *n_queries* corresponds to the Q in the formula (1.1) and therefore will contain the number of queries on which we want to calculate the mAp.
The *groundtruth matrix* would be a matrix of dimension $Q \times N$ where N are the total number of objects in the database. If its cell $[i, j]$ has a value of $True$ then it will indicate that the i-th query and the j-th database object have the same class, otherwise its value will be $False$. The *ranks matrix* is a matrix containing in the i-th row a vector of identifiers ordered by proximity with the relative query. Therefore the element $[i, j]$ of the ranks matrix will contain the j-NN neighbor of the i-th query. So in row two we are simply saying how to sort the groundtruth matrix in order to rearranging its columns appropriately following what the ranks matrix suggests.
The third row calculates the *precision@i*, which is the number of relevant documents retrieved up to i divided by the total number of relevant documents. It will be calculated up to the first best k only.
In the fourth row, the various precisions found above are multiplied with the ranked_relevance matrix in order to count only those relating to the relevant objects that have been recovered, and then divide by k.

Differently from the definition which envisaged dividing by the total of *ground truth positives*, it was decided to normalize this quantity in order to obtain mAp 1 when all the k results contain all relevant documents. The idea is to obtain a normalization factor for each class (npfc) as follows:

$$nfpc_i = \frac{k}{R_i}$$

where k is the number of *nearest neighbors* to return and R_i is the total number of ground truth positives in the database for that i-th class. But dividing the standard AveP by this quantity essentially means dividing it only by k.
Other alternatives would have been to use the total number of objects in the database as k. But it is evident how this choice (also given the countless tests made in the following paragraphs) is expensive. Another alternative was the following:

$$k \geq \max(R_i)$$

However, this measure, although it does not penalize the majority class queries, leads to having to return a lot of elements since the majority class is *Lactarius* with 1250 relevant objects.
The choice of nfpc is also very flexible to test the performance of the real system as k varies.

It was also decided to calculate the mAp-per-class to understand on which class the system performs best and on which it does not. Obviously the same $compute\_AveP\ function$ will be called but giving as arguments the rank and groundtruth matrix relating only to the rows of interest.

In the last paragraph, the total mAp and the mAp-per-class were also calculated with the variation of k (from 1 to 1000 in steps of 20). In this case it is important to foresee that for certain k some classes will have a maximum of relevant objects less than this k. For example the *Suillus* class has only 248 relevant objects. Continuing to normalize for k, when this would exceed 248, even if 248 objects on k are relevant, the remaining $k - 248$ obviously cannot be anymore. We will therefore have a fixed numerator and a denominator (k) which increases thus reducing the mAp. For this reason, in this last type of analysis the following k was chosen for each class:

$$k_i = \min(k, R_i)$$

So that when k exceeds R_i then we will normalize the mAp of that class using R_i itself.

Both the mAp and the mAp-per-class (together with the average query time) were calculated on the results that the various LSH indexes (standard and binary version) produced. As we will see, we will try various combinations of the parameters g and h to finally understand which is the final one that the system will adopt. In particular the g will vary between 1 and 10 as well as the same h. We will see that the LSH will always assume values of large g and small h, while the binary will tend to have values of g on average high but more central for the h. Through the heatmaps it will be easy to understand how the average query time increases as the map increases. However, thanks to the heatmap it is easy to discover combinations that *maximize the mAp and minimize the AQT*.

In order to have a benchmark on the results, a *brute force search* of the k nearest neighbors was done comparing the features with the *cosine similarity*. In reality, for the type of approach it is, the search foresees a complete calculation of all the similarities between queries and database objects and then only at the end take the best k-values.
The function that implements the brute force search is the following:

```
def brute_force_search(queries_features, dataset_features, nq, ndb):
        start_time = time.time()
        scores = np.empty((nq, ndb), dtype = 'float')

        for row, query_feature_i in enumerate(tqdm(queries_features)):
                score_query = cosine_similarity(query_feature_i.reshape(1, -1), dataset_features)
                scores[row] = score_query

        elapsed_time = time.time() - start_time

        ranks = scores.argsort(axis=1)[:, ::-1][:, :k]
        return ranks, elapsed_time / len(queries_features)
```

An important note is the reason for the single calculation of cosine similarity per query. The calculation of the cosine similarity was deliberately avoided in which all the queries and the entire dataset are given as input since the sklearn function is already very optimized on its own and also to have a correct comparison with the results obtained from the various indices, since LSH (standard and binary) calculate the distance taking into account one query at a time.

1.Performance Evaluation of conv-base Mobilenet_v3

In this section we will use the basic mobilenet_v32. Therefore, the features of all queries will be extracted from this network and we will proceed to search for the best k with the support of the standard and binary LSH index and then with the brute force approach. This search will be done first without the presence of the distractor, to then see what changes when this dataset is present.
For each possible g and h value of the standard and binary LSH, the mAp and the average query time (AQT) will be calculated. Finally, the same will be calculated using the brute force approach.

1.1 Without Distractor

The first case involves the analysis of performance without the presence of the distractor.
Below are the mAp and AQT using various combinations of g and h for the standard and binary LSH indexes.

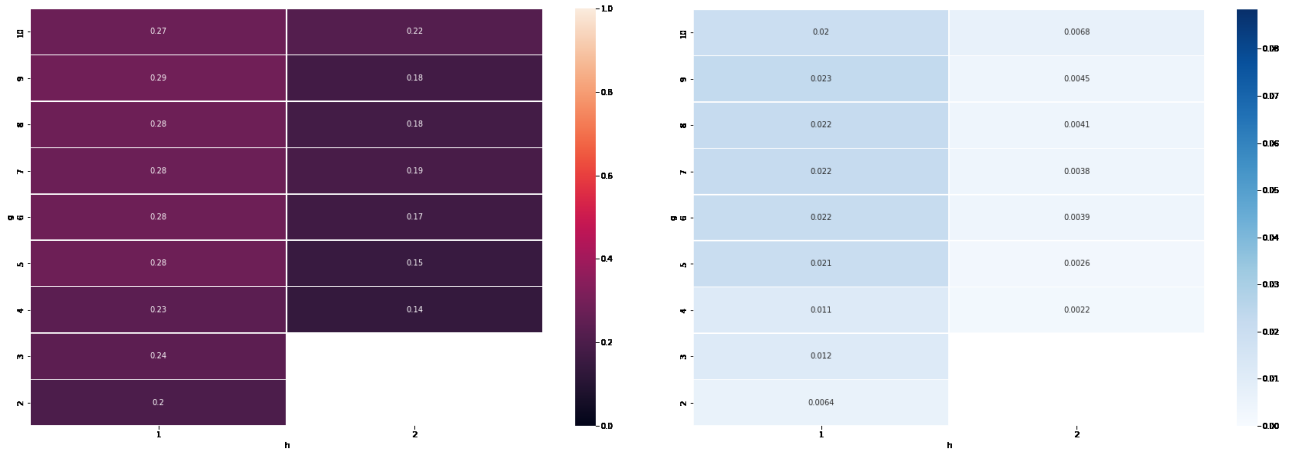

Fig. 3: Heatmap of the MAP and AQT using the standard LSH index

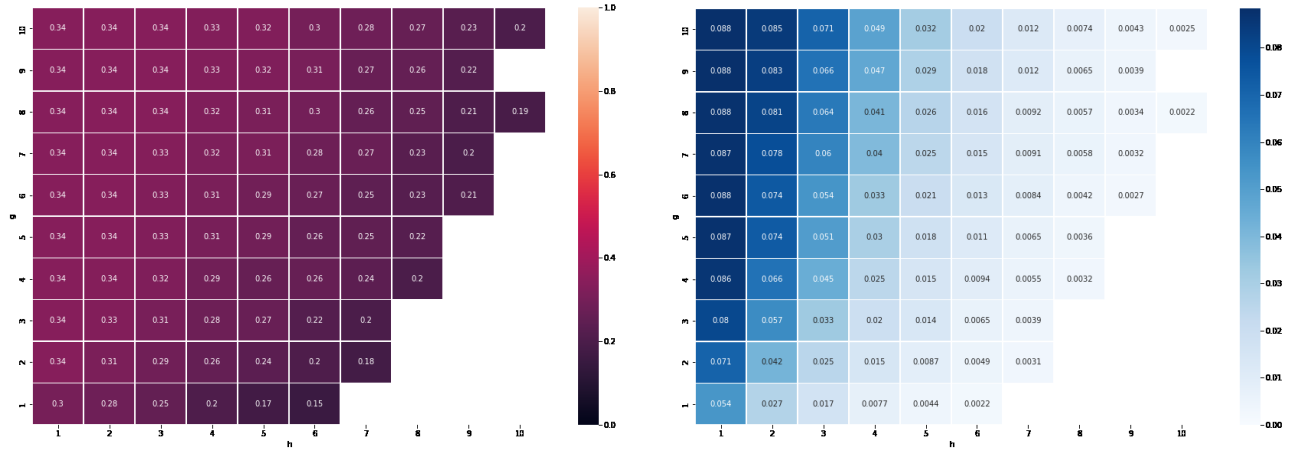

Fig. 4: Heatmap of the MAP and AQT using the binary LSH index

The best combination of the h and g parameters of the LSH is the following:

h = 1 g = 9 with MAP = 0.29 and AQT = 0.023

The best combination of the h and g parameters of the binary LSH is the following:

h = 5 g = 9 with MAP = 0.32 and AQT = 0.029

The mAp-per-class are shown below:

	Class	Map
0	Boletus	0.456780
1	Russula	0.413492
2	Lactarius	0.321758
3	Cortinarius	0.161293
4	Amanita	0.470665
5	Agaricus	0.215791
6	Entoloma	0.089208
7	Suillus	0.149981
8	Hygrocybe	0.290381

	Class	Map
0	Boletus	0.573009
1	Russula	0.421931
2	Lactarius	0.394847
3	Cortinarius	0.169033
4	Amanita	0.488532
5	Agaricus	0.210714
6	Entoloma	0.148659
7	Suillus	0.155963
8	Hygrocybe	0.324467

Fig 5: Best mAp-per-class using LSH (left) and binary LSH (right)

The following images are the k = 20 best results obtained on the 270 queries. For graphical reasons, only the first 10 of them were reported, and also only one sample of queries per class was shown.

Fig. 6: the k = 20 best results obtained with the LSH index. In the example, one sample per class is shown

Fig. 7: the k = 20 best results obtained with the binary LSH index. In the example, one sample per class is shown

The following is instead the result of the mAp, mAp-per-class and AQT obtained following a brute force search approach.

MAP: 0.33
AQT: 0.041

	Class	Map
0	Boletus	0.551564
1	Russula	0.418364
2	Lactarius	0.383276
3	Cortinarius	0.219287
4	Amanita	0.466947
5	Agaricus	0.255990
6	Entoloma	0.158635
7	Suillus	0.176979
8	Hygrocybe	0.335725

Fig. 7: best mAp-per-class using brute force search

Fig. 8: the k = 20 best results obtained with brute force search. In the example, one sample per class is shown

1.2 With Distractor

The second case concerns the analysis of the performance in the presence of the distractor.
Below are mAp and AQT using various combinations of g and h for standard and binary LSH indexes.

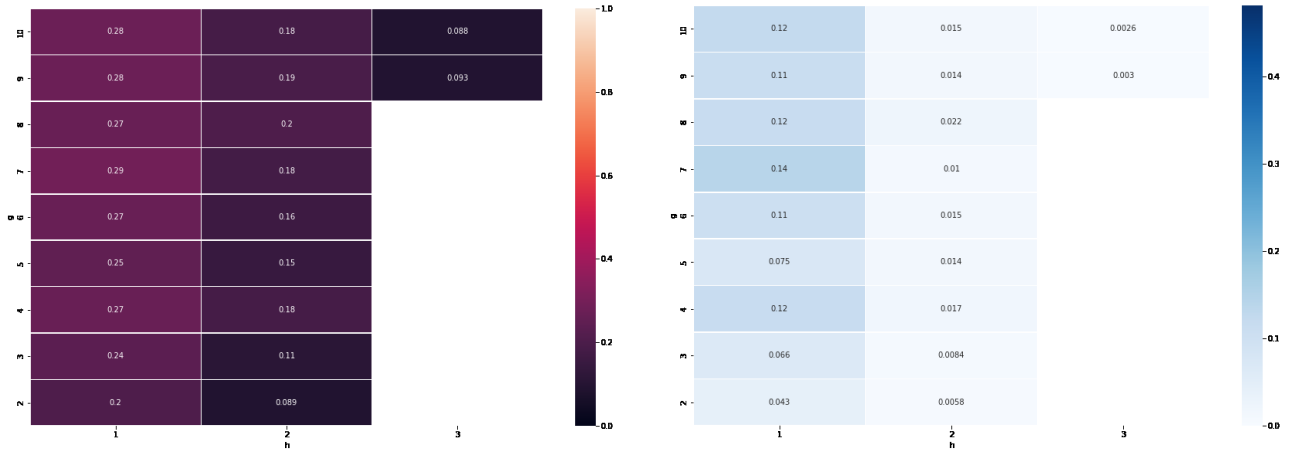

Fig. 9: Heatmap of the MAP and AQT using the standard LSH index

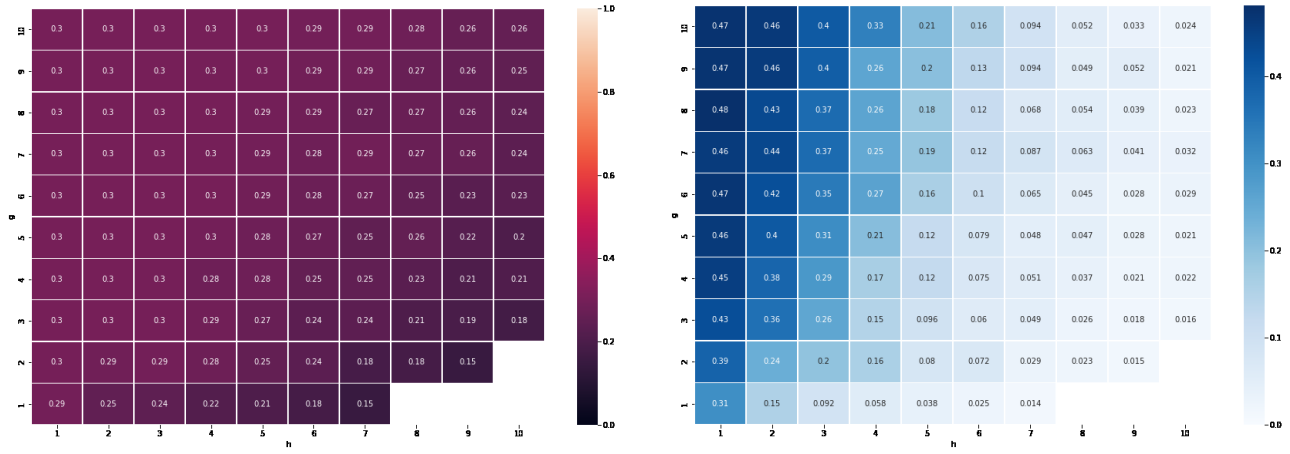

Fig. 10: Heatmap of the MAP and AQT using the binary LSH index

Best combination of LSH h and g parameters:

h = 1 g = 5 with MAP = 0.25 and AQT = 0.075

Best combination of the h and g parameters of the binary LSH:

h = 7 g = 7 with MAP = 0.29 and AQT = 0.087

The mAp-per-class are shown below:

	Class	Map
0	Boletus	0.420579
1	Russula	0.362339
2	Lactarius	0.262364
3	Cortinarius	0.111390
4	Amanita	0.488611
5	Agaricus	0.164097
6	Entoloma	0.067898
7	Suillus	0.108053
8	Hygrocybe	0.242040

	Class	Map
0	Boletus	0.522110
1	Russula	0.390720
2	Lactarius	0.371491
3	Cortinarius	0.138376
4	Amanita	0.407531
5	Agaricus	0.210735
6	Entoloma	0.133586
7	Suillus	0.164924
8	Hygrocybe	0.273241

Fig 11: Best mAp-per-class using LSH (left) and binary LSH (right)

The following images are the k = 20 best results obtained on the 270 queries. For graphical reasons, only the first 10 of them were reported, and also only one sample of queries per class was shown.

Fig. 12: the k = 20 best results obtained with the LSH index. In the example, one sample per class is shown

Fig. 13: the k = 20 best results obtained with the binary LSH index. In the example, one sample per class is shown

The following is instead the result of the mAp, mAp-per-class and AQT obtained following a brute force search approach.

MAP: 0.33

AQT: 0.244

	Class	Map
0	Boletus	0.550755
1	Russula	0.418061
2	Lactarius	0.381735
3	Cortinarius	0.217607
4	Amanita	0.466822
5	Agaricus	0.254146
6	Entoloma	0.158120
7	Suillus	0.173162
8	Hygrocybe	0.335275

Fig. 13: best mAp-per-class using brute force search

Fig. 14: the k = 20 best results obtained with brute force search. In the example, one sample per class is shown

2.Performance Evaluation of Mobilenet_v3 with a MLP classifier

In this section we will use the basic mobilenet_v32 with a MLP on the top. Therefore, the features of all queries will be extracted from this network and we will proceed to search for the best k with the support of the standard and binary LSH index and then with the brute force approach. This search will be done first without the presence of the distractor, to then see what changes when this dataset is present.
For each possible g and h value of the standard and binary LSH, the mAp and the average query time (AQT) will be calculated. Finally, the same will be calculated using the brute force approach.

2.1 Without Distractor

The first case involves the analysis of performance without the presence of the distractor.
Below are the mAp and AQT using various combinations of g and h for the standard and binary LSH indexes.

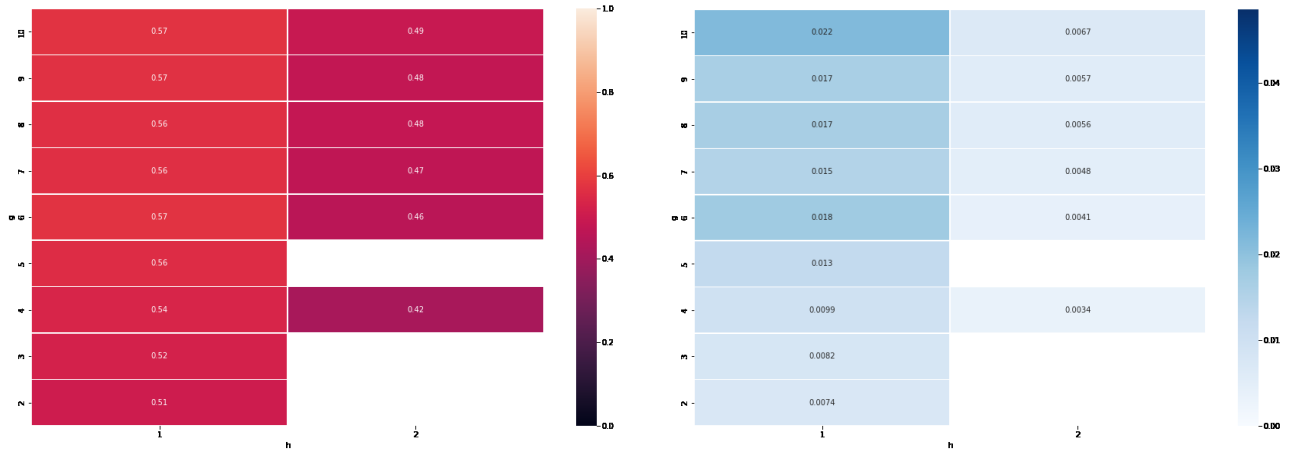

Fig. 15: Heatmap of the MAP and AQT using the standard LSH index

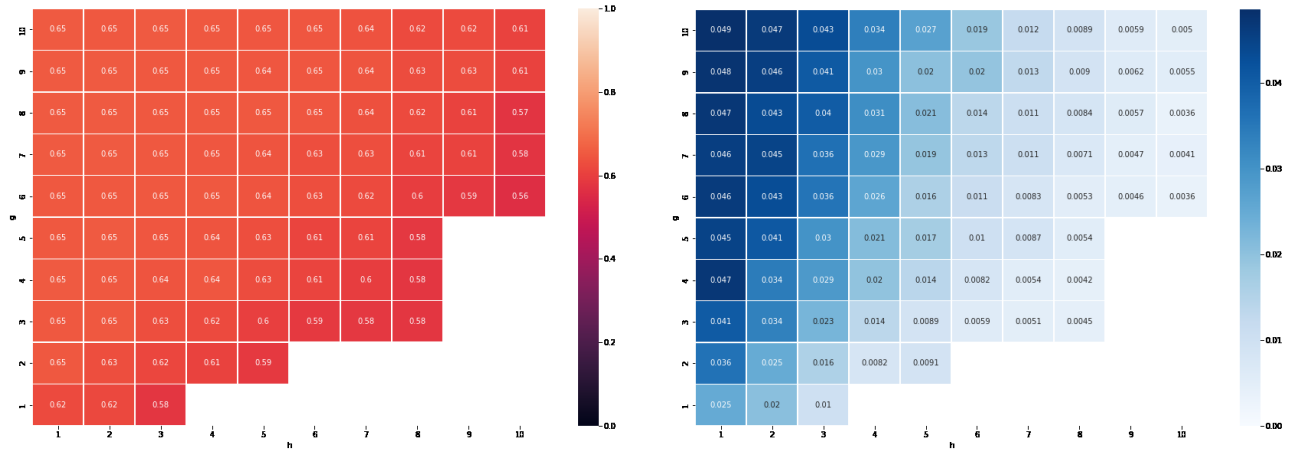

Fig. 16: Heatmap of the MAP and AQT using the binary LSH index

Best combination of LSH h and g parameters:

h = 1 g = 9 with MAP = 0.57 and AQT = 0.017

Best combination of the h and g parameters of the binary LSH:

h = 6 g = 8 with MAP = 0.65 and AQT = 0.014

The mAp-per-class are shown below:

	Class	Map
0	Boletus	0.815697
1	Russula	0.678073
2	Lactarius	0.599971
3	Cortinarius	0.433755
4	Amanita	0.689483
5	Agaricus	0.426911
6	Entoloma	0.422703
7	Suillus	0.439070
8	Hygrocybe	0.605589

	Class	Map
0	Boletus	0.856660
1	Russula	0.719658
2	Lactarius	0.682959
3	Cortinarius	0.466806
4	Amanita	0.756584
5	Agaricus	0.563275
6	Entoloma	0.516958
7	Suillus	0.556447
8	Hygrocybe	0.690238

Fig 17: Best mAp-per-class using LSH (left) and binary LSH (right)

The following images are the k = 20 best results obtained on the 270 queries. For graphical reasons, only the first 10 of them were reported, and also only one sample of queries per class was shown.

Fig. 18: the k = 20 best results obtained with the LSH index. In the example, one sample per class is shown

Fig. 19: the k = 20 best results obtained with the binary LSH index. In the example, one sample per class is shown

The following is instead the result of the mAp, mAp-per-class and AQT obtained following a brute force search approach. Note how the binary LSH reaches the same mAp (obviously with much lower AQT).

MAP: 0.65
AQT: 0.019

	Class	Map
0	Boletus	0.885426
1	Russula	0.744069
2	Lactarius	0.686383
3	Cortinarius	0.491474
4	Amanita	0.762038
5	Agaricus	0.550653
6	Entoloma	0.523346
7	Suillus	0.519295
8	Hygrocybe	0.688276

Fig. 20: best mAp-per-class using brute force search

Fig. 21: the k = 20 best results obtained with brute force search. In the example, one sample per class is shown

2.1 With Distractor

The second case concerns the analysis of the performance in the presence of the distractor.
Below are mAp and AQT using various combinations of g and h for standard and binary LSH indexes.

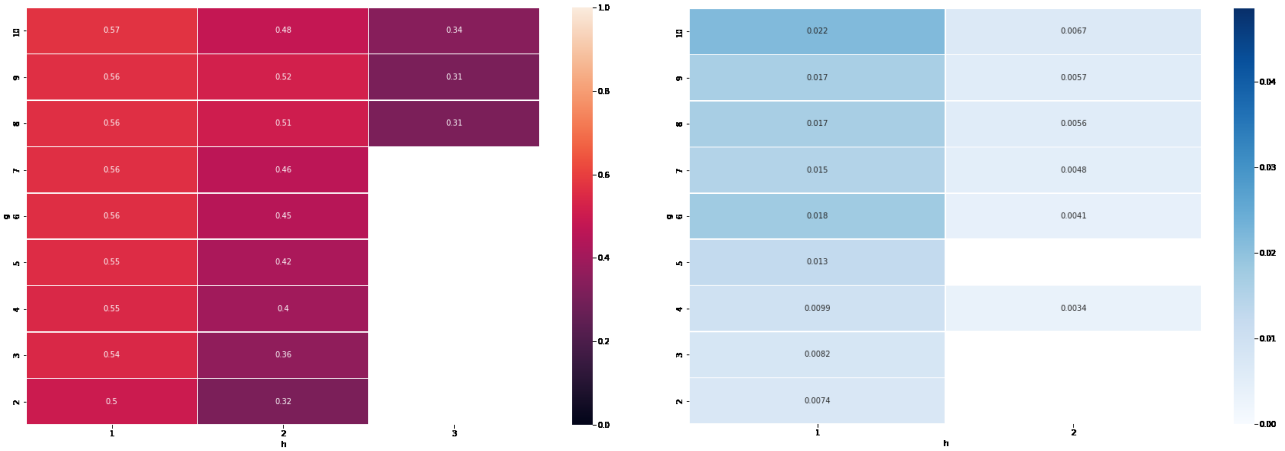

Fig. 22: Heatmap of the MAP and AQT using the standard LSH index

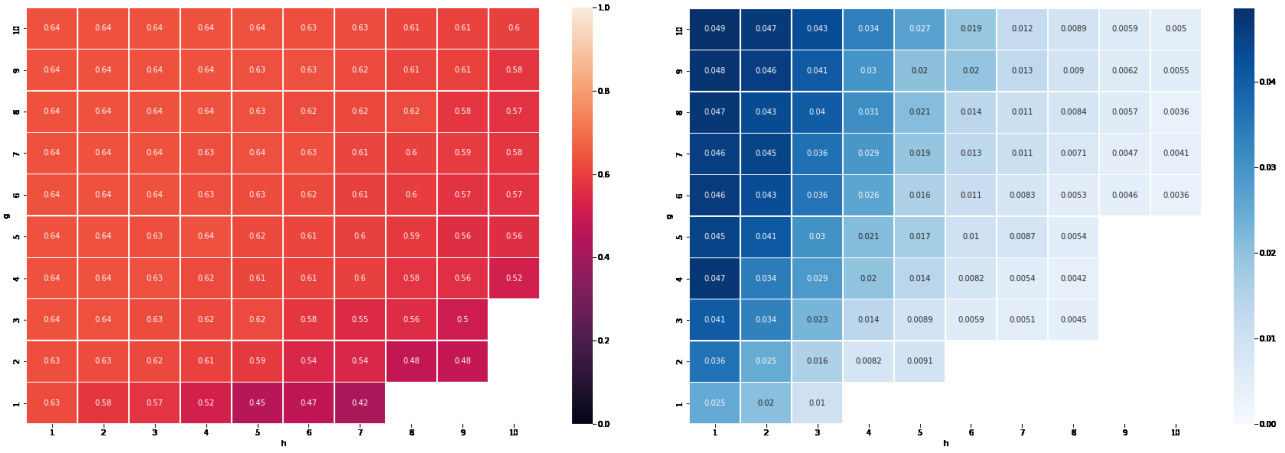

Fig. 23: Heatmap of the MAP and AQT using the binary LSH index

Best combination of LSH h and g parameters:

h = 1 g = 7 with MAP = 0.56 and AQT = 0.015

Best combination of the h and g parameters of the binary LSH:

h = 6 g = 7 with MAP = 0.63 and AQT = 0.013

The mAp-per-class are shown below:

	Class	Map
0	Boletus	0.807325
1	Russula	0.666852
2	Lactarius	0.594427
3	Cortinarius	0.451131
4	Amanita	0.680236
5	Agaricus	0.430280
6	Entoloma	0.444163
7	Suillus	0.403467
8	Hygrocybe	0.589602

	Class	Map
0	Boletus	0.855224
1	Russula	0.751410
2	Lactarius	0.707422
3	Cortinarius	0.454747
4	Amanita	0.753835
5	Agaricus	0.519449
6	Entoloma	0.489629
7	Suillus	0.503067
8	Hygrocybe	0.655855

Fig 24: Best mAp-per-class using LSH (left) and binary LSH (right)

The following images are the k = 20 best results obtained on the 270 queries. For graphical reasons, only the first 10 of them were reported, and also only one sample of queries per class was shown.

Fig. 25: the k = 20 best results obtained with the LSH index. In the example, one sample per class is shown

Fig. 26: the k = 20 best results obtained with the binary LSH index. In the example, one sample per class is shown

The following is instead the result of the mAp, mAp-per-class and AQT obtained following a brute force search approach.

MAP: 0.64

AQT: 0.1141

	Class	Map
0	Boletus	0.882244
1	Russula	0.743956
2	Lactarius	0.675985
3	Cortinarius	0.488506
4	Amanita	0.759521
5	Agaricus	0.548550
6	Entoloma	0.517645
7	Suillus	0.511422
8	Hygrocybe	0.675662

Fig. 27: best mAp-per-class using brute force search

Fig. 28: the k = 20 best results obtained with the brute force search. In the example, one sample per class is shown

3.Performance Evaluation of Mobilenet_v3 fine tuned

In this section we will use the basic mobilenet_v32 with the MLP on the top, but now fine tuned. Therefore, the features of all queries will be extracted from this network and we will proceed to search for the best k with the support of the standard and binary LSH index and then with the brute force approach. This search will be done first without the presence of the distractor, to then see what changes when this dataset is present.
For each possible g and h value of the standard and binary LSH, the mAp and the average query time (AQT) will be calculated. Finally, the same will be calculated using the brute force approach.

3.1 Without Distractor

The first case involves the analysis of performance without the presence of the distractor.
Below are the mAp and AQT using various combinations of g and h for the standard and binary LSH indexes.

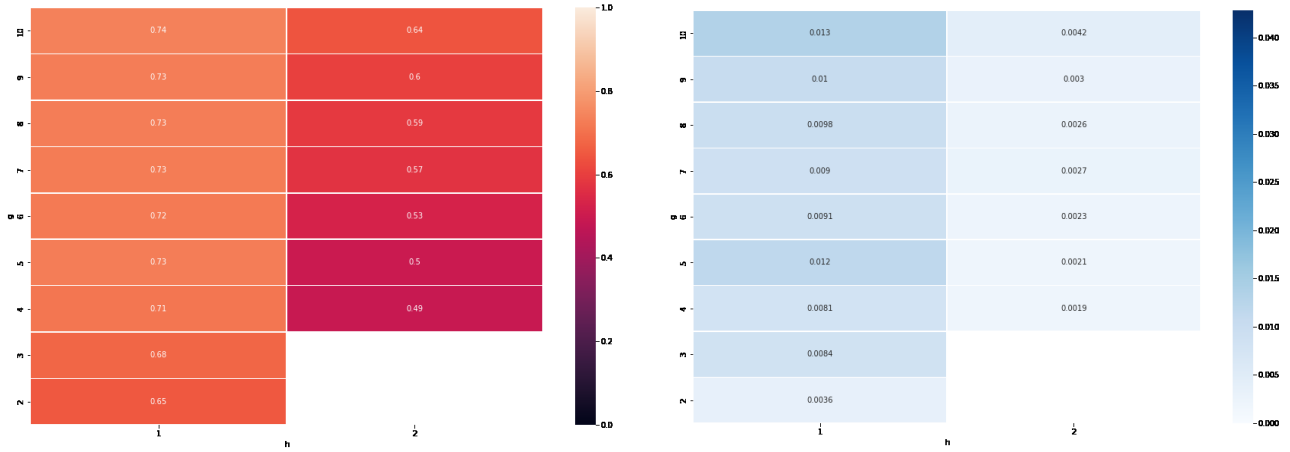

Fig. 29: Heatmap of the MAP and AQT using the standard LSH index

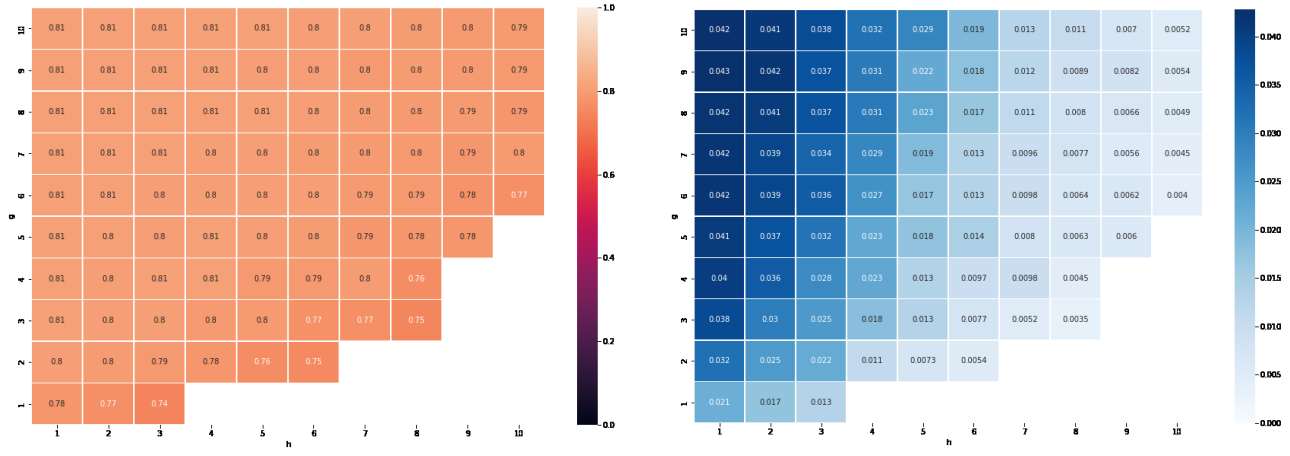

Fig. 30: Heatmap of the MAP and AQT using the binary LSH index

Best combination of LSH h and g parameters:

h = 1 g = 7 with MAP = 0.73 and AQT = 0.009

Best combination of the h and g parameters of the binary LSH:

h = 10 g = 7 with MAP = 0.80 and AQT = 0.0045

The mAp-per-class are shown below:

	Class	Map
0	Boletus	0.921367
1	Russula	0.804420
2	Lactarius	0.748127
3	Cortinarius	0.738713
4	Amanita	0.774538
5	Agaricus	0.616016
6	Entoloma	0.563516
7	Suillus	0.676247
8	Hygrocybe	0.692218

	Class	Map
0	Boletus	0.956230
1	Russula	0.821800
2	Lactarius	0.735366
3	Cortinarius	0.812156
4	Amanita	0.836418
5	Agaricus	0.784096
6	Entoloma	0.737773
7	Suillus	0.698186
8	Hygrocybe	0.783476

Fig 31: Best mAp-per-class using LSH (left) and binary LSH (right)

The following images are the k = 20 best results obtained on the 270 queries. For graphical reasons, only the first 10 of them were reported, and also only one sample of queries per class was shown.

Fig. 32: the k = 20 best results obtained with the LSH index. In the example, one sample per class is shown

Fig. 33: the k = 20 best results obtained with the binary LSH index. In the example, one sample per class is shown

The following is instead the result of the mAp, mAp-per-class and AQT obtained following a brute force search approach.

MAP: 0.81
AQT: 0.020

	Class	Map
0	Boletus	0.971401
1	Russula	0.808644
2	Lactarius	0.751943
3	Cortinarius	0.829747
4	Amanita	0.841096
5	Agaricus	0.761376
6	Entoloma	0.712757
7	Suillus	0.705633
8	Hygrocybe	0.795812

Fig. 34: best mAp-per-class using brute force search

Fig. 35: the k = 20 best results obtained with brute force search. In the example, one sample per class is shown

3.2 With Distractor

The second case concerns the analysis of the performance in the presence of the distractor.
Below are mAp and AQT using various combinations of g and h for standard and binary LSH indexes.

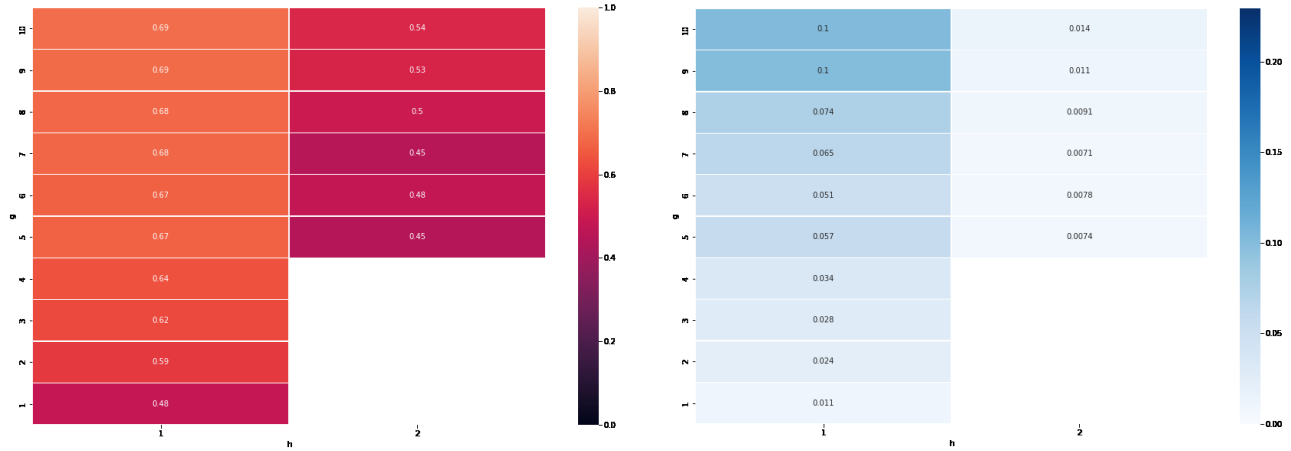

Fig. 36: Heatmap of the MAP and AQT using the standard LSH index

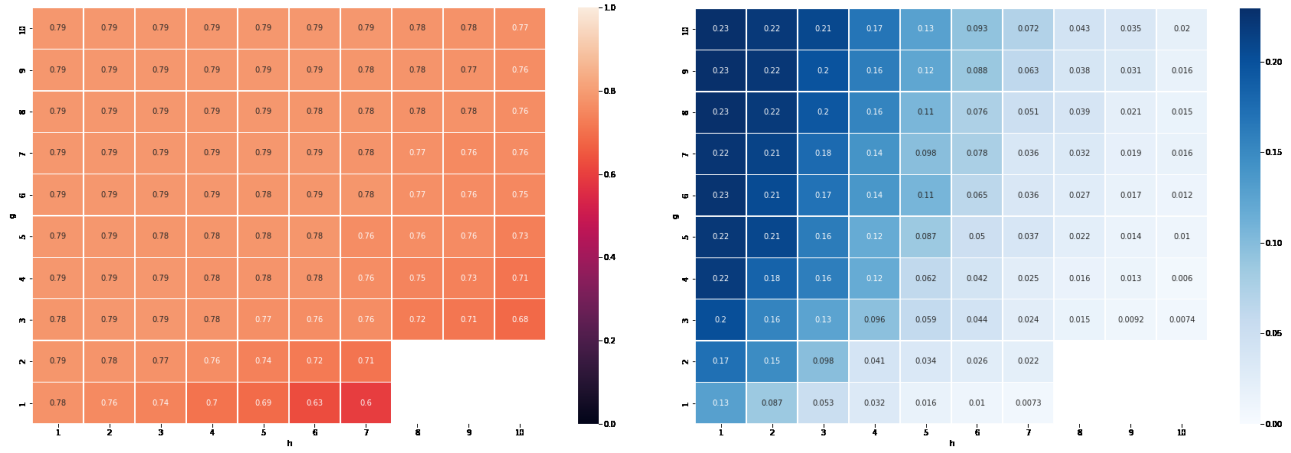

Fig. 37: Heatmap of the MAP and AQT using the binary LSH index

Best combination of LSH h and g parameters:

h = 1 g = 7 with MAP = 0.68 and AQT = 0.065

Best combination of the h and g parameters of the binary LSH:

h = 6 g = 6 with MAP = 0.79 and AQT = 0.065

The mAp-per-class are shown below:

	Class	Map
0	Boletus	0.911068
1	Russula	0.771297
2	Lactarius	0.701179
3	Cortinarius	0.687689
4	Amanita	0.720162
5	Agaricus	0.597792
6	Entoloma	0.511237
7	Suillus	0.599303
8	Hygrocybe	0.603833

	Class	Map
0	Boletus	0.949069
1	Russula	0.816904
2	Lactarius	0.752339
3	Cortinarius	0.766871
4	Amanita	0.822889
5	Agaricus	0.730777
6	Entoloma	0.687786
7	Suillus	0.577280
8	Hygrocybe	0.735494

Fig 38: Best mAp-per-class using LSH (left) and binary LSH (right)

The following images are the k = 20 best results obtained on the 270 queries. For graphical reasons, only the first 10 of them were reported, and also only one sample of queries per class was shown.

Fig. 39: the k = 20 best results obtained with the LSH index. In the example, one sample per class is shown

Fig. 40: the k = 20 best results obtained with the binary LSH index. In the example, one sample per class is shown

The following is instead the result of the mAp, mAp-per-class and AQT obtained following a brute force search approach.

MAP: 0.79

AQT: 0.1134

	Class	Map
0	Boletus	0.971401
1	Russula	0.808644
2	Lactarius	0.751943
3	Cortinarius	0.829747
4	Amanita	0.841096
5	Agaricus	0.761376
6	Entoloma	0.712757
7	Suillus	0.705633
8	Hygrocybe	0.795812

Fig. 41: best mAp-per-class using brute force search

Fig. 42: i k = 20 best results obtained with the brute force search. In the example, one sample per class is shown

4. In-depth performance evaluations of the best Mobilenet_v3 model with the best index

Currently the standard LSH works using the Euclidean distance as the distance. Let's analyze how the map changes when cosine similarity is used instead of using the Euclidean distance. What we achieved earlier were the following stats:

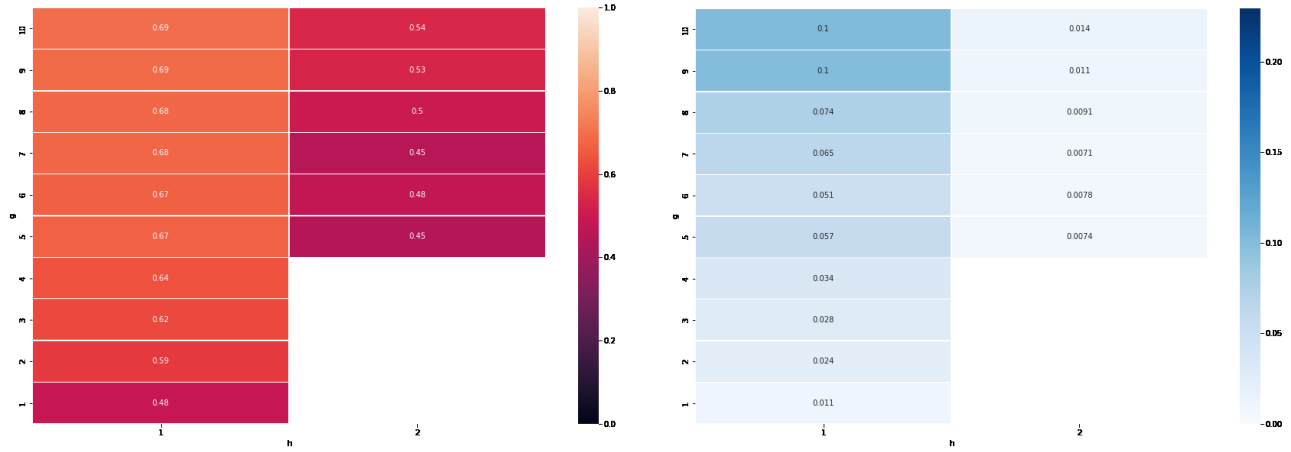

Fig. 43: Heatmap of the MAP and AQT using the standard LSH index with euclidean distance

These instead are those obtained using the cosine similarity:

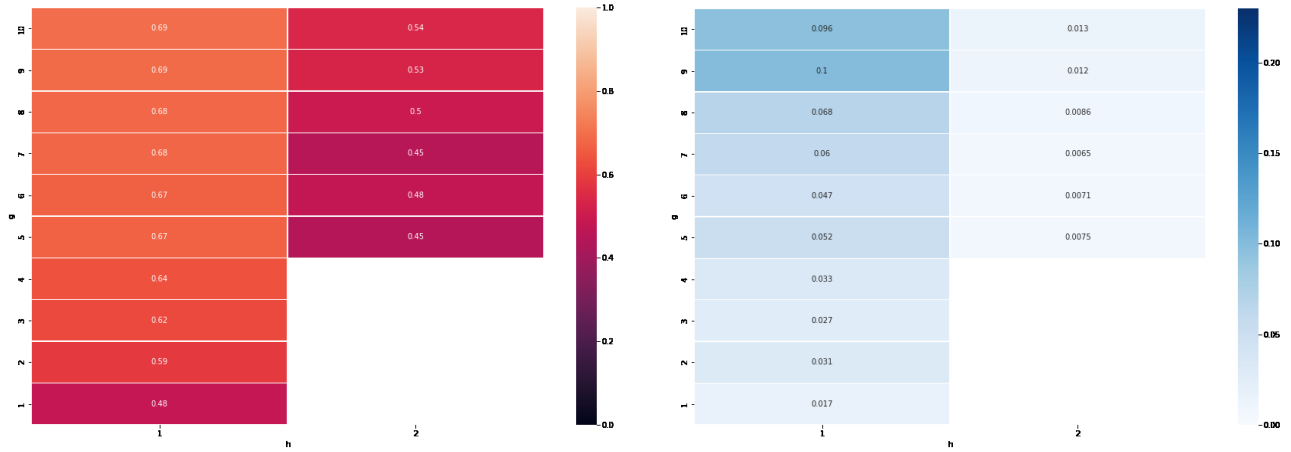

Fig. 44: Heatmap of the MAP and AQT using the LSH index with cosine similarity

As you can see, the choice of distance does not change anything except some very small improvement in times.

Let's see how the map changes when I increase the number of k-NNs to return. The best binary LSH was found to be the one with parameters g = 6 and g = 6 and distance used by cosine similarity.

As already mentioned, due to the fact that k will vary between 1 and 1000, in steps of 10, the relevant documents of some classes may not be sufficient and therefore I will divide by k = min (k, R).

As we can see from the last figures it is recommended (for the final system) not to return more than 30 results otherwise the map degrades too much.

The following figure shows how the total mAp varies as k varies.

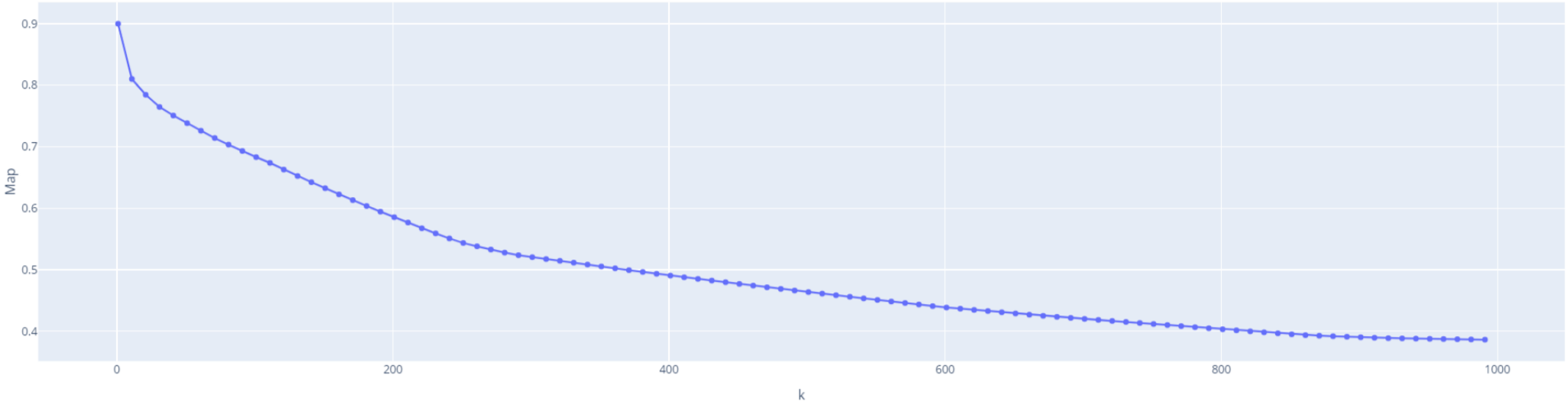

Fig. 45: variation of the total mAp as the parameter k varies

Below is a graph showing how the mAp-per-class varies as k varies.

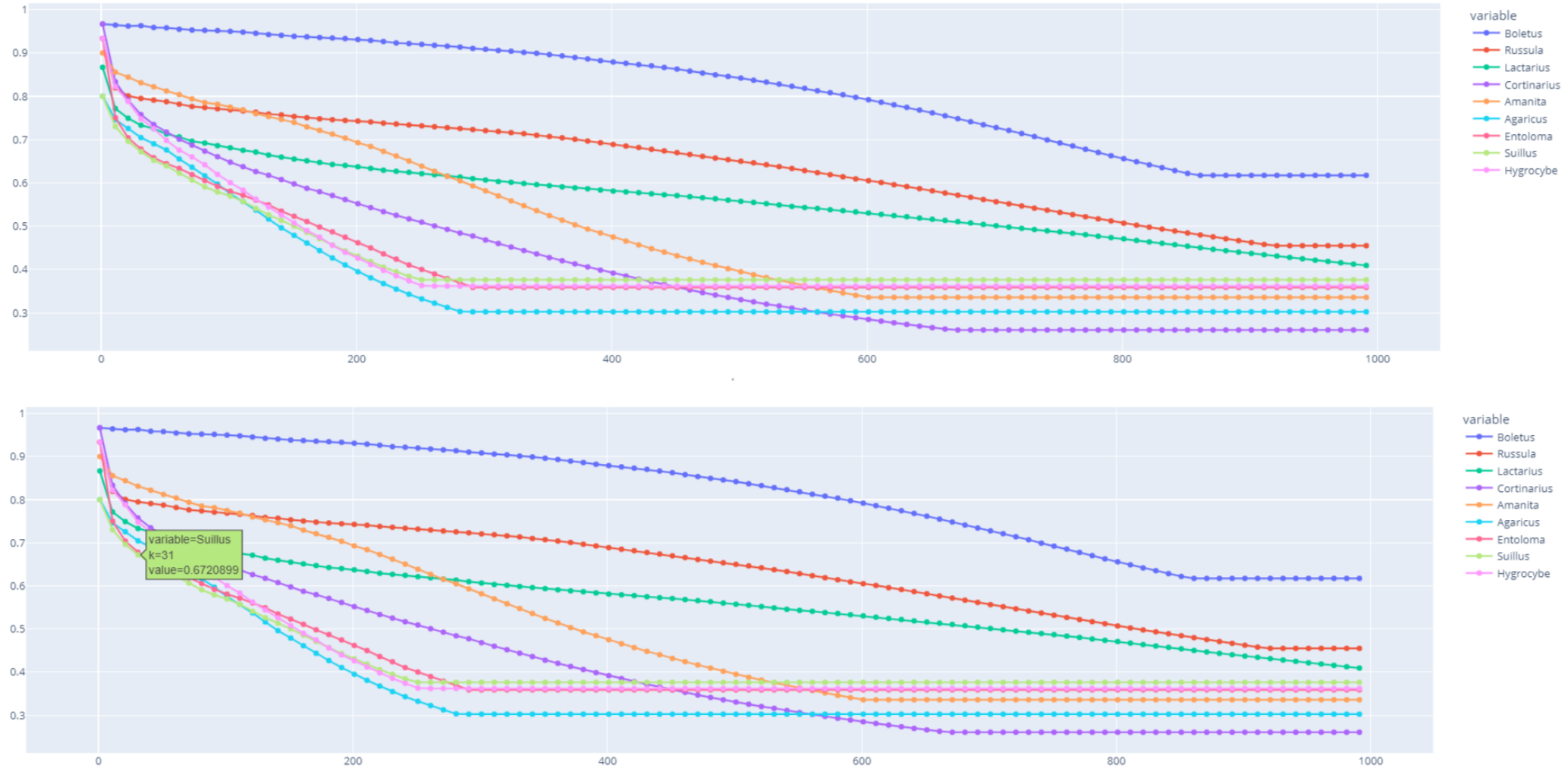

Fig. 46: variation of the mAp-per-class as the parameter k varies

WebApp – Mushrooms Search Engine

After the analysis of the performance of the LSH index and the MobileNetV3, we exploit all these information to develop, with the best LSH index and the best model of the MobileNet, a webapp, named “Mushrooms Search Engine”. It is a very simple web application where its aim is to retrieve the ten near neighbours similar images given as a query an image. In this case, our search engine, will be specialized to retrieve in a precise manner mushrooms images given a query that represents a specific type of mushrooms but can be tested even with any image. The web application is based on a RESTFUL model, in particular consist of:

- **Client Side**: it consists in a simple web page, implemented with html/css/javascript, in which the user can test the application loading an image. The image that the user load is encoded in base64 and it is sent to the Server Side through a POST request to a specific REST API exposed by the Server.

- **Server Side**: it is a FLASK server, written in python, which exposes an endpoint, with path “<url>/knn”, for the Client Side that, as mentioned before, accept POST request and retrieve the ten best similar images respect to the image received by the client. The server is exposed through a public URL with the use of NGROK.
  The server side is very simple and it is composed by 2 files:
  - the first, “Server_Flask”, implement the server which accept the request and response to the client.
  - The second, “Manage_Request” in which are present the functions to compute the actual operation to retrieve the ten near neighbours with the best LSH index and the best MobileNet

After the retrieval of the images from the dataset, the server in the body of the response a json message with this type of structure:

```
/**
 *
 * @param response
 * {
 *    stat: {'time_LSH': timeLsh, 'time_total': totalTime},
 *    images:[
 *       {"label":"imageLabel", "image":<base64Image>},
        {"label":"imageLabel", "image":<base64Image>},
        {"label":"imageLabel", "image":<base64Image>},
                ……
                ……
 *    ]
```

```
*}
*/
```

- “stat” is a field in which are report some information about the time to make the query to the dataset with LSH index, in particular:
  - “time_LSH” is the time of the index to retrieve the ten near neighbours from the dataset
  - “total_time” is the total time for the composition od the message shown above.

- “images” is an array in which there are the ten near neighbours. An image has 2 fields:
  - “label” that represent the class of the image, it contains the class, so the type of the mushroom, if the image belongs to mushrooms dataset otherwise it contains the string “other”
  - “image” is the image retrieved encoded in base64

At the end, when the response from the server is arrived, the Client convert the images and shows to the user the ten near neighbours and relative label and also the statistic about the query processing time.

User Manual

1) First of all, we need to have the public URL (underlined in red) provided by the FLASK Server (Figure 1) and insert the URL in the popup (Figure 2).

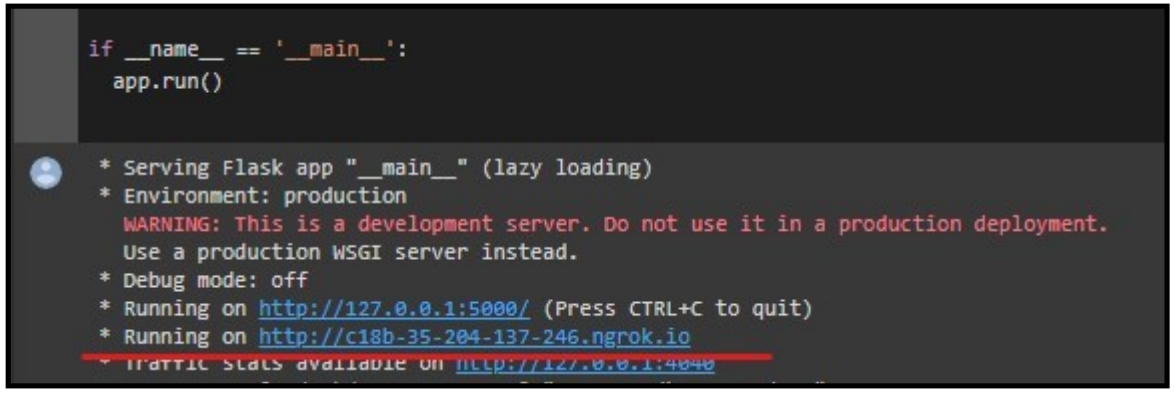

Figure 1: URL Server Flask

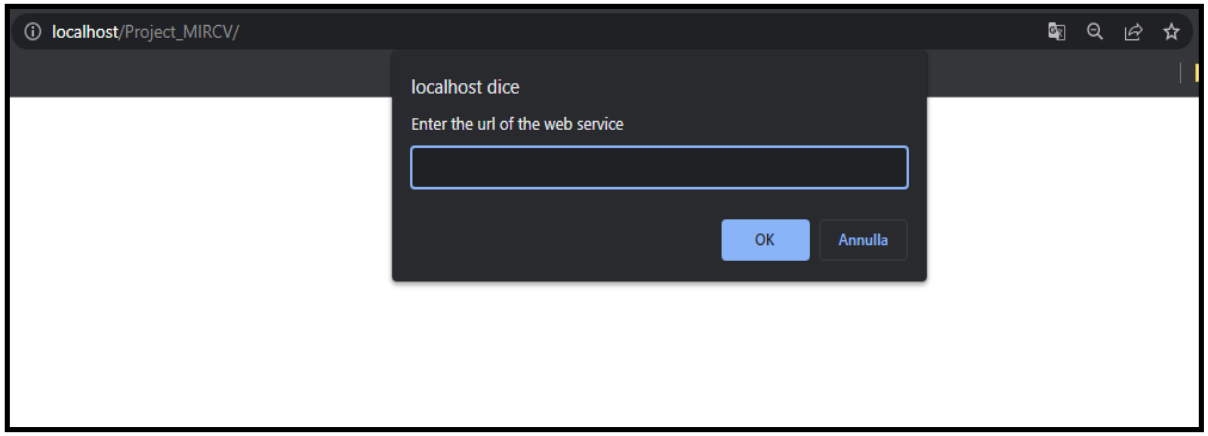

Figure 2: Popup where insert the public URL

2) Click on “Upload Image” and select an image from your local device (Figure 3)

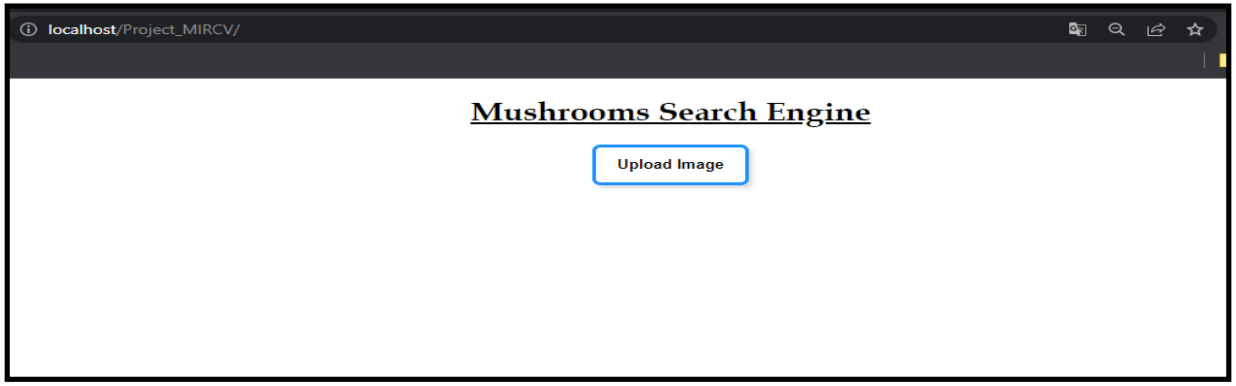

Figure 17: homepage of the Web application

4) After the load of the image, the Web application shows the ten near neighbours respect to loaded image (Figure 4)

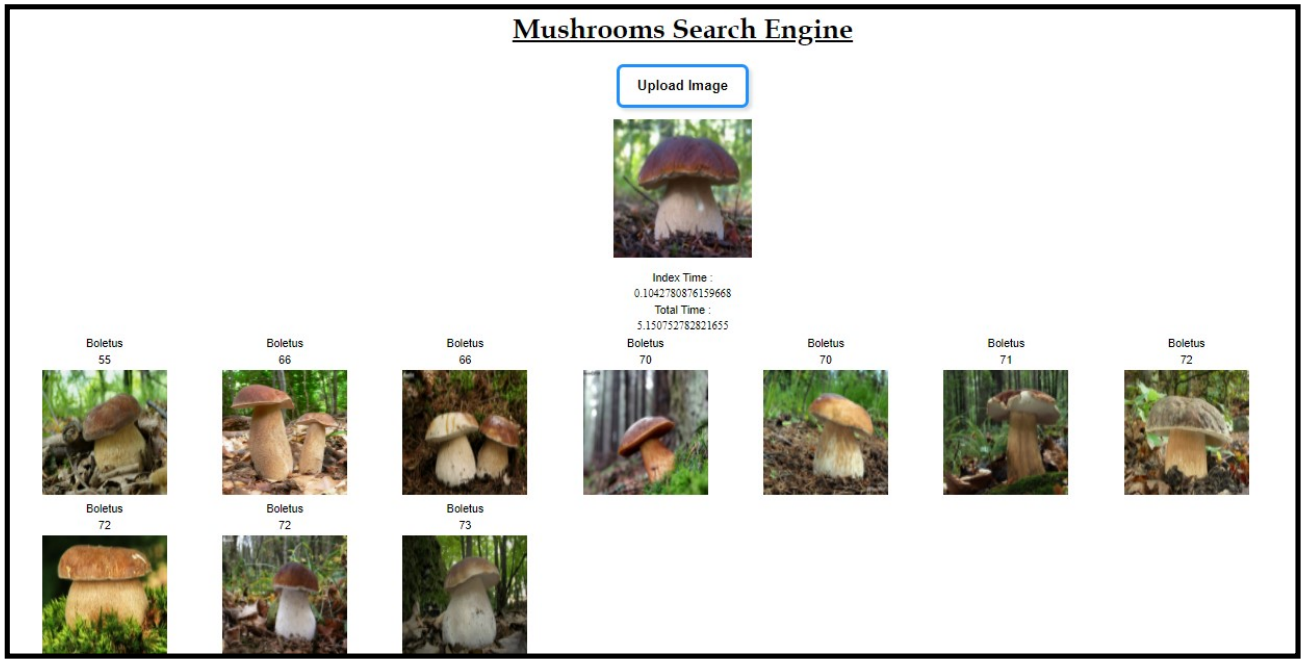

Figure 18: Web application shows the results

Conclusion

Analysing the results of the Web application, we can say that the performances of the entire system are very good. Indeed, despite the dataset that we used is quite small, the system can able to recognize in correct way the typology of the mushroom images submitted and their images more similar present on the database. However, there are some exceptions.

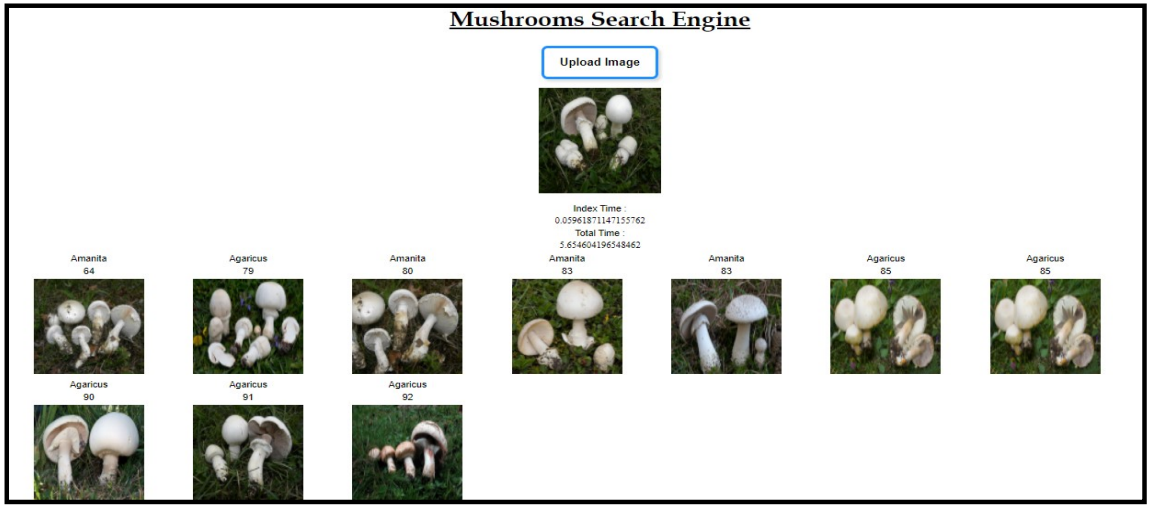

Figure 1

How we can see in the Figure 1, we loaded an image relative to a particular moment of the life of a mushroom of type Agaricus and the system returns us other different typology of mushrooms beyond the Agaricus typology that however are very similar to the query image. This probably happens because the MobileNet is not able to extract some feature of the Agaricus mushroom in this form that it is not present in the others typology of mushrooms retrieved. Probably a dataset with more images could be a solution to solve this type of problem in order to train the MobileNet in a better way.