MULTIMEDIA INFORMATION RETRIEVAL AND COMPUTER VISION PROJECT

Academic Year 2020/21

Barsellotti Luca, Bongiovanni Marco, Gholami Sina and Paolini Emilio

INDEX

INTRODUCTION

The aim of this project is to build a Web Search Engine based on features extracted from art images, which works efficiently thanks to a Vantage Point Tree Index implementation. The list of images that can be retrieved are contained in the “Art Dataset”, a dataset split into five classes (“Drawings”, “Engraving”, “Iconography”, “Paintings” and “Sculpture”), and in a “Distractor Dataset”, based on the Mirflickr25k dataset. For features extraction we used a Convolutional Neural Network. We use two models: one is based on a Pre-Trained Network (“InceptionV3”) with weights computed on “Imagenet” and the other one is a fine-tuned version of the same Pre-Trained Network through the “Art Dataset”. The VPtree is used to run efficiently k-NN and range queries on the features extracted.

DATASET ANALYSIS

TRAINING SET

First, we have analyzed the datasets. Both “Art Dataset” and “Distractor Dataset” contained some corrupted images and so we have removed them. At the end of the removing process, the training dataset contained in the “Art Dataset” was composed by

Total: 7721

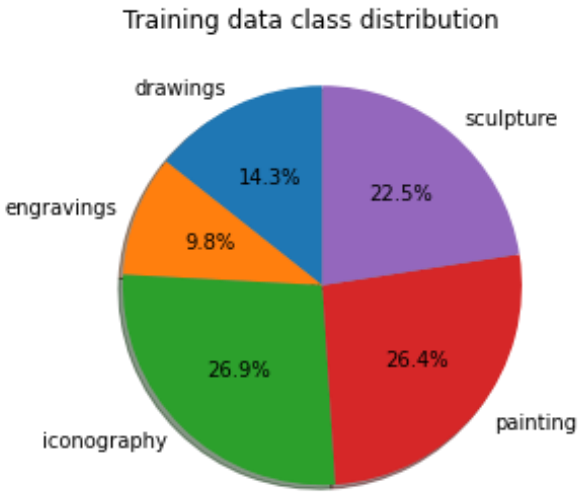
Drawings: 1107

Engraving: 757

Iconography: 2077

Painting: 2042

Sculpture: 1738



TEST SET

The provided dataset didn’t contain a test set. Hence, we used the validation set provided as the test set.

The test set included some replicated images in the training set, specifically inside the “Painting” class. So, after removing the duplicates, the test set was composed of

Total: 603

Drawings: 52

Engravings: 24

Iconography: 200

Painting: 206

Sculpture: 121

VANTAGE POINT TREE IMPLEMENTATION

INTRODUCTION

The Vantage Point Tree is an Exact Similarity Searching Method based on ball regions that recursively divide the given datasets. At each step, a vantage point *p* (also called pivot), belonging to the actual dataset *X*, is chosen. After that, the median m of all the distances from *p* of the points belonging to X is computed

$$m = median(d(x,p)) \quad \forall x \in X$$

and the dataset is split into two subsets according to:

$$S_1 = \{x \in X - \{p\} | d(x,p) \leq m\}$$

$$S_2 = \{x \in X - \{p\} | d(x,p) \geq m\}$$

At the end, the objects are stored into the leaves. The resulting Vantage Point Tree is a balanced binary tree. This approach requires a time cost of $O(n \log n)$ during the building phase and $O(\log n)$ in a 1-NN search.

BUILD

The Vantage Point Tree has been built using the approach described in the introduction, using as stopping condition the maximum bucket size that can be contained by a leaf. As the bucket size for the implemented index, we have chosen 300.

```
1.  def build(node, feature_subset, leaf=False):
2.      if leaf is True:
3.          node.set_leaf(feature_subset)
4.      else:
5.          pivot = random.choice(feature_subset)
6.          distances = list()
7.          for feature_schema in feature_subset:
8.              distances.append(feature_schema.distance_from(pivot))
9.          distances = np.array(distances)
10.         median = np.median(distances)
11.         subset1 = list()
12.         subset2 = list()
13.         for feature_schema in feature_subset:
14.             if feature_schema.distance_from(pivot) <= median:
15.                 subset1.append(feature_schema)
16.             else:
17.                 subset2.append(feature_schema)
18.         node.set_median(median)
19.         node.set_pivot(pivot)
20.         node.add_left(Node())
21.         node.add_right(Node())
22.         nodes += 2
23.         if len(subset1) <= bucket_size:
24.             build(node.get_left(), subset1, leaf=True)
25.         else:
26.             build(node.get_left(), subset1)
27.         if len(subset2) <= bucket_size:
28.             build(node.get_right(), subset2, leaf=True)
29.         else:
30.             build(node.get_right(), subset2)
```

RANGE SEARCH

During the Range Search, at each step the distance between the query object and the pivot is evaluated to decide if it is necessary to access the children nodes. The condition to access the left node is

$$d(q,p) - r \leq m$$

instead, the condition to access the right node is

$$d(q,p) + r \geq m$$

If the node considered in the step is a leaf, then all the distances from the objects contained in its bucket are evaluated to decide if they must be added to the result.

```
def range_search(node, query, range):
    if node.is_leaf() == True:
        for object in node.get_subset():
            if query.distance_from(object) <= range:
                range_search_return_list.append(object)
        return
    if query.distance_from(node.get_pivot()) <= range:
        range_search_return_list.append(node.get_pivot())
    if query.distance_from(node.get_pivot()) - range <= node.get_median():
        recursive_range_search(node.get_left(), query, range)
    if query.distance_from(node.get_pivot()) + range >= node.get_median():
        recursive_range_search(node.get_right(), query, range)
    return
```

K-NEAREST NEIGHBORS SEARCH

In the k-NN Search a Priority Queue is used to maintain the retrieved objects ordered by a distance. This Priority Queue is initialized with the first objects encountered while traversing the Vantage Point Tree. At each step, if the node is an internal node then the distance between the query and its pivot is computed. If this distance is lower than d_{MAX}, the current highest distance in the Priority Queue, the pivot is added to the queue. Hence, if

$$d(q,p) - d_{MAX} \leq m$$

then the k-NN Search is called on the left node and if

$$d(q,p) + d_{MAX} \geq m$$

Then the k-NN Search is called on the right node. Instead, if the node is a leaf, the distances from all the objects are computed to evaluate if they must be added to the Priority Queue.

```
def knn(node, query, k):
    if node.is_leaf() == True:
        for object in node.get_subset():
            distance = query.distance_from(object)
            if not nn.is_full():
                nn.push(priority=distance, item=object)
                d_max = distance if d_max < distance else d_max
            elif abs(distance) < abs(d_max):
                nn.pop()
                nn.push(priority=distance, item=object)
                tmp = nn.data[0][0]
                d_max = abs(tmp)
        return
    distance = query.distance_from(node.get_pivot())
    if not nn.is_full():
        nn.push(priority=distance, item=node.get_pivot())
        d_max = distance if d_max < distance else d_max
    elif abs(distance) < abs(d_max):
        nn.pop()
        nn.push(priority=distance, item=node.get_pivot())
        tmp = nn.data[0][0]
        d_max = abs(tmp)
    if distance - d_max <= node.get_median():
        knn(node.get_left(), query, k)
    if distance + d_max >= node.get_median():
        knn(node.get_right(), query, k)
    return
```

DISTANCE EVALUATION

The distance between two objects was evaluated through the opposite of the Cosine Similarity, namely

$$d(A,B) = 1 - similarity(A,B) = 1 - \cos(\theta) = 1 - \frac{A \cdot B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i}\sqrt{\sum_{i=1}^{n} B_i}}$$

We experimented also the Euclidean Distance, defined as

$$d(A,B) = \sqrt{\sum_{i=1}^{n} (B_i - A_i)^2}$$

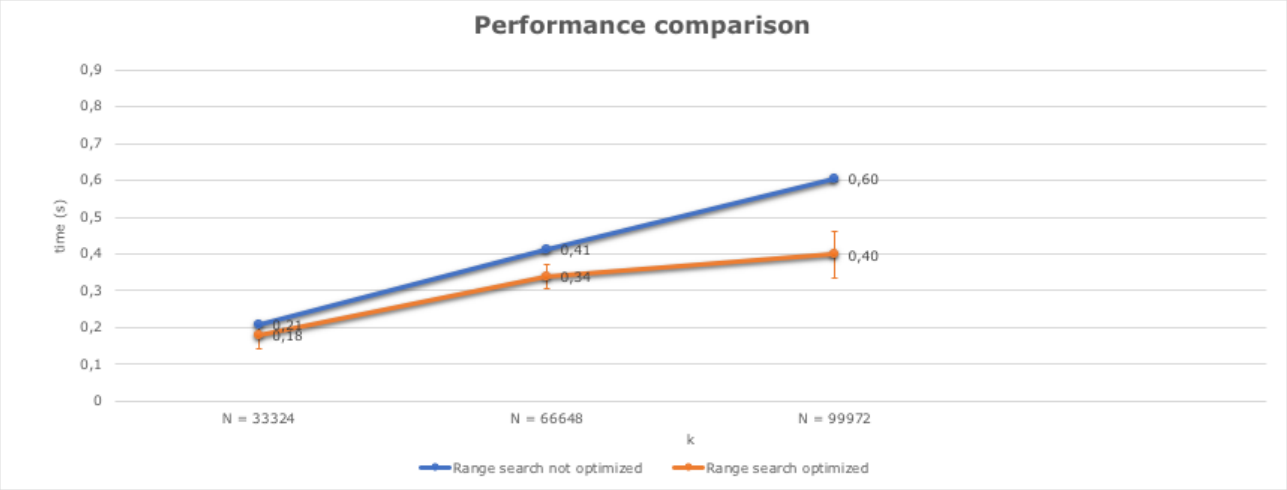
but the Cosine Similarity performed better during the searches.

TIME PERFORMANCE

Range Search with VPtree VS Range Search without index

In this section we have compared the results of executing a range search using the VPtree and the results obtained without using the index.

To evaluate the time performance of the index we run different simulations. Initially we set the length of the dataset to N=33324 and we run 10 simulations for the range search with VPTree and 10 simulations for the range search without index. Then we have repeated this for N=66648 and N=99972. Doing more simulations for each value of N allowed us to evaluate the confidence interval at 95%, so we can have a more reliable result. The experiments are reported in the figure below:



These results confirm that the index helps in improving the time performance for the executing range searches.

k-NN with vptree VS k-NN without index

In this comparison we have considered the k-NN Search on the built Vantage Point Tree using the algorithm presented before and a not optimized approach in which all the distances from the objects are computed and then sorted.

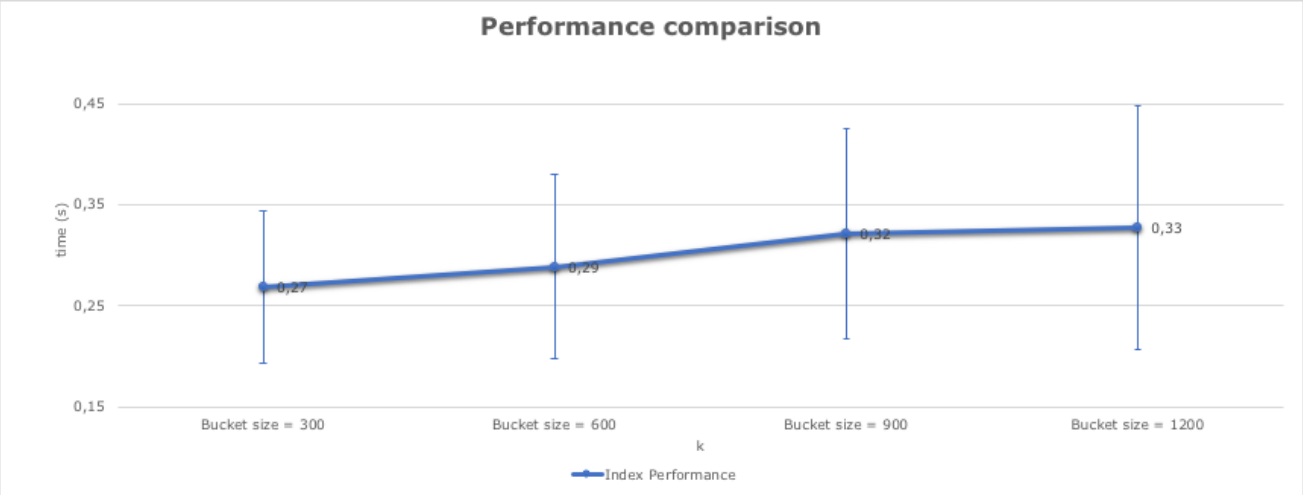
To evaluate the time performance of the index we run different simulations. Initially we set the length of the dataset to N=33324 and we run 10 simulations for the K-nn with VPTree and 10 simulations for the K-nn without index. Then we have repeated this for N=66648 and N=99972. Doing more simulations for each value of N allowed us to

evaluate the confidence interval at 95%, so we can have a more reliable result. The experiments are reported in the figure below:

As we can see, the k-NN optimized (the one using the VPtree) performs better and these empirical results confirm our initial thought: the index can help in improving the performance with respect to time.

k-NN with a small bucket size VS a big bucket size

Another aspect we considered during our tests was the bucket size (with bucket size we refer to the maximum number of elements that a leaf node can contain) of the VPtree, and specifically how it influences the time performance. To do this, we considered 10 queries and 4 possible bucket sizes: 300, 600, 900, 1200. The results obtained are reported in the figure below:

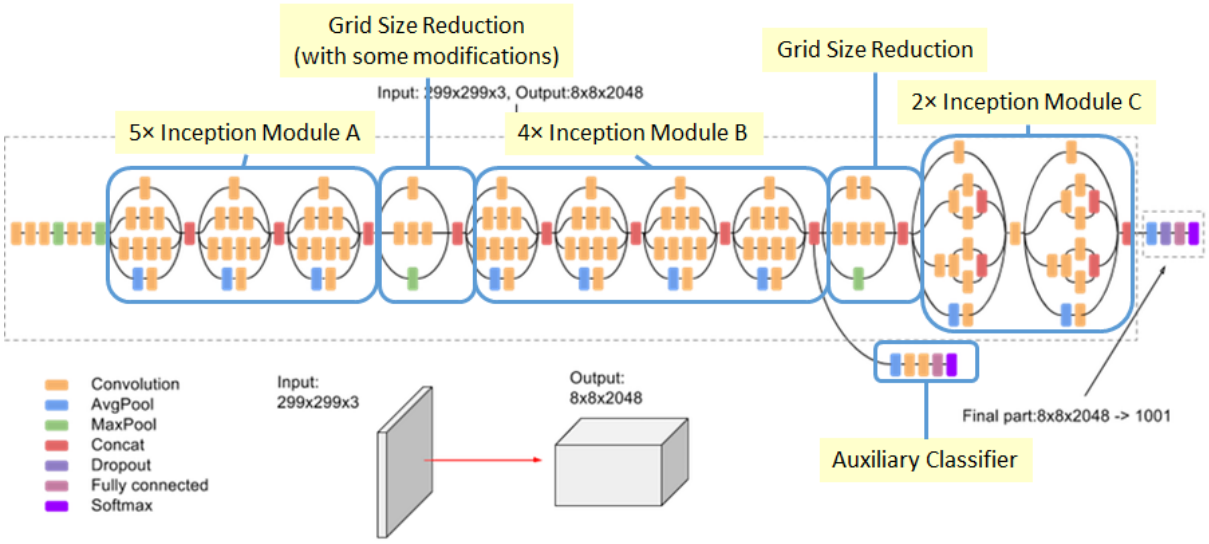


What we can get from this chart is that the bucket size doesn't influence performance so much. So, at the end we decided to go for bucket size equal to 300.

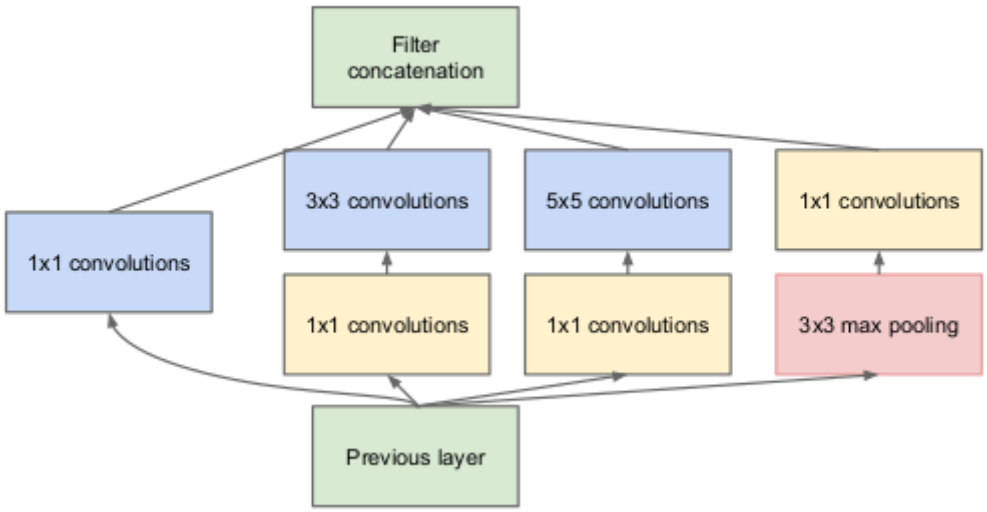
FEATURE EXTRACTION WITH A PRE-TRAINED NETWORK

STRUCTURE OF INCEPTION V3

Inception v3 is a Convolutional Neural Network model based on the paper "Rethinking the Inception Architecture for Computer Vision" presented by Szegedy et al. that showed an 78.1% accuracy on the ImageNet dataset. Its structure is composed of 42 layers split into different modules.



The main module and the core idea of the Inception model that made it one of the most used pre-trained models is the Inception Layer:



This layer is used in the Neural Network to select the filter size that will be relevant to learn the required information of the next layer, to learn the best weights and so automatically select the more useful features. This job is performed by the parallel Convolution Filters and the 1x1 Convolution Filters are used to perform dimensionality reduction.

FEATURE EXTRACTION WITH A FINE-TUNED NETWORK

STRUCTURE

```
_________________________________________________________________
Layer (type) Output Shape Param #
=================================================================
inception_v3 (Functional) (None, 5, 5, 2048) 21802784
_________________________________________________________________
gap (GlobalAveragePooling2D) (None, 2048) 0
```

```
_________________________________________________________________
dense (Dense) (None, 512) 1049088
_________________________________________________________________
dropout (Dropout) (None, 512) 0
_________________________________________________________________
last_relu (Dense) (None, 512) 262656
_________________________________________________________________
last_dropout (Dropout) (None, 512) 0
_________________________________________________________________
classifier_hidden (Dense) (None, 5) 2565
=================================================================
Total params: 23,117,093
Trainable params: 23,082,661
Non-trainable params: 34,432
_________________________________________________________________
```

TRAINING

We had to work with small dataset, so the network can easily overfit during training. To avoid this issue, we used several techniques: early stopping, class weighting and data augmentation.

1) **Data augmentation**: The objective is to augment images to train the network. To do this, we did the following:

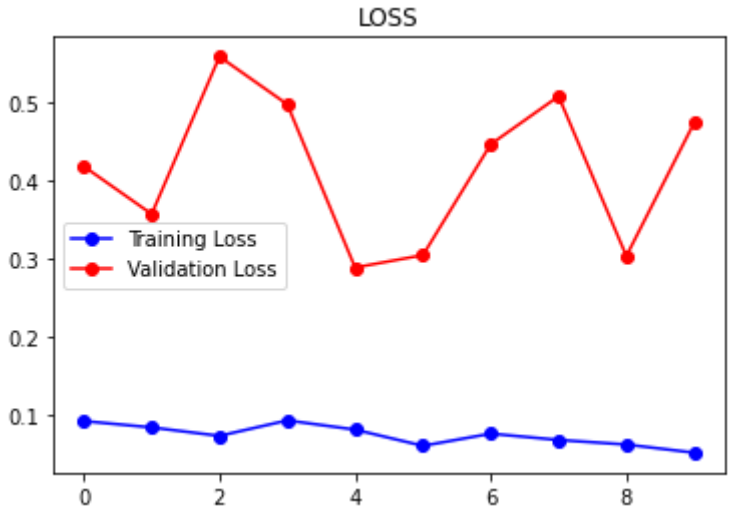
- `rotation_range` = 40, rotate images with random angles between –40 and 40 degrees.
- `width_shift_range` = 0.2, shift images horizontally by a random percentage between –20% and 20%.
- `height_shift_range` = 0.2, same but vertically.
- `shear_range` = 20, a shearing transformation with an angle of 20 degrees.
- `zoom_range` = 0.2, zoom images by a random percentage between –20% and 20%.
- `horizontal_flip` = True, randomly flip images horizontally.

2) **Class weighting**: The objective is to penalize the misclassification made by minority classes by setting higher weights for those classes and at the same time reducing weights for majority classes. To evaluate our weights, we did the following:

$$Class_i = \frac{N - |Class_i|}{N}, \quad N: Total\ number\ of\ elements$$

3) **Early stopping**: as stopping criteria we used validation loss, since it's a more reliable metric to halt training. The main reason is that accuracy merely accounts for the number of correct predictions while loss quantifies how certain the model is about a prediction. Moreover, we set the patience equal to 5, that is the maximum number of epochs that we have to wait before stopping if loss doesn't decrease.

This is shown in the previous plot, where we can see that we stop at epoch 9 since validation loss didn't decrease from the model in epoch 4.



CLASSIFICATION PERFORMANCE

After fine-tuning, we evaluated the results of the classifier using some common metrics:

Accuracy: 0.9237147569656372

Precision: 0.8402793040900282

Recall: 0.8503232871864823

F1 Score: 0.8419587479457562

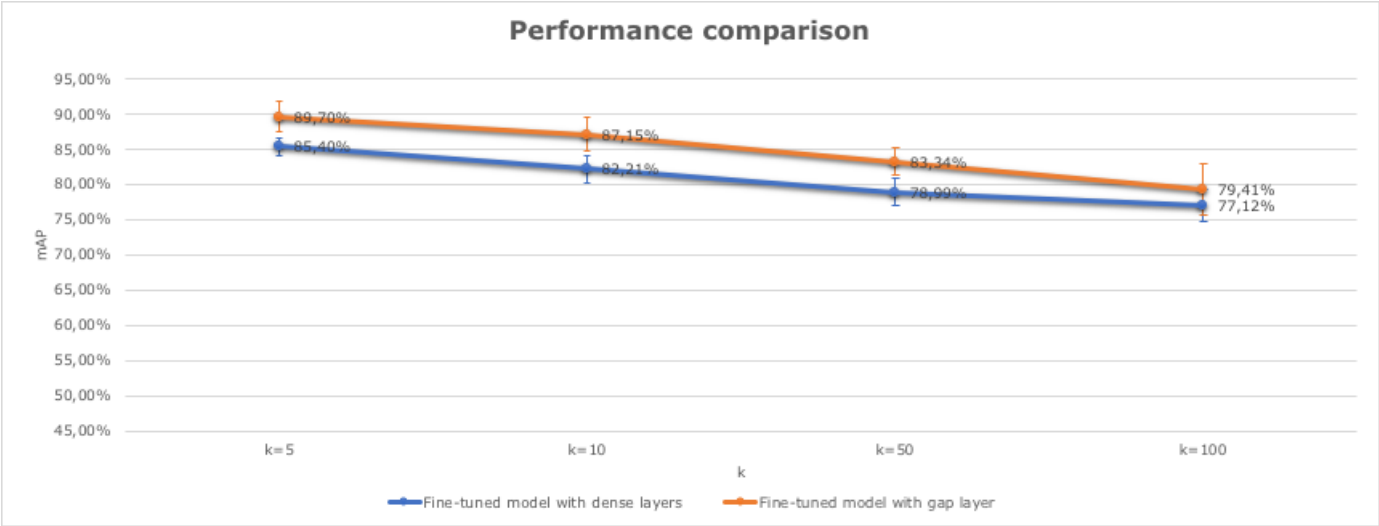
As we can see, the results can be considered good enough to go on with this model. In the next section we are going to split this model in two networks: one that stops at the gap layer (in the following we will refer to this model as gap model) and the other that stops at the last_dropout layer (in the following we will refer to this model as dense model).

This is done in order to perform features extraction and see which of the three models perform better (gap model, dense model, pre-trained model).

PERFORMANCE COMPARISON BETWEEN NETWORKS

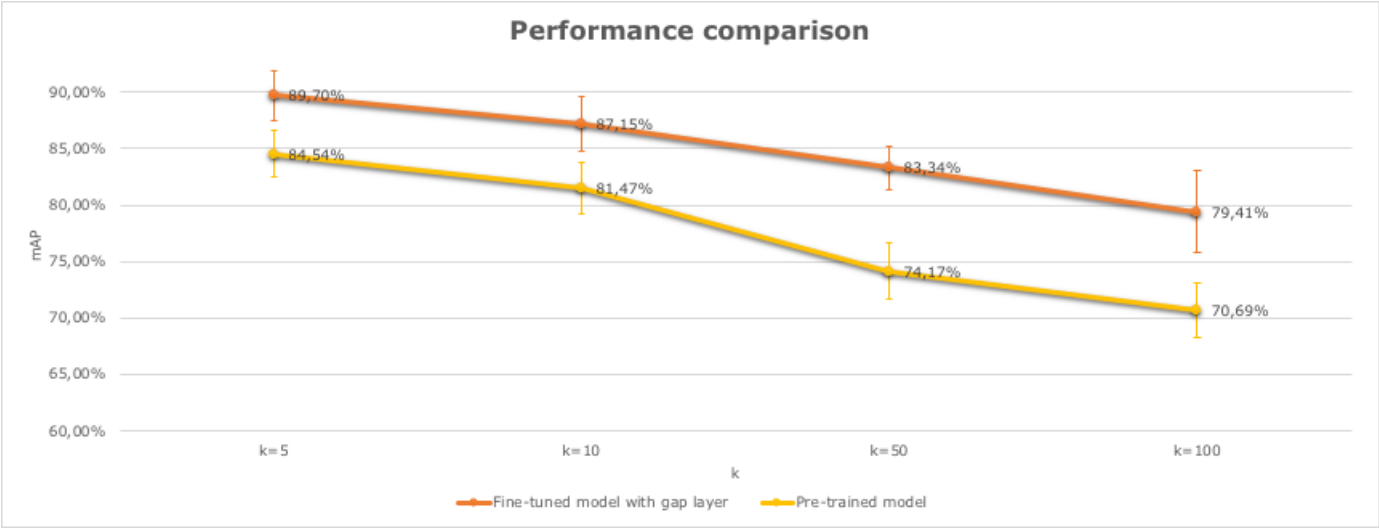
To evaluate the performances of the three models we used mean average precision (mAP) on K-nn queries. In particular, we ran 10 tests for each value of K (K=5,10,50,100). In each run we selected 50 random queries from the test dataset and evaluated the mAP.

First, we compared the two fine-tuned models:



The model with gap layer surely performs better for k=5 and k=10. When k increases to 50 and 100, the two models perform quite the same. But since in many real-world applications, we are not interested in such big values for K, we can say that the gap model is better.

Then, we compared the gap model with the pre-trained one:

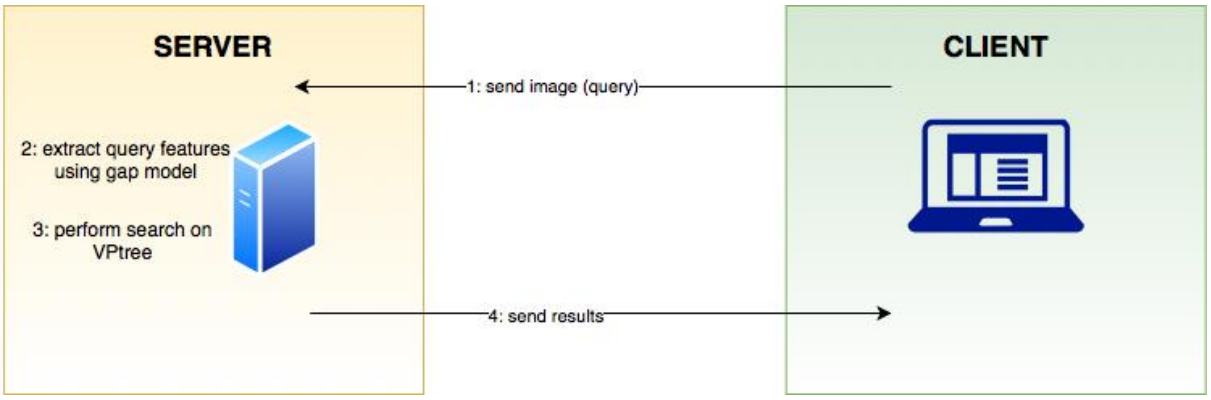


The gap model performs better (about 5/6% better for k=5 and k=10) than the pre-trained one. This was predictable, since we fine-tuned the pre-trained on our art images dataset.

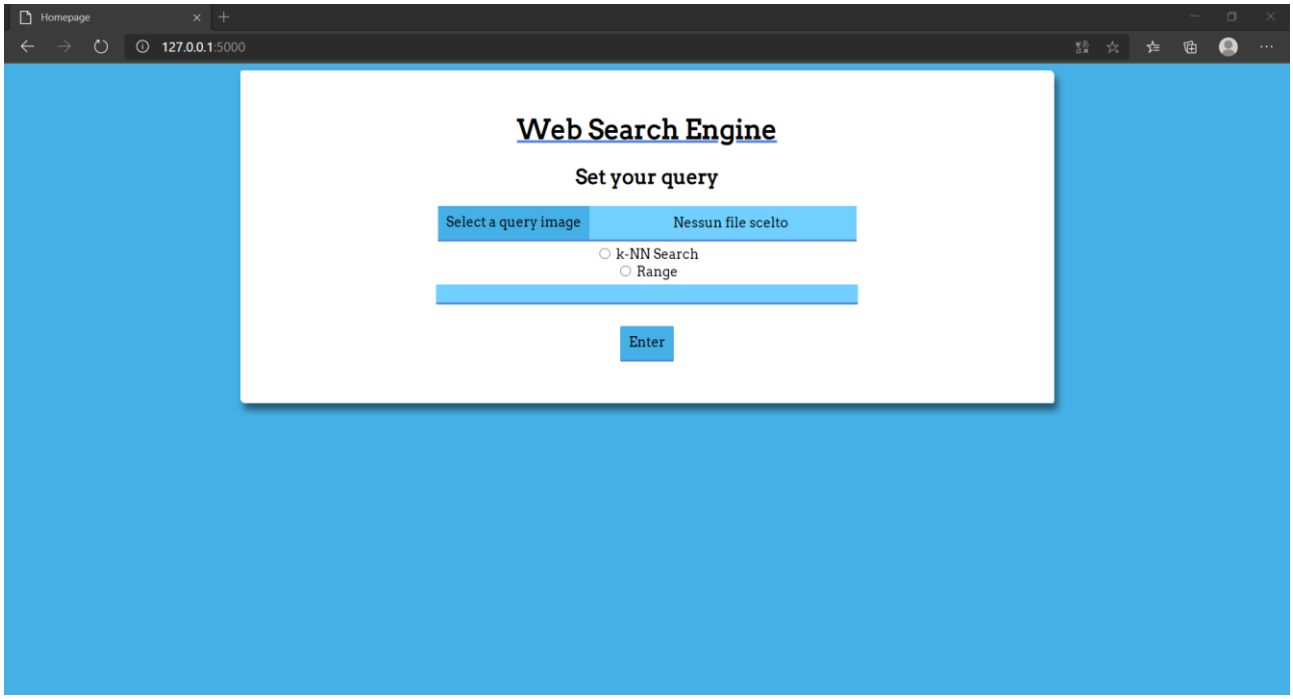
USER INTERFACE FOR WEB SEARCH ENGINE

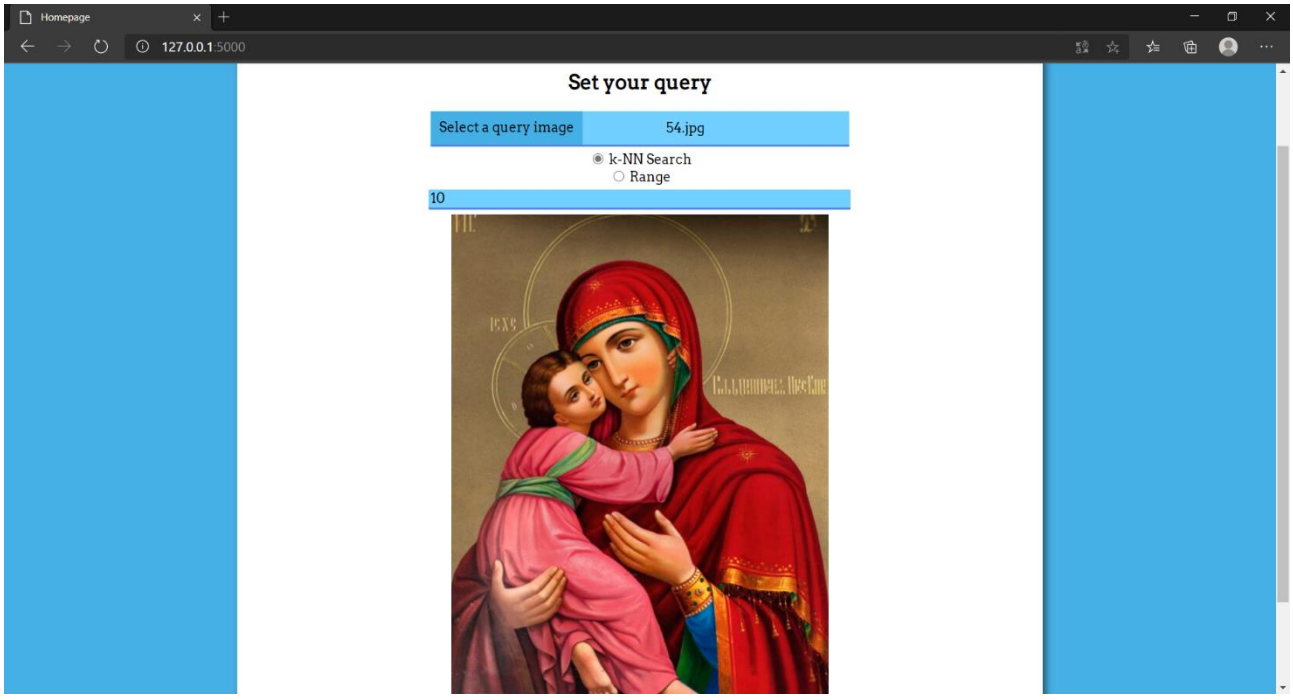
ARCHITECTURE OF THE WEB SEARCH ENGINE

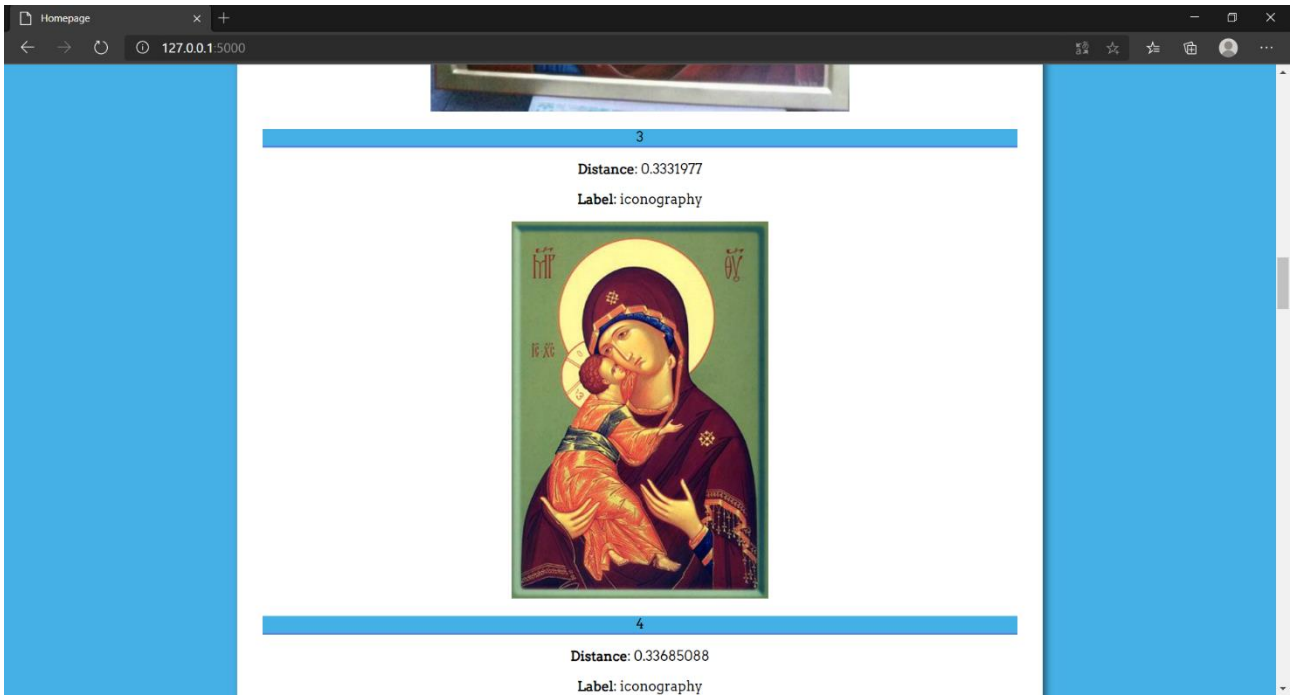
The web search engine is a client/server architecture. The server is built in python with flask, the client is a web interface that sends ajax requests to get the query responses.



SCREENSHOTS






Homepage
127.0.0.1:5000
3
Distance: 0.3331977
Label: iconography
4
Distance: 0.33685088
Label: iconography