Fruits and Vegetables

Multimedia Information Retrieval and Computer Vision final project
Prof. Giuseppe Amato
Prof. Claudio Gennaro
Prof. Fabrizio Falchi

Leonardo Turchetti
Lorenzo Tonelli
Gianluca Sirigu
Francesco Campilongo

Msc. in Artificial Intelligence and Data Engineering

April 21, 2022

Contents

1 Introduction

In this project we have created a search engine for fruit and vegetables. By providing an image of fruit or vegetables as input, the system returns a set of similar images.

The search engine can be structured in two macro areas: the first relating to the generation of features and the second relating to the indexing of the features.

We used the pretrained neural network InceptionV3 developed by Google by omitting its classifier. The network was trained following a finetuned approach. The best finetuned inceptionV3 network was used to generate the features.

The features have been indexed according to the VPT index, a tree data structure that allows you to perform queries in a more efficient way than the exhaustive query of the dataset.

Furthermore, the project reports experiments to measure the goodness of the result set obtained, to compare the performance of the VPT with respect to the brutal method.

A web app was created to graphically illustrate the search engine developed with its features and to demonstrate the difference between a finetuned network and a base one.

2 Preprocessing

The dataset used for this work was taken from Kaggle[1]. This dataset contains images of the following food items:

- **fruits**: banana, apple, pear, grapes, orange, kiwi, watermelon, pomegranate, pineapple, mango;
- **vegetables**: cucumber, carrot, onion, potato, lemon, tomato, raddish, beetroot, cabbage, lettuce, spinach, soy bean, cauliflower, bell pepper, chilli pepper, turnip, corn, sweet potato, jalepeño, ginger, garlic, peas, eggplant.

The original dataset structure consisted of three folders:

- `train` (100 images each);
- `test` (10 images each);
- `validation` (10 images each);

each of the above folders contains subfolders for different fruits and vegetables where in the images for respective food items are present.

```
train
├── apple
│   ├── img_1.jpg
│   ├── img_2.jpg
│   ├── img_3.jpg
│   └── ...
├── banana
│   ├── img_1.jpg
│   ├── img_2.jpg
│   ├── img_3.jpg
│   └── ...
├── beetroot
├── carrot
└── ...
test
├── apple
│   ├── img_1.jpg
│   ├── img_2.jpg
│   ├── img_3.jpg
│   └── ...
└── banana
    └── img_1.jpg
```

[1] `https://www.kaggle.com/kritikseth/fruit-and-vegetable-image-recognition`

```
    │   ├── img_2.jpg
    │   ├── img_3.jpg
    │   └── ...
    ├── beetroot
    ├── carrot
    └── ...
└── validation
    ├── apple
    ├── banana
    ├── beetroot
    ├── carrot
    └── ...
```

The images in the dataset were originally scraped from Bing Image Search.

After starting to do some preliminary tests, building the first scratch classifiers, we began to find very high anomalous values of Testing and Validation accuracy in the results. We have therefore started a very thorough data exploration phase of the dataset. From this operation a series of anomalies and distortions emerged that we went to correct in the preprocessing phase.

2.1 Duplicate Images

The first thing we found is that in the original structure of the dataset there were duplicates between the images present in the training set and those present in the validation and in the test set.

Figure 1: Example of duplicated images between train, test and validation.

So first of all we have unified training, testing and validation by only preserving the labels of each type of fruit and vegetable. The initial dataset was then transformed into the following structure:

```
├── apple
│   ├── img_1.jpg
│   ├── img_2.jpg
│   ├── img_3.jpg
│   └── ...
└── banana
```

```
    ├── img_1.jpg
    ├── img_2.jpg
    ├── img_3.jpg
    ├── ...
├── beetroot
    ├── img_1.jpg
    ├── img_2.jpg
    ├── img_3.jpg
    ├── ...
├── carrot
├── garlic
├── ginger
├── kiwi
├── lemon
├── ...
```

Using Python `cv2` module we identified images with the same exact content but different file names; such images were deleted.

2.2 Manual Cleanup

Another problem that came up was that some many of the images found in the directories were not related to the class label but accidentally ended up in the dataset as a result of the web scraping procedure used to collect images from Bing Images; after many attempts of writing a functioning interactive script to visualize and delete such images, the choice was made to perform this task manually since the Jupiter notebook does not provide the level of interaction required in order to progressively visualize all images in the dataset and decide which ones to keep and which ones to delete.

Figure 2: Examples of wrong images from apple and ginger.

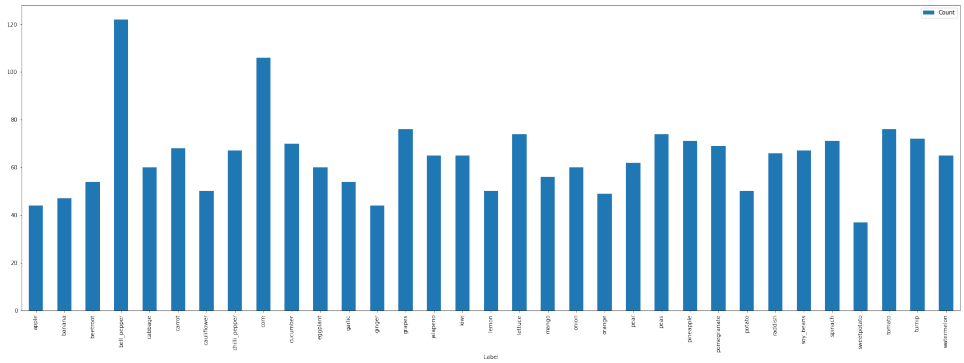
Figure 3: Examples of wrong images from jalapeno and potato.

The manual cleanup was conducted with the following criteria in mind:

- images containing dishes and meals based on the class label fruit or vegetable were deleted;
- images containing fields of the given fruit or vegetable were deleted;
- images completely unrelated with the class label were deleted;

2.3 Distribution of the Preprocess Dataset

After the Preprocessing phase the classes distribution of this dataset resulted as follows:



3 Deep Learning Phase

This section describe the use of the pre-trained Network InceptionV3 to build two different Fine-Tuned models. The structure of the InceptionV3 is shown below

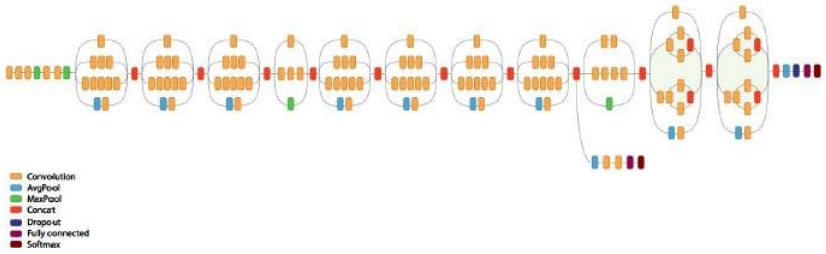


Figure 4: InceptionV3 Architecture

3.1 Fine Tuning

We want to build two Fine-Tuned models starting from the pre-trained network InceptionV3. All of these models are trained using the optimizer Rmsprop with the default "learning rate" (0.001), the loss function "categorical_crossentropy", 10 epochs, "batch size" = 36 and a resize of the input images of 299x299. The dataset was divided as follows: the Test Set was obtained using the "test_size" attribute (with value 0.1) of the "train_test_split" function, and the Validation Set using the "validation_split" attribute (with value 0.1) of the "fit" function.

3.1.1 Data Augmentation

In order to try to improve the Overfitting problem during the training of the different models we introduce the concept of Augmentation for generate more training data from existing training samples, by "augmenting" the samples via a number of random transformations that yield believable-looking images. The code of the layer to insert the augmentation inside the models to train is shown below.

```
data_augmentation = keras.Sequential(
  [
  layers.RandomFlip("horizontal"),
  layers.RandomRotation(0.3),
  layers.RandomZoom(0.2),
  ]
)
```

Figure 5: Data Augmentation code

3.1.2 Base Model

In this first simple model we combined the pre-trained network with a fully connected net with a Global Average Pooling layer, a Flatten layer, a Dense layer of 256 and using the Dropout method with value 0.2 in order to reduce OverFitting. In this model we will initially freeze all the layers of the pre-trained network and train only the new layers added to build the fruit/vegetables classifier. Here we show the summary of the Base Model.

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 299, 299, 3)]     0

 sequential (Sequential)     (None, 299, 299, 3)       0

 tf.math.truediv (TFOpLambda  (None, 299, 299, 3)      0
 )

 tf.math.subtract (TFOpLambd  (None, 299, 299, 3)      0
 a)

 inception_v3 (Functional)   (None, 8, 8, 2048)        21802784

 global_average_pooling2d (G  (None, 2048)             0
 lobalAveragePooling2D)

 flatten (Flatten)           (None, 2048)              0

 dense (Dense)               (None, 256)               524544

 dropout (Dropout)           (None, 256)               0

 dense_1 (Dense)             (None, 33)                8481

=================================================================
Total params: 22,335,809
Trainable params: 533,025
Non-trainable params: 21,802,784
_________________________________________________________________
```

Figure 6: Base Model Summary

As we can see from the summary all the layers of the pre-trained network are freezed and the only weights trainable are from the two dense layer that we add.

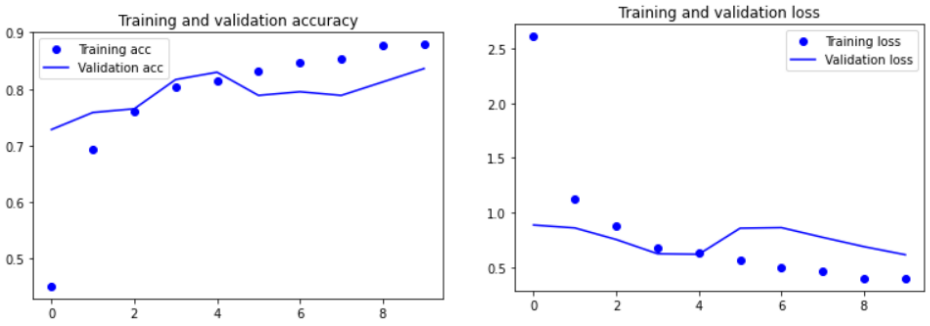


Figure 7: Base Model Accuracy and Loss

Weights Unfreezed	Testing Accuracy	Precision	Recall	F1-Score
0	0.853	0.888	0.852	0.850

Table 1: Metrics Base Model

The value of Testing Accuracy of 0.853 is high, but with the fine-tuning we try to improve this value. The Confusion Matrix is quite homogenous and the model as no particular difficult to recognise any particular type of Fruit and Vegetables compared to others. The Confusion Matrix is shown below.

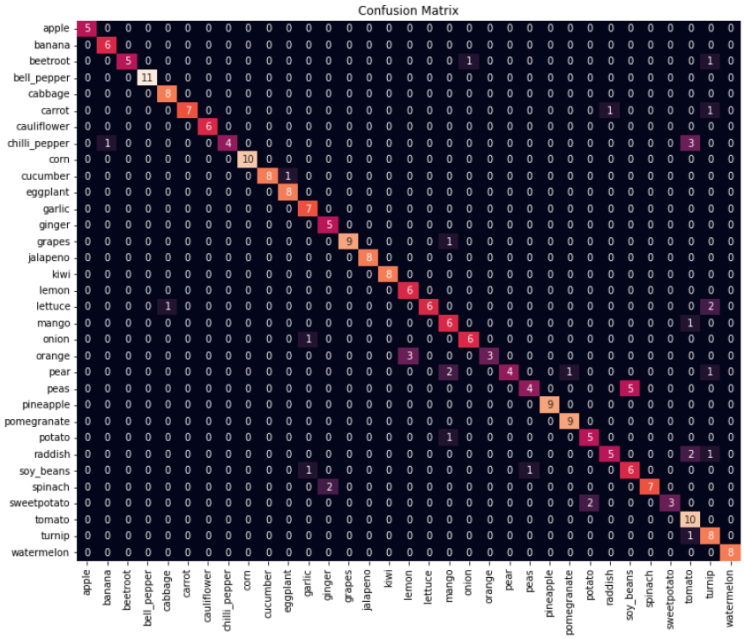


Figure 8: Confusion Matrix Base Model

3.1.3 Fine-Tuned Model 1

To fine-tune the network and so to build the second model we unfrozen the last block of InceptionV3 before the training, and we re-perform training on the training set with the same configuration of the Base Model.

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 299, 299, 3)]     0

 sequential (Sequential)     (None, 299, 299, 3)       0

 tf.math.truediv (TFOpLambda  (None, 299, 299, 3)      0
 )

 tf.math.subtract (TFOpLambd  (None, 299, 299, 3)      0
 a)

 inception_v3 (Functional)   (None, 8, 8, 2048)        21802784

 global_average_pooling2d (G  (None, 2048)             0
 lobalAveragePooling2D)

 flatten (Flatten)           (None, 2048)              0

 dense (Dense)               (None, 256)               524544

 dropout (Dropout)           (None, 256)               0

 dense_1 (Dense)             (None, 33)                8481

=================================================================
Total params: 22,335,809
Trainable params: 926,753
Non-trainable params: 21,409,056
_________________________________________________________________
```

Figure 9: Fine-Tuned Model 1 Summary

As we can see from the summary not all the layers of the pre-trained network are freezed and the weights trainable are from the two dense layer and from the pre-trained network.

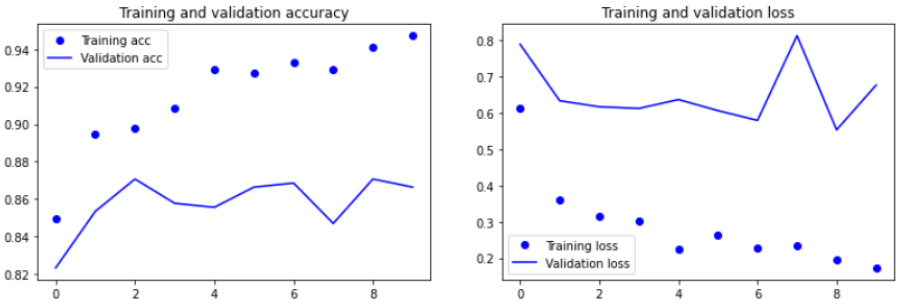


Figure 10: Fine-Tuned Model 1 Accuracy and Loss

Weights Unfreezed	**Testing Accuracy**	**Precision**	**Recall**	**F1-Score**
393728	0.903	0.921	0.903	0.899

Table 2: Metrics Fine-Tuned Model 1

As we can see from the table the value of Testing Accuracy is increased to 0.903, so the fine-tuning take the benefits that we expect. The Confusion Matrix is more homogenous than the previous model. The Confusion Matrix is shown below.

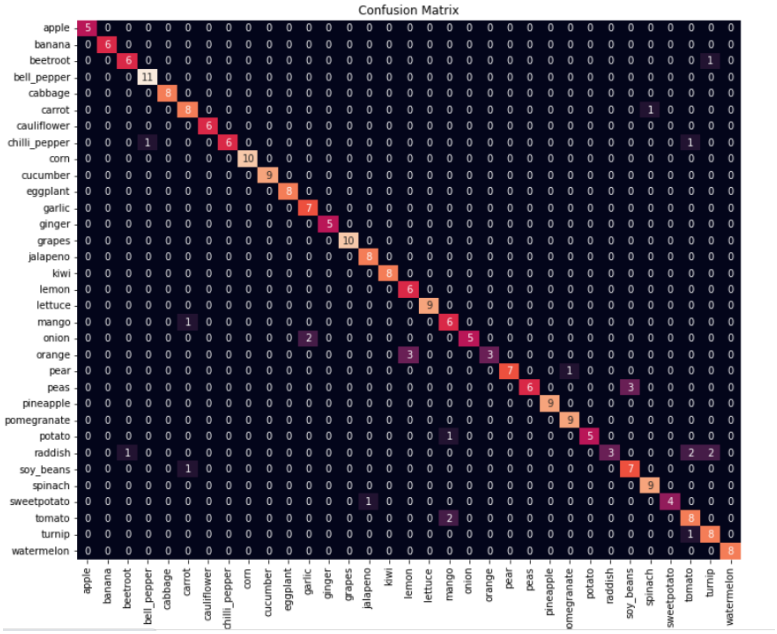


Figure 11: Confusion Matrix Fine-Tuned Model 1

3.1.4 Fine-Tuned Model 2

At the end we decide to do another test unfreezing more block in the pre-trained network. The configuration of the model remain the same as before.

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 299, 299, 3)]     0

 sequential (Sequential)     (None, 299, 299, 3)       0

 tf.math.truediv (TFOpLambda  (None, 299, 299, 3)      0
 )

 tf.math.subtract (TFOpLambd  (None, 299, 299, 3)      0
 a)

 inception_v3 (Functional)   (None, 8, 8, 2048)        21802784

 global_average_pooling2d (G  (None, 2048)             0
 lobalAveragePooling2D)

 flatten (Flatten)           (None, 2048)              0

 dense (Dense)               (None, 256)               524544

 dropout (Dropout)           (None, 256)               0

 dense_1 (Dense)             (None, 33)                8481

=================================================================
Total params: 22,335,809
Trainable params: 3,353,121
Non-trainable params: 18,982,688
```

Figure 12: Fine-Tuned Model 2 Summary

As we can see from the summary now more layers of the pre-trained network are unfreezed and the weights trainable are from the two dense layer and from the pre-trained network.

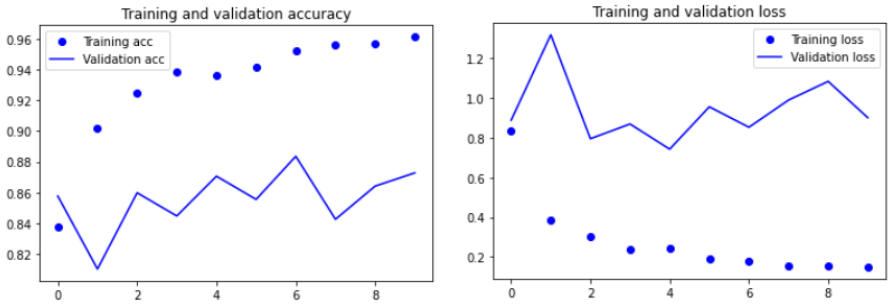


Figure 13: Fine-Tuned Model 2 Accuracy and Loss

Weights Unfreezed	**Testing Accuracy**	**Precision**	**Recall**	**F1-Score**
2820096	0.895	0.909	0.895	0.889

Table 3: Metrics Fine-Tuned Model 2

As we can see from the table the value of Testing Accuracy has a little decresing to 0.895, so this fine-tuning test don't take the benefits that we expect. The Confusion Matrix is very similar to the previous model. The Confusion Matrix is shown below.

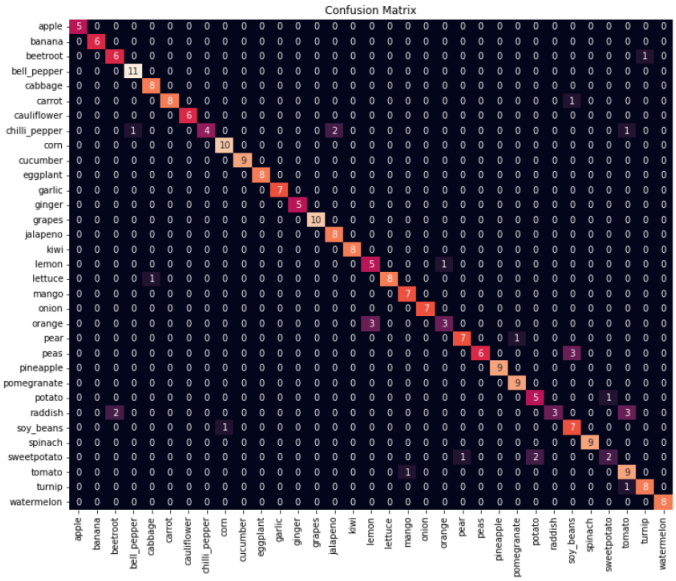


Figure 14: Confusion Matrix Fine-Tuned Model 2

After all these experiment we decide to take the first fine-tuned model to extract the features from the dataset comparing the result with only the pre-trained network.

4 VPT Index

4.1 Creation

The Vantage Point Tree(VPT) index is an exact similarity searching method that exploits the ball partitioning technique dividing recursively tha dataset of interest. During the construction of the VPTree, at each iteration, we choose as pivot, also called Vantage Point, a point of the dataset and then we compute the median m respect to the pivot. The median m represents the radious that splits the points in two equal parts. The stopping condition is the number of points in a subset to split, if it is less than a given threshold, all the points of that subset are put into one leaf.

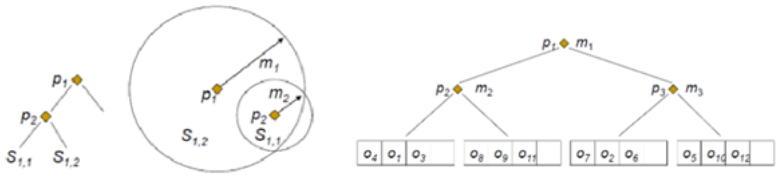


Figure 15: Ball Partitioning and a example of Vantage Point Tree.

We have two other structures that help us when using the VP Tree, a stack to store the results and a Node class that represents the nodes in the tree.

```
def build(self, node, feature_subset):
    pivot = random.choice(feature_subset)

    #In distances computations we have to exclude the Vantage Point
    feature_subset.remove(pivot)
    distances = []

    #Compute distances to the chosen pivot from the other points and take the median
    for feature in feature_subset:
        dist = distanceOfTwoPoints(feature,pivot,self.distanceMetric)
        distances.append(abs(dist))

    distances = np.array(distances)
    median = np.median(distances)

    subset1 = []
    subset2 = []

    #Split the remaining features in accoring to computed median
    for feature in feature_subset:
        dist = distanceOfTwoPoints(feature,pivot,self.distanceMetric)
        dist = abs(dist)
        if dist <= median:
            subset1.append(feature)
        else:
            subset2.append(feature)

    node.median = median
    node.pivot = pivot
    node.left = Node()
    node.right = Node()
    self.internalNodes += 1
    self.nodes += 1

    #If a left/right branch has less than bucket_size elements, then it's a leaf
    if len(subset1) <= self.bucket_size:
        node.left.subset = subset1
        node.left.leaf = True
        self.leafNodes += 1
        self.nodes += 1
    else:
        self.build(node.left, subset1)

    if len(subset2) <= self.bucket_size:
        node.right.subset = subset2
        node.right.leaf = True
        self.leafNodes += 1
        self.nodes += 1
    else:
        self.build(node.right, subset2)
```

As stopping condition we have the bucket dimension that act as a parameter which controls the leaves size. We tried various values and the results will be discussed further.

The functions available to query our tree are two, one for the NN-query and one for the range search.

4.2 K-NN query

```
def knn(self, query, k):
    self.knn = Stack(k)
    self.d_max = math.inf
    self.visited = 0
    self.recursive_knn(self.root, query)
    results = []

    for elem in self.knn.data:
        results.append((elem[0],elem[1]))
    return results

def recursive_knn(self,node,query):
    if node.leaf == True:
        for point in node.subset:
            self.visited += 1
            distance = distanceOfTwoPoints(query,point,self.distanceMetric)
            distance = abs(distance)
            if not self.knn.isFull():
                self.knn.addElement(distance,point)
                self.d_max = self.knn.update_d_max()
            elif distance < self.d_max:
                self.knn.addElement(distance,point)
                #remove last element inside the stack
                self.d_max = self.knn.update_d_max()

        return

    self.visited += 1
    distance = distanceOfTwoPoints(query,node.pivot,self.distanceMetric)
    distance = abs(distance)
    if not self.knn.isFull():
        self.knn.addElement(distance,node.pivot)
        self.d_max = self.knn.update_d_max()
    elif distance < self.d_max:
        self.knn.addElement(distance,node.pivot)
        #remove last element inside the stack
        self.d_max = self.knn.update_d_max()

    if distance - self.d_max <= node.median:
        self.recursive_knn(node.left, query)
    if distance + self.d_max >= node.median:
        self.recursive_knn(node.right, query)
    return
```

Invoking knn first we prepare a stack to receive the elements returned by the query; the recursive_knn function is called on the root node that will explore the whole tree and at the end of the exploration we will return the requested elements. For each node that we are going to explore the first thing we do if it is not a leaf is to calculate the distance between the query and the pivot. If there is space in the stack we add it regardless, if it is full we add it only if the distance just calculated is better than the maximum distance in the stack. At this point, by checking the query-pivot distance, we can see whether to descend in the right or left node to continue the query. When we reach a leaf, we analyze each point and add to the stack a point with the same criteria used for the pivot (addition if there is space in the

stack or because the distance is less than that of the k-nn)

4.3 Range query

```
def range_search(self,query,range):
    self.feature_list = []
    self.recursive_range_search(self.root,query,range)
    return self.feature_list

def recursive_range_search(self,node,query,range):
    if node.leaf == True:

        for point in node.subset:
            dist = distanceOfTwoPoints(query,point,self.distanceMetric)
            dist = abs(dist)
            if(dist <= range):
                tmp = []
                tmp.append(point)
                tmp.append(dist)
                self.feature_list.append(tmp)
        return

    #We insert the pivot if it is near to the query
    dist = distanceOfTwoPoints(query,node.pivot,self.distanceMetric)
    dist = abs(dist)
    if(dist <= range):
        tmp = []
        tmp.append(node.pivot)
        tmp.append(dist)
        self.feature_list.append(tmp)


    if(dist - range <= node.median):
        self.recursive_range_search(node.left, query, range)
    if(dist + range >= node.median):
        self.recursive_range_search(node.right, query, range)
    return
```

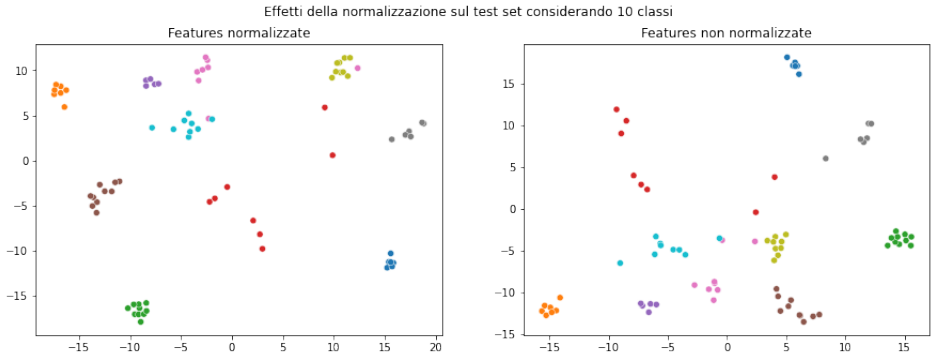
Through range_search we access to the root of the tree which will be recursively explored. If the distance between the query and the pivot of the node where we are is less than the specified range, then we add the pair pivot - distance to a list. Then we explore all other elements of the node: if the range is less than the median of the node, we explore the left side, otherwise we go to the right side. If we have arrived at a leaf, for each element of it we check if there are points that satisfy the query and if so we add them to the list to return.

5 Performance Evaluation

We will analyze the performance of indexing with VP Tree from the aspects of precision and execution time. Accuracy will be evaluated for non-normalized, normalized, and normalized features with in addition PCA. The execution times compared instead will be those of the bruteforce approach, VP Tree with random pivot, VP Tree with pivot chosen by outlier detection.

5.1 Normalization

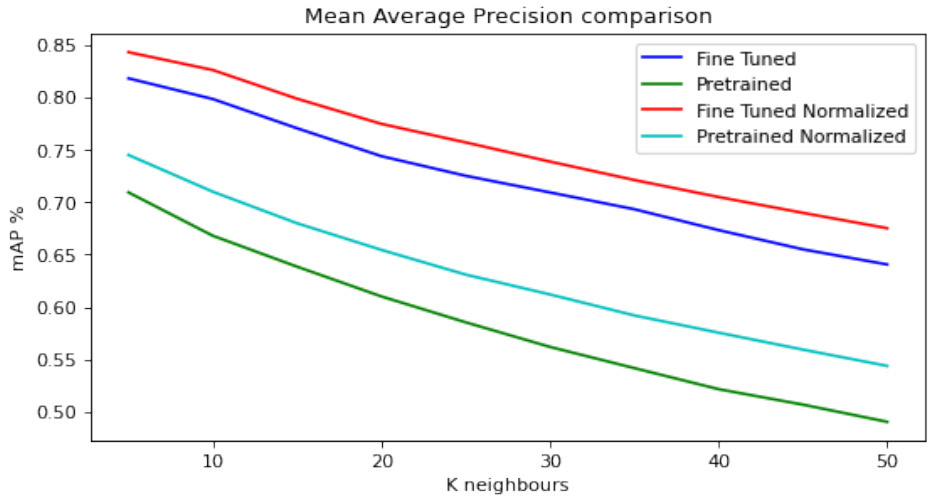
We want to see how feature normalization infuences model accuracy. We apply a dimensionality reduction using the t-SNE algorithm and see how the 2D representation of a feature sample changes.



As we can see in the transformed space it seems that some classes are more clustered in the normalized version, but it's not a rule and to the eye it doesn't look like there are big differences. Let's see how it changes regarding precision.

5.2 Model Precision

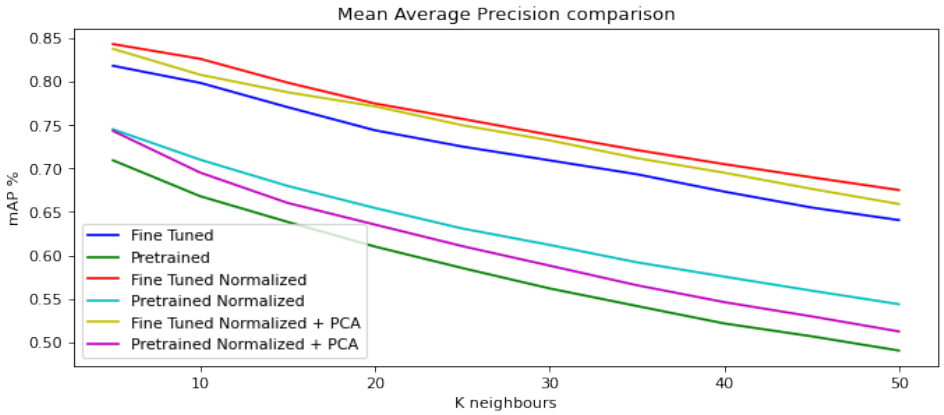
In order to evaluate the precision of our model we evaluated the MAP for many k's in the K-NN query; thanks to MAP we can get an overview of the model's quality when recovering images with the same label as the query image.
Preliminarily, we extracted two test sets (one from the features of the fine tuned network and one from the features of the non-optimized one). The images that are part of the test set (about one percent of the total) have not been indexed and do not contain elements from the distractor.



As we might expect, the average accuracy decreases as the number of objects retrieved from the index increases. The difference in performance between the pretrained and the optimized network is immediately apparent. In particular we see that the normalized version, in both cases, systematically gains percentage points.

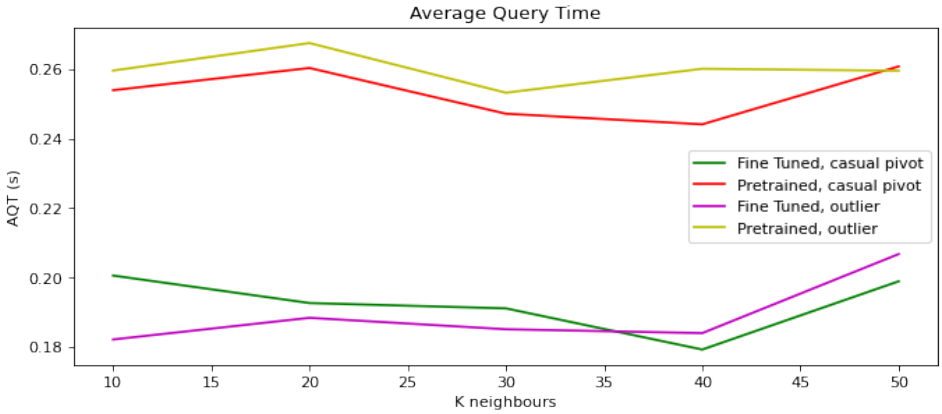
5.2.1 PCA

We decided to investigate the effect of a PCA on the features used, to evaluate possible tradeoffs between a loss of precision and a gain in terms of saved calculations. The PCA algorithm was applied maintaining a variance of 95% with respect to the original data distribution. In the case of the pretrained features, the size was decreased from 2048 to 833, while for the fine tuned features it was halved, from 256 to 128. We see the difference in mAP compared to the full features.

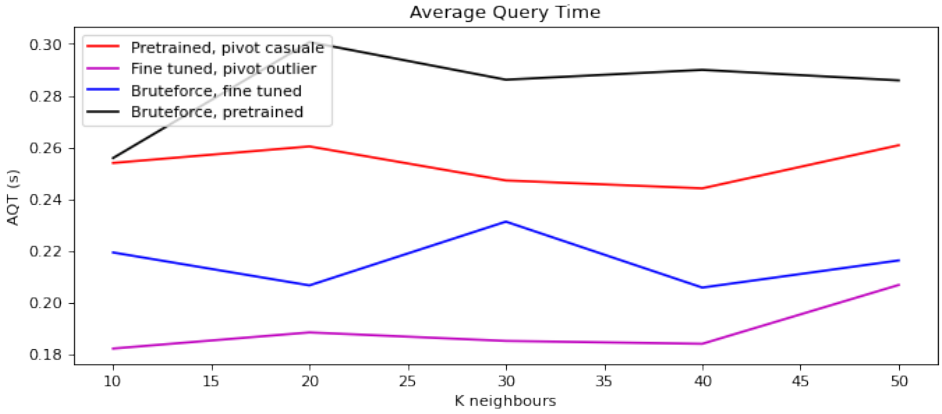


5.3 Index performance

Now we are interested to understand the course of the times of the queries for various configurations of our model. In particular way we will make a comparison not only between the two types of features, but also between two types of trees: the one with the choice of the random pivot and the one with the choice based on the calculation of the outlier. Finally we will review these results compared with the brute force approach.



Let's take pretrained features. As we can see, contrary to what we might have expected, the tree with the random pivot choice behaves better than its counterpart. In the case of the fine tuned features there are no clear differences. The times reported on the graph, for each curve, are the result of the average of the times of 10 trees on 15 queries each.



Comparing two trees to their respective bruteforces, we see that the VPTree approach is better than the bruteforce approach in all cases. The bruteforce times were averaged over 30 queries for each k.

6 Web Interface

A simple web interface is developed to perform an image similarity search among a Fruit and Vegetables dataset and a Distractor dataset.
The image is uploaded on the web interface, and then using the chosen tuned model to extract feature and the creation of the VPT index earlier discussed, twelve of the most similar images are returned and displayed on the web search interface.

6.1 Web Application Structure

The application is characterized by two part:

- REST Server written in python through flask and deployed on Google Colab using ngrok.
- The web client, written in HTML, CSS and Javascript.

All the file necessary for this web app to work are into the webapp folder into the google drive shared.

- webapp.py actual code for the REST server
- utils.py support code for the webapp.py
- VPT.py code that implements the Vantage Point Tree
- point.py class that implements a feature point
- RESTserver.ipynb the notebook that deploys the REST server on Google Colab, by installing flask and flask-ngrok on the virtual machine, and installing the ngrok agent on the virtual machine and let start the webapp.py

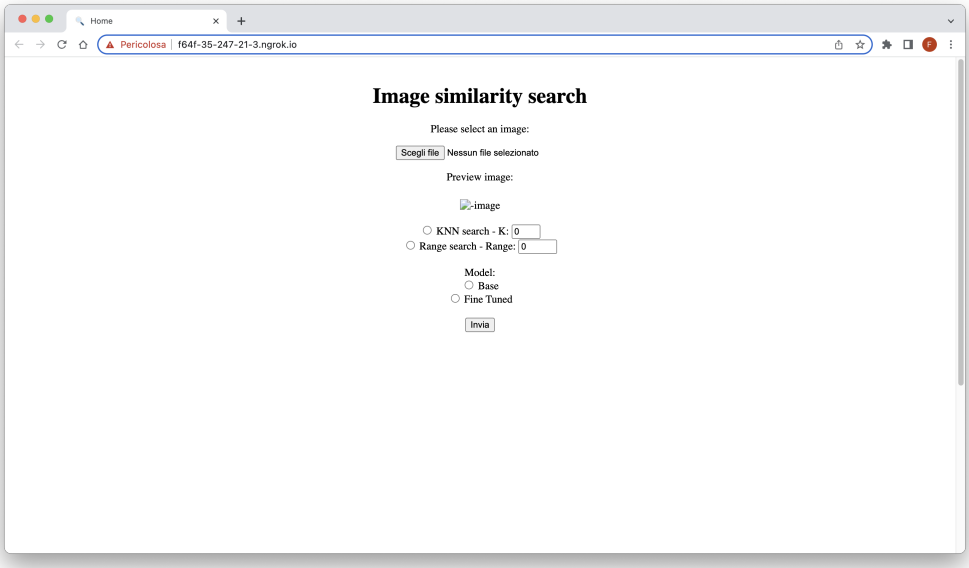
Flask renders the page into the folder "templates", in this folder is stored the index.html page, which is the homepage, and only page of this webapp, since it has been used ajax (asynchronous requests to the REST server) to implement request and response. The static folder contains the CSS file for the index.
The request contain all the field of the form, the image chosen by the user, what kind of search to perform, the opportunity to execute the search with the Base model or with the Fine Tuned model. The image is an input for the chosen model the features are returned, these are used to perform the search through the VPT index opportunely created.
The response is organized as a json file, which contains all the similar images in base64 format, these images are resized and displayed in a specific ¡div¿ of the web page. Since the image similarity search is carried out with the feature vector, the functions that manage the two different search, range

and knn, returns the label and id of the image, field of the point class, that let us to retrieve the actual images.

6.2 User Manual

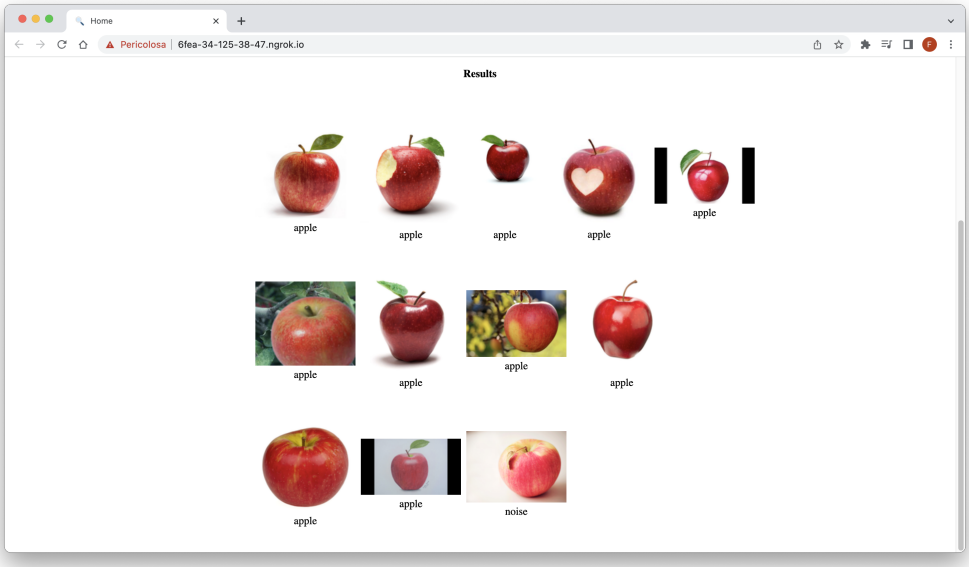
1. Go into "webapp" folder
2. Open RESTserver.ipynb
3. Compile every cell into this notebook
4. Compiled the last cell, select the second link that appears, something like this "http://2710-34-73-234-202.ngrok.io"

Now a security message from the browser could appear, this because with the base version of ngrok is not expected https connection and for this kind of use this is not a problem, so go ahead and the homepage will render.



1. Pressing "scegli file" button to load an image from the personal computer
2. Select the type of search to perform (KNN or Range) and choose the parameter properly
3. Select the model to use to perform the search
4. Press "invia"

This is an example of a search result.



For better readability, under every image is shown the class they belong to. Could appear also "noise" this is the class of the distractor.
The search take a while, because other then search the most similar image, the app retrieve from google drive the right images, this could take some time, so resize them convert them in base64 and so on.