# Programming Task 1

## Task general description

In this task you will

* Implement an object recognition pipe
* Train it on a subset of the Caletch101 dataset
  + Including tuning of hyper parameters
* Test your accuracy on another subset
* Analyze and report your results

We will use publicly available implementations for all of the algorithms, and the main programming effort is composing the code into a functional pipe and experimenting with it to get reasonable results.

The exercise goals:

* Experience building a full pipe including representation.
* Get acquainted with classic representation techniques.

## Data set

We will work with Caltech 101 dataset.

* The data contains 101 classes, with 31-800 images each.
* Data homepage: <http://www.vision.caltech.edu/Image_Datasets/Caltech101/>
* Download the data folder from this link only: https://www.dropbox.com/s/h690aizla6m2857/101\_ObjectCategories.zip?dl=0
* We will train on 25 images per class, test on 25 others. If there are less than 50 images for the class, we will train on half the images (number rounded up if the number of images is odd), and test on the other half. The images should be sorted by file names alphabetical order (ascending), then the first 25 (or half) images should be chosen for training.
* When loading the data:
  + The data is in 101 folders in the top folder 101\_ObjectCategories\, one folder per class
  + To load it write a function which uses the function listdir() (**os package**) on the top folder 101\_ObjectCategories\, then loops through the subfolders and through the images in each subfolder. At the end of the process you should have the images in a single 3D Data array of size (with a side length parameter, and the number of images) and a single Labels vector of size
    - Load images using imread() (**cv2 package**)
    - Get them to gray scale using rgb2gray() (**cv2 package**)
    - Resize them to using imresize() (**cv2 package**) and stack them into the data array
  + **Highly recommended** – to save time in your work, once you load the data, save it as a .pkl file (**pickle package**). This way, instead of loading the images from scratch each time you run (which may take some time), you can load the .pkl file. This is way faster. For more information see <https://docs.python.org/2/library/pickle.html> (this file should not be submitted!)
* Data split (classes should be ordered by the folder names ascending alphabetical order)
  + Debug and tune your pipe on classes 21-30 (fold 1).

\*Please remember that when we tune model hyper parameters, we split our training set (which in this case contains 25 or half the images from each set) into 2 sets - training set and validation set (or k sets if we use cross-validation). See “task1 programming guidelines” presentation. Don’t use the test set of classes 21-30 for hyper parameter validation!

When the final configuration is debugged and stable, run the algorithm with the best hyper parameter configuration found, on classes 41-50 (fold 2). The test results on these classes (fold 2) should be reported in the report you submit.\* Don’t forget that python indices start from 0. This means that class number 21 is actually the class of index 20 in the folder names list. Make sure to use the correct classes. For your own validation: class number 21 should be cougar\_body, and class number 41 should be flamingo\_head.

## Optional pipes to implement

You may choose to implement one of the following pipes:

* SVM + HOG pipe
* SVM + KMeans + SIFT pipe
* NBNN pipe (using SIFTs and a KDTree for nearest neighbor computation)

In pipes using **SVM**, you should use a linear kernel as the default. However, you are required to try at least one more kernel (polynomial, RBF) and report the best results.

If you are not implementing NBNN, you will have to use **SVM** for multi-class classification. You are required to implement the SVM by the "one vs. all" approach – if M is the number of classes, then M binary classifiers are constructed. More details on that further down this document.

In pipes using **SIFT**, you're required to use dense SIFT – run SIFT on a dense grid of locations in the image at a fixed scale and orientation. In order to run dense SIFT, you'll need to pre-define the keypoints to cover the entire grid of the image. An example to how this can be done is in the Moodle under the name "dense\_sift\_example".

## Required packages:

**General packages:** os, numpy (version 1.18.1), pickle, matplotlib, cv2 (version 3.4.2.16)

**HOG algorithm:** skimage ([link](http://scikit-image.org/docs/dev/api/skimage.feature.html)) (name for installation using pip: scikit-image. version: 0.16.2)

**SVM algorithm:** sklearn ([link](https://scikit-learn.org/stable/modules/svm.html)) (version 0.22.1).

**SIFT algorithm:** cv2 ([link](https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_sift_intro/py_sift_intro.html) to sift description). For dense sift, see dense\_sift\_example.py. In order to run dense SIFT, make sure you install version 3.4.2.16 of cv2 and its expansion (it won’t work in newer versions!). To do so, run the following commands in the terminal:

pip install opencv-python==3.4.2.16

pip install opencv-contrib-python==3.4.2.16

**Kmeans algorithm:** sklearn ([link](https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html))

**KDTree algorithm** – SciPy ([link](https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.spatial.KDTree.html)) (version: 1.4.1). KDtree is a data structure used to store a set of D-dimensional points and efficiently find the nearest neighbors of a new query point.

## Important details for algorithms we use

**HOG algorithm**:

Important parameters - spatial cell size, number of orientation bins.

**SVM algorithm:**

Important parameters -C the SVM tradeoff parameter, when using an RBF kernel, the polynomial degree when using a polynomial kernel.

In order to implement "one vs. all" approach with a linear case of SVM, use "LinearSVC" function and set the variable multi\_class=”ovr”. For other cases, there is no "one vs. all" implementation by sklearn, and you'll need to implement it. Here are guidelines and **suggested** code structure for implementation:

* Write a function m\_classes\_SVM\_train() which loops through the M classes and trains M binary classifiers in one-versus-all method
  + That is: for classifier , the examples of class get the label +1, and the examples of the rest of the classes get the label .
  + The function returns an m\_classes\_SVM structure which contains the M SVM models. (structure can be a list of models, for example. Implement as you wish)
* Write the m\_classes\_SVM\_predict() function which accepts a set of N examples to test and an m\_classes\_SVM structure with the M models, returned by m\_classes\_SVM\_train(). The function predicts using the following procedure:
  + Apply the M SVMs to the data and put the predictions in an class score matrix.
  + Compute the predicted class (an vector) by taking the argmax over the class score matrix columns (the highest score in the row determines the winning class)
  + Return the predicted class and the class score matrix

**SIFT algorithm**:

We extract SIFTs around every point in a dense 2D grid of points over the image. The grid stride\step (i.e. the distance between points) is likely to be an important parameter. Also, at each point SIFTS may be extracted at multiple scales. I suggest using the scales used at <http://vision.cse.psu.edu/seminars/talks/2009/random_tff/bosch07a.pdf>, section 5, ’Appearance’. This is a good starting point.

**Kmeans algorithm**:

When training the dictionary, it is enough to extract some SIFTS from some images (that is: you do not have to use all the SIFTS from all the images). For example, using a subset of 100 SIFTS from 1000 images (100,000 SIFTs overall) should be enough for good clustering). The most important parameter is K (*num\_clusters)*. Several hundred is a good area to look in.

## Submission details

You will submit the code (single .py file) and a short report of the results (single .doc/.pdf file).

**Code**

* The code is required to run directly, without modifications, on my machine.
* This machine will have Python 3.6 run with Pycharm IDE including the all the required packages.
* **The only required modification in the code will be the path to the dataset folder. This path should appear at the beginning of the code in a variable named – “data\_path”.**
* The code will be generic and will be able to run on any subset of 10 classes (from the 101 classes) defined by a single variable called **class\_indices** which will be initiated at the beginning of the main file (in one of the first 5 lines). class\_indices will be a list of 10 containing the indices of classes on which the experiment will run. For example:

class\_indices = [ 0, 6, 10, 60, 65, 67, 81, 83, 86, 90]

**Important**: The code you submit should set class\_indices to "fold 2" classes (41-50), i.e. class\_indices = [40, 41, 42, 43, 44, 45, 46, 47, 48, 49]. However, I will test your code on an arbitrary set of 10 classes by changing the class\_indices variable. You should verify that your code is able to run on such an arbitrary set (with indices between 0 and 100).

The code should use fixed hyper parameters (the best configuration of hyper-parameters you found on "fold 1", i.e. classes 21-30). If you are using SVM, the code should use the kernel you found to be best. The code you submit should include the code you used for parameter tuning (so I’ll be able to review it), but not run it. You can leave it commented out. Write “### Tuning” at its beginning so I’ll be able to find it easily.

* The code should print to the python console as output in a clear and readable format: the names of the classes tested, the number of images used for training and testing from each class, the final test error result and the () confusion matrix. It should also produce the error images (see below).
* The submitted code should run in a reasonable time (not more than a few minutes).

**Report**

The report should include

* Hyper parameter tuning (on fold1 classes): The pipe includes hyper parameters (like – the image size, – the number of codewords for Kmeans, – the SVM tradeoff parameter, etc..). Such parameters should be tuned to get the best accuracy. I list below the most important hyper parameters for each module, but you may consider tuning other parameters.
  + For at least two parameters, systematic tuning should be done, in which:
    - The training set (25 or half the images of each class) is split to two subsets, termed 'training' and 'validation'
    - Several values of the parameter are tested, by training on the training set and estimating the error on the validation set.
    - The parameter value giving the lowest validation error is chosen.

\*This procedure is can be replaced by k-fold cross validation if you want to try it.

* + The report should include an explanation of the validation procedure you performed (how the training set was split into ‘training’ and ‘validation’ sets, or cross validation subsets, and how many images in each) the tuning procedure (which hyper parameters were tuned, at what order), which SVM kernels you tried (if you used an SVM pipe).
  + The report should include graphs showing the validation error as a function of hyper parameter value for at least 2 hyper parameters that were systematically tuned. The chosen value should be stated.
    - Each graph should include a coherent and informative caption and axis titles.
    - For each tuned hyper-parameter, a short explanation of the parameter should be included.
* Test results:
  + The error rate obtained over fold2 test set.
  + A confusion matrix (you can use sklearn.metrices.confusion\_matrix(…)), and description of the most interesting observations you can extract from this matrix in at most 10 sentences. Some questions of interest: is the error focused more on certain classes/class pairs while other classes are perfectly predicted? If yes, which classes/class pairs are the problematic? Can you conjecture why (maybe with the help of the error visualization images)? Can you suggest what can be done further in order to reduce the error?
* Error visualization: For each class, show images of the two largest errors on images of the class (i.e. images from the class which were miss-classified). The error images should be displayed only if they exist (i.e. if there are at least two errors from the class). If there is only one error from the class –show it, and if there are none – just state that there were no errors for this class. By *largest error* I mean the images which got the lowest margin, following this definition:
  + Class\_score(: For SVM-based system Class\_score( is defined as the SVM score of the classifier. For NBNN, Class\_score( is the image-to-class-i similarity.
  + The margin for an example of class *i* is . This is the difference between the score of the correct class score and the maximal score of incorrect classes. Larger values indicate higher confidence. A value below 0 is an error.

\*use matplotlib/seaborn package for plotting

\*Don’t use cv2.imshow() or cv2.destroyAllWindows() in the submitted code. Please use plt.show() and plt.imshow() for showing the images. For example:

image = cv2.imread(...)  
plt.imshow(image)  
plt.show()