Hanoi University of Science and Technology
School of Information and Communication of Technology

CAT AND DOG CLASSIFICATION

IT3190E

Nguyen The An	20210006
Do Dinh Kien	20214906
Dau Van Can	20214879

Hanoi, 2023

Contents

1 Abstract

The classification of images is a fundamental task in computer vision, and distinguishing between cats and dogs has long been a popular challenge in this field. This abstract summarizes the key components and approaches used in the classification of cat and dog images.
First, the image classification process typically begins with data preprocessing, including resizing, normalization, and augmentation techniques. These steps aim to enhance the model's ability to extract relevant features and reduce the impact of variations in image size and lighting conditions.
Next, feature extraction plays a crucial role in cat and dog classification. Traditional methods involved hand-crafted features, such as texture descriptors and shape-based representations that in this project, we use feature descriptors: Histogram of Gradient (HOG).
After data processing, we feed the dataset into the models (K-NN, SVM, Random Forest, Gradient Boosting, K-means, CNN): for training on the training set and prediction on the test set then hyperparameter tuning to find optimal parameters.
To evaluate the performance of cat and dog classification models, various metrics are used, including accuracy, precision, recall, confusion matrix and F1 score. Cross-validation techniques are often employed to ensure robustness of the models and to mitigate overfitting.

2 Exploratory data analysis

The dataset we used in this project can be found here. It contains 25000 images of dogs and cats.
Our data set is equally divided into 2 categories: dog and cat. In this figure, we encoded dog with 0 and cat with 1.

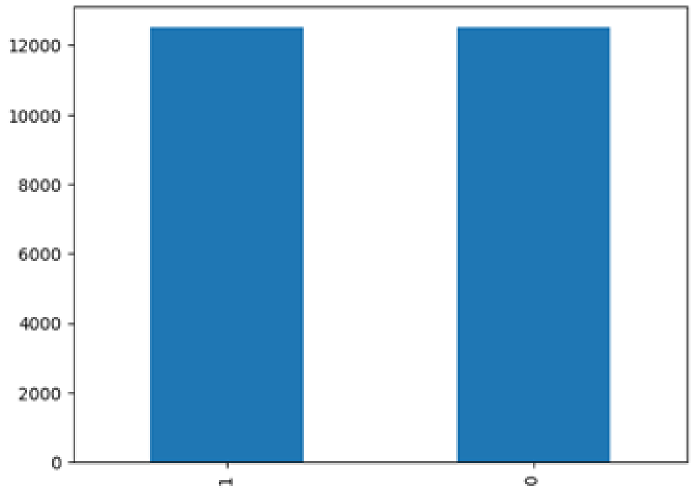


Figure 1. Image Distribution

One more thing to notice about images is their resolution. As you can see in the figure below, our data is varied in resolution.

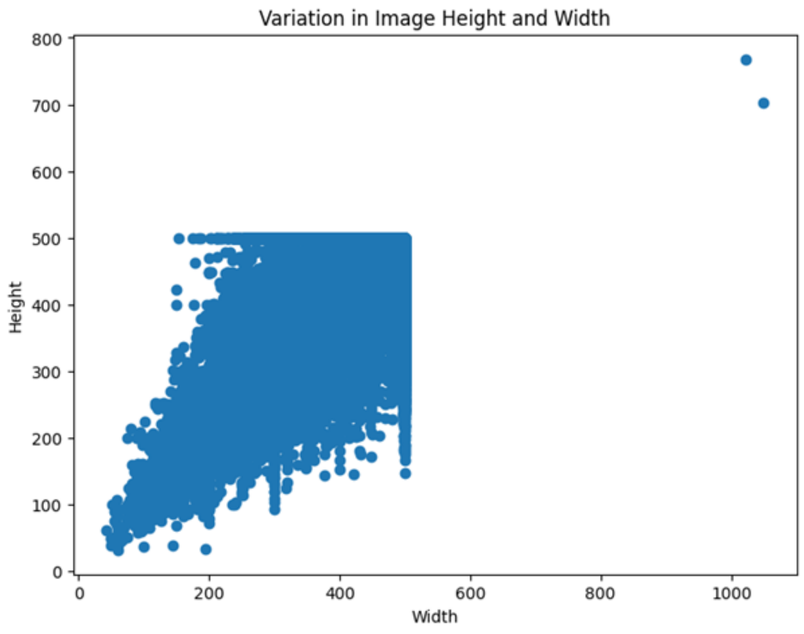


Figure 2. Variation in Image Resolution

After running some codes, we obtain the smallest resolution is 50x38 and the largest resolution is 1023x768. That is a lot of difference in size and shape. So, we need to resize them into one resolution for later processing. There are even some noises in our data. For example, in a cat image there is a dog in it or in another image there is a person holding a dog. All these noises can affect our model's result.

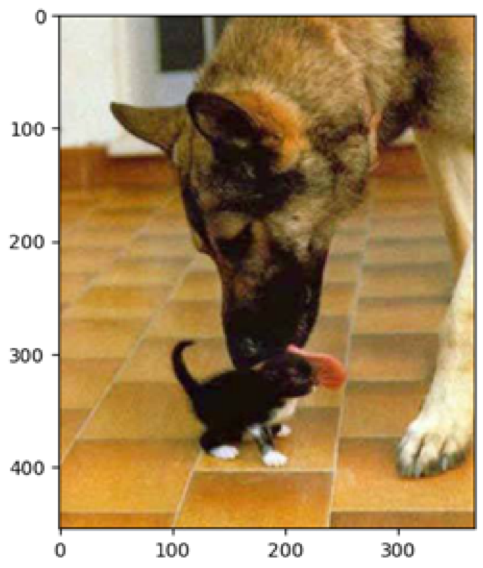


Figure 3. Image cat.724.jpg: a dog in a cat picture

3 Preprocessing

3.1 Image preprocessing

As we said before, the dataset used with images of different sizes is not uniform, so we need to resize the image to a certain size for later processing. Specifically, we resize the image to 150*150.

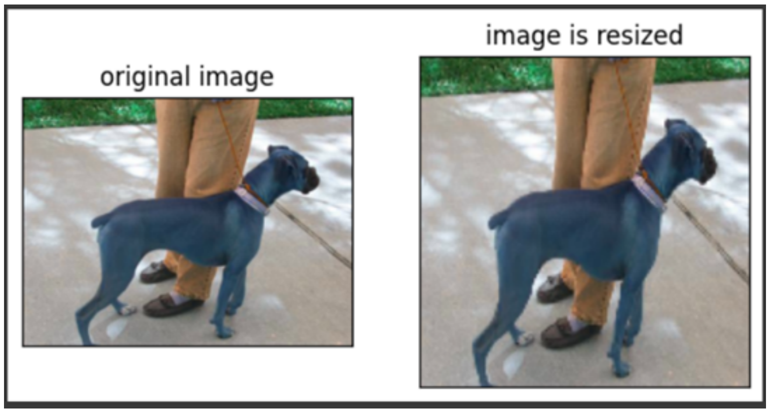
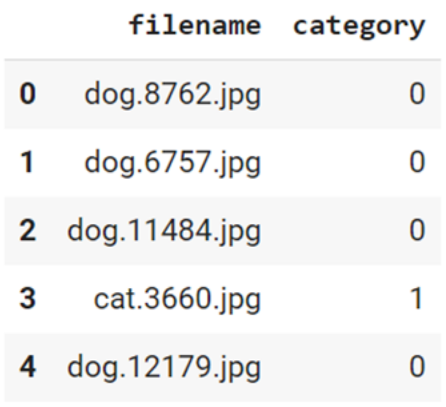


Figure 4. Original image vs. resized image

The resized image will be represented as a tensor 3D of size 150 * 150 * 3 with 3 channels Red, Green, Blue. Then, we astype tensor 3D image representation in format value 'float32'.

3.2 Label encoding

Before putting the data into the machine learning model, we need to label encode 2 classes dog and cat. Here dog will be encoded to 0 and cat will be encoded to 1.



	filename	category
0	dog.8762.jpg	0
1	dog.6757.jpg	0
2	dog.11484.jpg	0
3	cat.3660.jpg	1
4	dog.12179.jpg	0

Figure 5. Class encoded

3.3 Feature extraction (HOG)

To put data into training in machine learning, we need to turn each image into a feature vector. Initially we planned to turn each pixel into a feature, but this approach has some problems that make the problem difficult: become costly in terms of computation and time as well as reduce the performance of the model. So, we decided to use the Histogram of Oriented Gradient algorithm to extract features in each image.
If we sign I is the original image matrix and Gx, Gy are two image matrices where each point on it is a derivative of the axis x and axis y. We can compute the kernel as follows:

- Horizontal derivative:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * I$$

- Vertical derivative:

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * I$$

The * symbol is similar to the convolution between the left filter and the right input image. The gradient magnitude and gradient direction can be generated from two derivatives Gx and Gy according to the formula below:

- Gradient magnitude:

$$G = \sqrt{G_x^2 + G_y^2}$$

- Gradient direction:

$$\theta = arctan(\frac{G_y}{G_x})$$

We find that the characteristics of each image are represented through two parameters that are the degree of change in color intensity (gradient magnitude matrix) and the direction of color intensity change (gradient direction matrix). So we need to create a feature descriptor that transforms the image into a vector that represents both of these information.
To do so, the image is divided into a grid of squares, each of which is 8x8 pixels. So we have a total of 64 pixels corresponding to each cell. On each cell of 64 pixels we will need to calculate 2 parameters that are gradient magnitude (gradient magnitude) and gradient direction. So a total of 8x8x2 = 128 values to be calculated including 64 gradient magnitude values and 64 gradient direction values as shown in the matrix shown below:

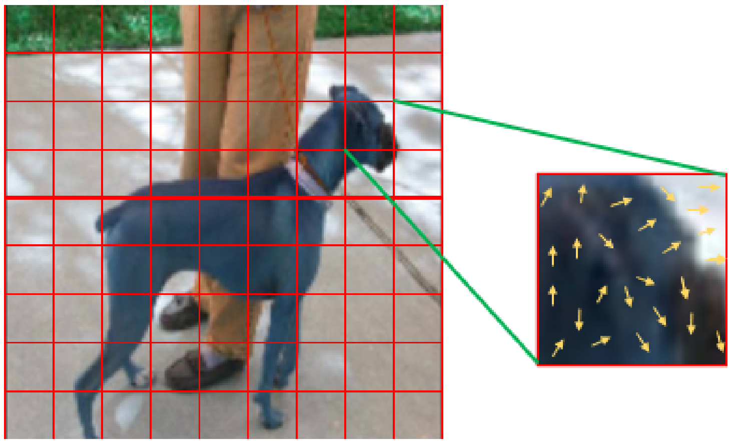
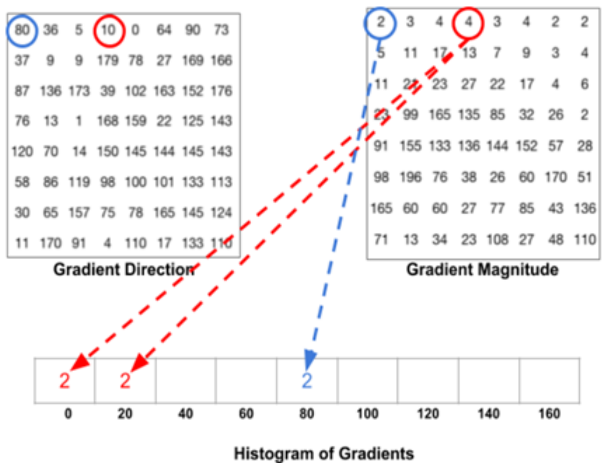


Figure 6. Example of generated histogram vector

The histogram vector will be generated as follows:

- Step 1: Mapping the gradient magnitude to the corresponding bins of the gradient direction. Sort the gradient values in order from smallest to largest and divide them into 9 bins. The magnitude of the gradient direction will be in the range [0, 180] so each bin will have a length of 20 as shown below.
  Each gradient direction will pair with a gradient magnitude at the same coordinate position. When we know which bins the gradient direction belongs to in the bins vector, we will fill in the value of the gradient magnitude in that bin. Can you imagine?
  For example, in the figure below, the box circled in the blue circle corresponds to the gradient direction of 80 and the magnitude of the gradient is 2. Then at the bins vector of HOG, the gradient direction of 80 will fall at the 5th position, so in this cell, we fill in the value 2 corresponding to the gradient magnitude.



Gradient Direction

80	36	5	10	0	64	90	73
37	9	9	179	78	27	169	166
87	136	173	39	102	163	152	176
76	13	1	168	159	22	125	143
120	70	14	150	145	144	145	143
58	86	119	98	100	101	133	113
30	65	157	75	78	165	145	124
11	170	91	4	110	17	133	110

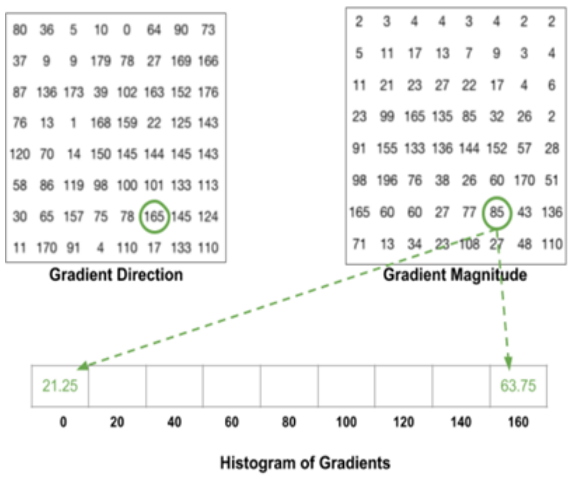
Gradient Magnitude

2	3	4	4	3	4	2	2
5	11	17	13	7	9	3	4
11	21	23	27	22	17	4	6
23	99	165	135	85	32	26	2
91	155	133	136	144	152	57	28
98	196	76	38	26	60	170	51
165	60	60	27	77	85	43	136
71	13	34	23	108	27	48	110

Figure 7. Mapping magnitude of gradients with bins

Ends are values that are divisible by the width of a bin (e.g. 0, 20, 40, etc. are bin ends). In case the magnitude of the gradients does not fall on the endpoints, we will use linear interpolation to divide the magnitude of the gradient into 2 adjacent bins that the gradient value falls into. For example, the gradient value is equal to x paired with gradient magnitude equal to y, x belong to [x0, x1] .i.e. the direction of gradients falls in the middle bin (l-1) and bin l. Then at 2 bins l-1 and l is filled in the intensity value according to the interpolation formula:

- Value at bin l-1:
$$x_(l-1) = \frac{x_1 - x}{x_1 - x_0} * y$$
- Value at bin l:
$$x_(l) = \frac{x - x_0}{x_1 - x_0} * y$$



80	36	5	10	0	64	90	73
37	9	9	179	78	27	169	166
87	136	173	39	102	163	152	176
76	13	1	168	159	22	125	143
120	70	14	150	145	144	145	143
58	86	119	98	100	101	133	113
30	65	157	75	78	165	145	124
11	170	91	4	110	17	133	110

Gradient Direction

2	3	4	4	3	4	2	2
5	11	17	13	7	9	3	4
11	21	23	27	22	17	4	6
23	99	165	135	85	32	26	2
91	155	133	136	144	152	57	28
98	196	76	38	26	60	170	51
165	60	60	27	77	85	43	136
71	13	34	23	108	27	48	110

Gradient Magnitude

0	20	40	60	80	100	120	140	160
21.25								63.75

Histogram of Gradients

Figure 8. Computing example

Calculating the sum of all gradient magnitudes belonging to the same bins of the bins vector, we get the Histogram of Gradients plot as shown below:

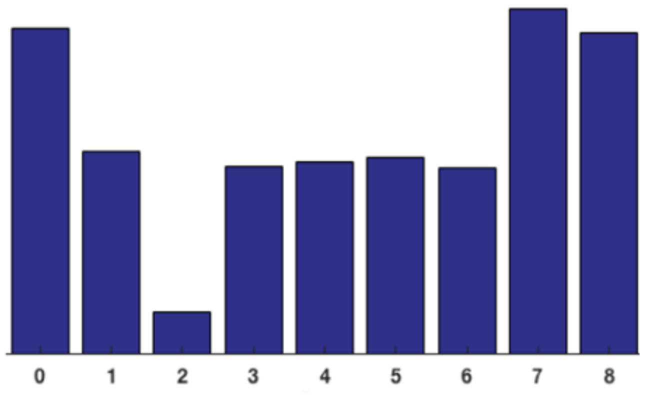


Figure 9. Graph of Histogram of Gradients including 9 bins

- Step 2: Normalize the histogram vector according to the 16x16 block We see that the histogram vector will be dependent on the intensity of the pixels of an image. With 2 photos with the same content but the darker variant image is made up of the original image matrix multiplied by 1/2. Then the value of the histogram vector of the original image will also be twice the histogram vector of the variant image. Therefore, it is necessary to normalize the histogram vector so that both images have the same representation vector.
  Normalize the second order norm:

  In addition, we can also use the normal norm of order 1.
  In there h is the histogram vector of the gradients.

The normalization process will be performed on a 2x2 block on the original grid (8x8 pixels each). Thus, we will have 4 histogram vectors of size 1x9, concatenate the vectors to obtain a synthetic histogram vector of size 1x36 and then normalize according to the 2nd order norm on this vector.

- Step 3: Calculate the HOG feature vector.
  After normalizing the histogram vectors, we will concatenate these 1x36 vectors into one large vector. This is the HOG vector representing the entire image.

That is the detailed step of how to compute HOG of a image. In our project, we will use get hog function from sklearn library.

```python
from skimage.feature import hog
#Feature extraction
def get_hog_features(img, orient=8, pix_per_cell=16, cell_per_block=4,vis=False, feature_vec
    =True):
    if vis == True:
        features, hog_image = hog(img, orientations=orient,
                                  pixels_per_cell=(pix_per_cell, pix_per_cell),
                                  cells_per_block=(cell_per_block, cell_per_block),
                                  transform_sqrt=True,
                                  visualize=vis, feature_vector=feature_vec,channel_axis=-1)
    else: # Otherwise call with one output
        features = hog(img, orientations=orient,
                       pixels_per_cell=(pix_per_cell, pix_per_cell),
                       cells_per_block=(cell_per_block, cell_per_block),
                       transform_sqrt=True, visualize=vis, feature_vector=feature_vec,
    channel_axis=-1)
        return features
```

4 Model Building

4.1 Import libraries

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import cv2
import os
import random
from skimage.feature import hog,SIFT
from sklearn.model_selection import train_test_split, GridSearchCV, cross_val_score
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier as RFC
from sklearn.cluster import KMeans
from catboost import Pool,CatBoostClassifier
from sklearn.preprocessing import LabelEncoder,StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.decomposition import PCA
from sklearn import metrics
from mlxtend.plotting import plot_decision_regions
from keras.preprocessing.image import ImageDataGenerator
from keras.utils import to_categorical

from keras.models import Sequential, load_model, Model
from keras.layers import Conv2D, MaxPooling2D, BatchNormalization, Dropout
from keras.layers import Dense, Flatten
from keras.callbacks import EarlyStopping
from sklearn.metrics import (
    accuracy_score,
    roc_curve,
    recall_score,
    confusion_matrix,
    precision_score,
    f1_score,
    accuracy_score,
    classification_report,
    ConfusionMatrixDisplay)
```

4.2 Models

An indispensable thing in machine learning is hyperparameter tuning to increase the performance and generalization of the model. Our initial dataset has 25000 images (20000 images in the training set and 5000 images in the testing set) with such large data set tuning will take a lot of time and resources. So, we decided to split the dataset with a ratio of 10% to tuning. This can indeed affect the generalizability of the model over our entire original data, but we would expect that the 10% of the dataset taken for some tuning will describe the properties quite well over the entire dataset original data. After tuning the hyperparameter with the divided data set, we select a set of hyperparameters whose model performance is 'best for each' to build the model on the entire original data.

4.2.1 KNN

K-Nearest Neighbors is one of the simplest methods in Machine Learning. It has an intuitive algorithm with only one hyperparameter k. For classification problems, a class label is assigned based on the majority vote of the nearest neighbors.
We use the KNeighborClassifier from scikit-learn for our model. We then train the model using the divided dataset to find the best hyperparameter k for the model.

```python
def train_and_test_knn(k_arr):
  for k in k_arr:
    knn = KNeighborsClassifier(n_neighbors = k)
    knn.fit(X_train_tune, y_train_tune)
    y_pred_tune = knn.predict(X_test_tune)
    report = classification_report(y_test_tune, y_pred_tune, target_names=['dog','cat'],
    digits=3, output_dict=True)
    print('k={}'.format(k))
    print('Accuracy: {}'.format(report['accuracy']))
    print('Precision for cats: {}'.format(report['cat']['precision']))
    print('Precision for dogs: {}'.format(report['dog']['precision']))
    print('Recall for cats: {}'.format(report['cat']['recall']))
    print('Recall for dogs: {}'.format(report['dog']['recall']))
    print('F1 Score for cats: {}'.format(report['cat']['f1-score']))
    print('F1 Score for dogs: {}'.format(report['dog']['f1-score']))
```

We then plotted a graph to compare the performance of the model for different k values.
After tuning, we found out that k=4 is the best value for k.

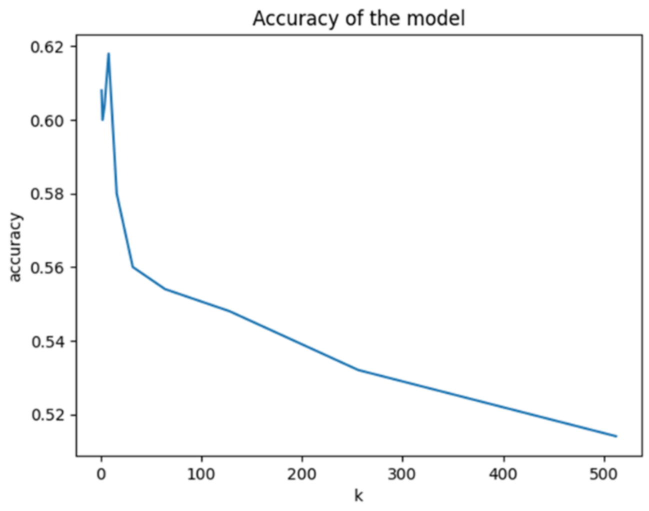


Figure 10. Accuracy of the model with corresponding k

4.2.2 Support Vector Machine (SVM)

SVM (Support Vector Machines) is a machine learning algorithm used for classification and regression tasks. It finds an optimal hyperplane to separate classes by maximizing the margin between them. SVM can handle nonlinear data using kernel functions and has regularization parameters to balance accuracy and margin width. It can be extended to multi-class classification and regression problems. SVM is effective in high-dimensional spaces but sensitive to hyperparameters and computationally expensive. In SVM there are 3 most common hyperparameters that we can tune which are:

- Kernel: Some common kernel functions include linear, radial basis function (RBF), and sigmoid.
- C: The parameter determines the trade-off between achieving a low training error and maximizing the margin violation
- Gamma: The gamma parameter is specific to the RBF kernel. It determines the reach of the individual training examples, influencing the smoothness of the decision boundary.

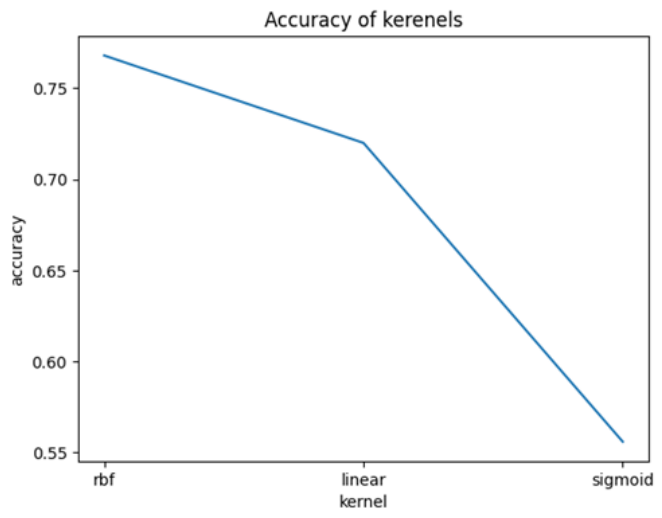


Figure 11. Performance of model with each Kernel

We can see that the rbf kernel gives us the best model performance among the 3 kernels. So we choose this kernel to tune the next hyperparameters (because the kernel is chosen as rbf, so the Gamma will be tuned).

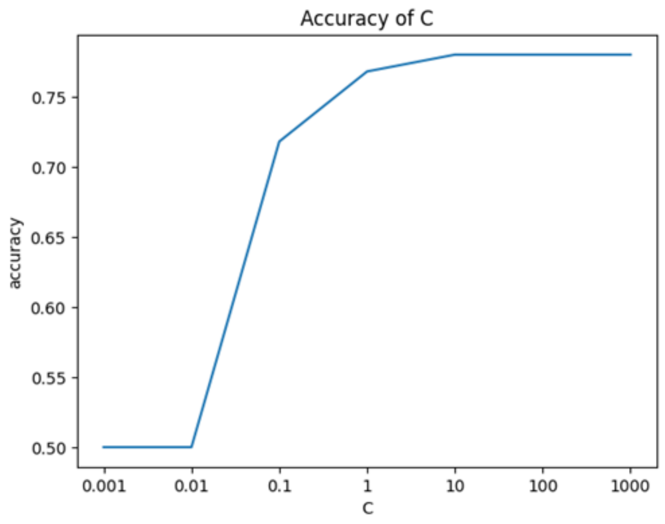


Figure 12. Performance of model with each C parameter when the permanent kernel is rbf

As can be seen, the performance of the model with the values of C:10,100,1000 is the same. Then we will choose C=10 with the desire to get a wider and clearer separation margin.

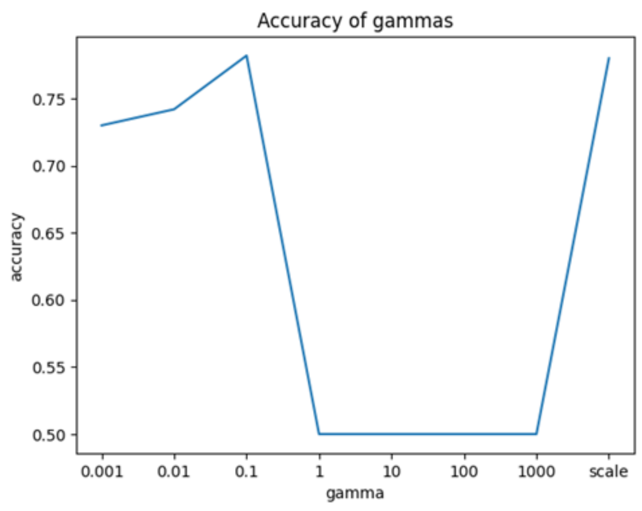


Figure 13. Performance of model with each Gamma parameter when we permanent kernel is rbf and C=10

From the histogram, we decided to choose the gamma value as 0.1 because then the model's performance is the best.
In short, after hyperparameter tuning, we have best parameter set to fit and predict on original data is (Kernel: rbf, C: 10, Gamma: 0.1)

```
svm_model=SVC(kernel='rbf',C=10,gamma=0.1)
svm_model.fit(X_train,y_train)
train_pred_svm=svm_model.predict(X_train)
test_pred_svm=svm_model.predict(X_test)
```

4.2.3 Gradient Boosting (CatBoost)

CatBoost is a powerful gradient boosting algorithm that excels at handling categorical features. It combines accuracy, robustness, and efficiency, making it a valuable tool for various machine learning tasks, including classification and regression. To improve model when use CatBoost, we are tuning 2 hyperparameter which are:

- Iterations: iterations refer to the number of boosting stages or trees that are built during the training process.
- Learning rate: controls the step size during the gradient descent process.

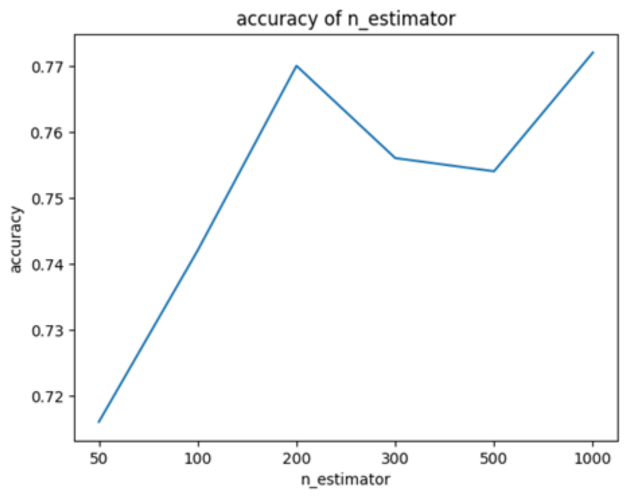


Figure 14. Model's performance with each iterations (n_estimators)

With iterations = 1000, we have best accuracy on test set to tune.

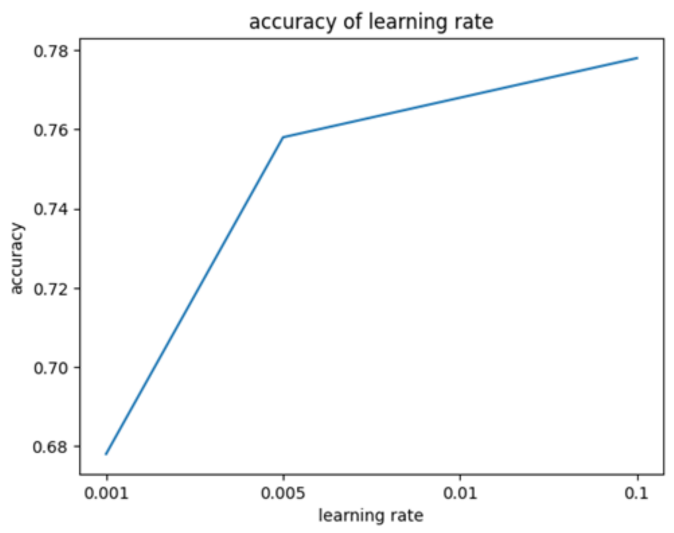


Figure 15. Model's performance with each learning rate when iterations = 1000

When iterations =1000 then learning rate = 0.1 is best learning rate to has best model's performance.
After hyperparameter tuning, we choose iterations=1000 and learning rate=0.1 to fit with original data.

```
cbc=CatBoostClassifier(iterations=1000,learning_rate=0.1)
cbc.fit(Pool(X_train,y_train))
train_pred_cbc=cbc.predict(X_train)
test_pred_cbc=cbc.predict(Pool(X_test))
```

4.2.4 Random Forest

Random Forest is an ensemble learning algorithm that combines multiple decision trees to make predictions. It uses randomness in data sampling and feature selection to create diverse trees, reducing overfitting. The algorithm combines predictions of all the trees through voting. Random Forest provides feature importance rankings and has advantages like high accuracy, resistance to overfitting, and scalability. It is used for classification, regression, and feature selection in various domains such as finance, healthcare, and remote sensing. The model we implemented is as below. Random forest provides some parameters for tuning, in this problem we choose 4 parameters for tuning which are:

- n_estimator: number of decision trees to be used.
- Max_depth: The maximum depth allowed for each decision tree
- Max_feature: Limits a count to select the maximum features in each tree.
- Max_leaf_nodes: Refers to the maximum number of leaf nodes that a tree in the forest can have.

4.2.5 K-means Clustering

K-means is a popular clustering algorithm used in machine learning and data analysis. It is an unsupervised learning algorithm, which means it doesn't require labeled data for training. K-means is used to partition a dataset into K distinct clusters, where K is a predefined number chosen by the user.
The algorithm works as follows:

- Initialization: Randomly select K points from the dataset as initial centroids.
- Assignment: Assign each data point to the nearest centroid based on the Euclidean distance between the point and centroids.
- Update: Recalculate the centroids as the mean of all data points assigned to each centroid.
- Repeat steps 2 and 3 until convergence or a maximum number of iterations is reached.

- Convergence: The algorithm converges when the assignments of data points to centroids no longer change or change very minimally.

The goal of K-means is to minimize the within-cluster sum of squares, also known as inertia. It seeks to find centroids that minimize the distance between data points within each cluster while maximizing the distance between different clusters. The algorithm iteratively adjusts the centroids to achieve this objective.
There is one thing very important in this algorithm that is the number of clusters. We only have 2 classes doesn't mean that our k = 2. In fact, k can have many values because there isn't an only shape for a dog nor a cat. So, we will use the elbow method to select the appropriate k.

```python
# elbow method
# squared distance
sse = []
list_k = [2,16,64,100,256]

for k in list_k:
    #km = KMeans(n_clusters=k)
    km = KMeans(init ='k-means++', n_clusters = k, verbose = 0)
    clusters = km.fit_predict(X_train_tune)
    sse.append(km.inertia_)

    reference_labels = get_reference_dict(clusters,y_train_tune)
    predicted_labels = get_labels(clusters,reference_labels)

    print(f"Accuracy for k = {k}: ",
          accuracy_score(predicted_labels,y_train_tune))

# Plot sse against k
plt.figure(figsize=(6, 6))
plt.plot(list_k, sse, '-o')
plt.xlabel(r'Number of clusters *k*')
plt.ylabel('Sum of squared distance')
```

4.2.6 Convolutional Neural Network (CNN)

A convolutional neural network (CNN) is a type of artificial neural network that is particularly effective in processing and analyzing visual data. It is widely used in various computer vision tasks such as image classification, object detection, and image segmentation.

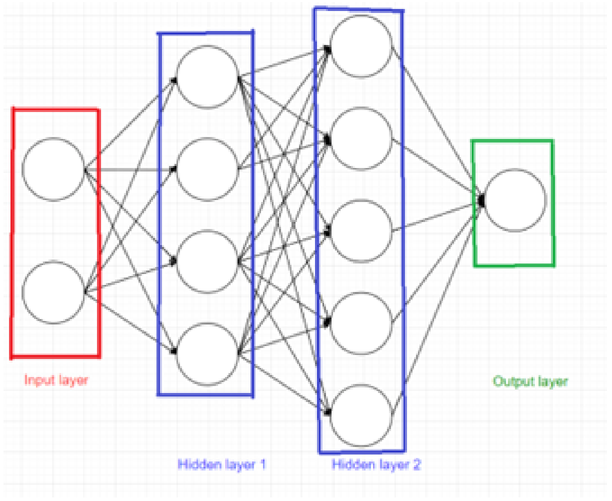


Figure 16. Example of a CNN

The architecture of CNN is designed to mimic the visual cortex of the human brain, which is specialized in processing visual information. The key component of a CNN is the convolutional layer, which applies filters to the input image to extract meaningful features. These filters detect edges, corners, and other visual patterns that are important for understanding the content of an image.
We treat our data a little differently compared to our previous model. Since a CNN model can extract features from an image itself, we don't have to do that anymore. We will replace it with ImageDataGenerator.

The ImageDataGenerator allows you to apply various data augmentation techniques to the images in real-time during training. Data augmentation involves applying random transformations or modifications to the images, resulting in new variations of the original data. This technique helps to increase the diversity of the training set and reduces overfitting. Some common data augmentation techniques include rotation, scaling, shearing, flipping, zooming, and brightness/contrast adjustments.

```python
train_generator = ImageDataGenerator(
    rotation_range=15,
    rescale=1./255,
    shear_range=0.1,
    zoom_range=0.2,
    horizontal_flip=True,
    width_shift_range=0.1,
    height_shift_range=0.1
)
test_generator = ImageDataGenerator(rescale=1./255)
```

Our CNN model:

```python
model = Sequential()

model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(150,150,3)))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Conv2D(128, (3, 3), activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.25))
model.add(Dense(2, activation='softmax')) # 2 because we have cat and dog classes

model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

model.summary()
```

Let's go through each layer in the CNN model and explain its purpose:

1. Conv2D Layer
    - The first layer in the model is a convolutional layer.
    - It has 32 filters with a kernel size of (3, 3), indicating that it will learn 32 different features or patterns from the input.
    - The activation function used is ReLU, which introduces non-linearity and helps the network model complex relationships.
    - The input_shape parameter defines the shape of the input images, which is (150, 150, 3) in this case (150 pixels height, 150 pixels width, and 3 color channels for RGB image
2. BatchNormalization Layer
    - The BatchNormalization layer normalizes the activations of the previous layer, making the network more stable during training.
    - It helps in reducing the internal covariate shift, which is the change in the distribution of layer activations due to the changing input distribution.
3. MaxPooling2D Layer

- The MaxPooling2D layer performs down-sampling by taking the maximum value within a window or pool size.
- It reduces the spatial dimensions (height and width) of the feature maps, which helps in reducing computational complexity and makes the network more robust to variations in translation and scale.

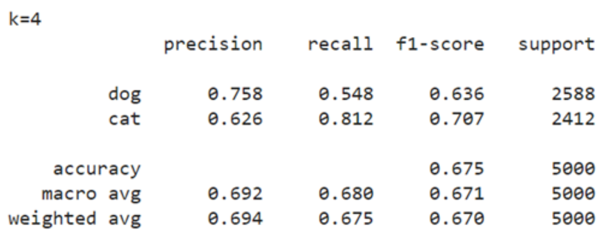
4. Dropout Layer
   - The Dropout layer randomly sets a fraction of the input units to zero during training.
   - It helps in regularizing the network by preventing overfitting and reducing the interdependencies between neurons.
   - In this model, a dropout rate of 0.25 is applied after each pooling layer. The above sequence of layers (Conv2D, BatchNormalization, MaxPooling2D, Dropout) is repeated three times, gradually increasing the number of filters from 32 to 64 and then to 128. This allows the network to learn more complex and abstract features as it goes deeper.
5. Flatten Layer
   - The Flatten layer is used to convert the multi-dimensional output from the previous convolutional layers into a one-dimensional vector.
   - It "flattens" the feature maps into a single long vector, preparing the data for input to the dense layers.
6. Dense Layer
   - The Dense layer is a fully connected layer that receives the flattened input from the previous layer.
   - It consists of 512 neurons and uses the ReLU activation function.
   - This layer learns high-level representations and complex relationships between features.
7. Another BatchNormalization and Dropout Layer
   - Following the dense layer, another BatchNormalization layer and Dropout layer are added.
   - These layers help regularize the dense layers and prevent overfitting.
8. Output Layer
   - The final Dense layer has 2 neurons, representing the two classes: cat and dog.
   - The activation function used is softmax, which produces probabilities for each class.

The model is then compiled with a categorical cross-entropy loss function, Adam optimizer, and accuracy metric. Overall, this CNN model consists of multiple convolutional layers for feature extraction, pooling layers for down-sampling, dropout layers for regularization, and dense layers for classification. The combination of these layers enables the model to learn and extract meaningful features from images and make predictions on cat and dog classes.

5 Results

5.1 KNN

After tuning, we get the best hyperparameter k = 4. We then fit and predict on the original data with the k value we just found and print the evaluation results:



```
k=4
              precision    recall  f1-score   support

         dog      0.758     0.548     0.636      2588
         cat      0.626     0.812     0.707      2412

    accuracy                          0.675      5000
   macro avg      0.692     0.680     0.671      5000
weighted avg      0.694     0.675     0.670      5000
```

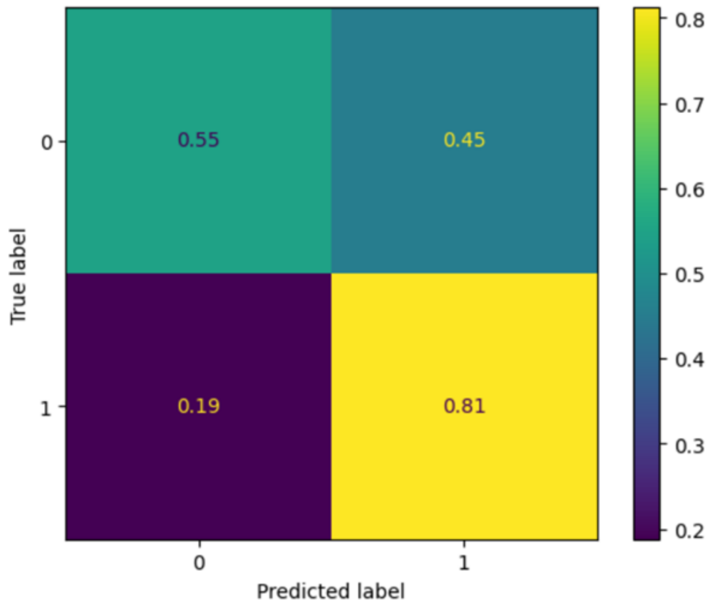


Figure 17. Result of KNN with k=4

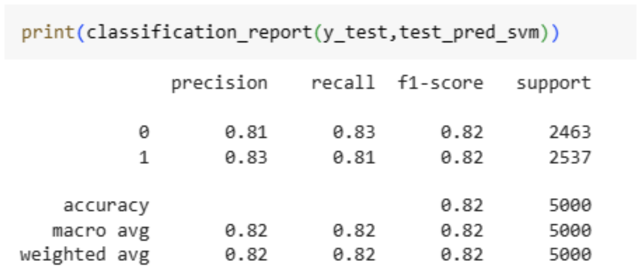
The model has an accuracy of 67.5%, which is not bad, but still poor for a cat/dog classifier. It also seems biased towards classifying images as dogs, with an 81% recall for dogs and only 55% for cats, and the precision for cats (76%) is significantly higher than that of dogs (63%).
We also use a 5-fold cross validation to see the consistency of the model.

```
#KNN + K-FOLD
pred_kfold = cross_val_score(knn, X_train, y_train, cv=5)
print("Accuracy with K-NN and K-FOLD CROSS VALIDATION: %0.2f (+/- %0.2f)" % (pred_kfold.mean
    (), pred_kfold.std() * 2))
```

The result shows that the accuracy is roughly the same as without k-fold, with the standard deviation of only 1%.

5.2 SVM

This is the result of the metrics used to evaluate the model after fitting and predicting.

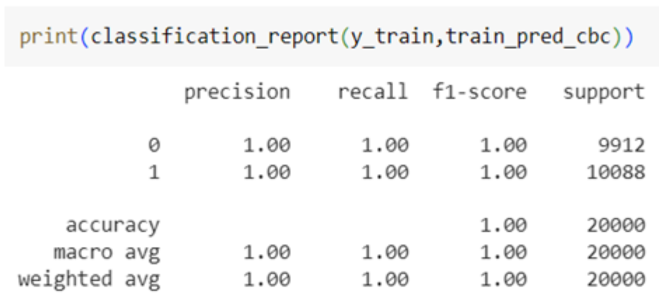


```
print(classification_report(y_test,test_pred_svm))

              precision    recall  f1-score   support

           0       0.81      0.83      0.82      2463
           1       0.83      0.81      0.82      2537

    accuracy                           0.82      5000
   macro avg       0.82      0.82      0.82      5000
weighted avg       0.82      0.82      0.82      5000
```

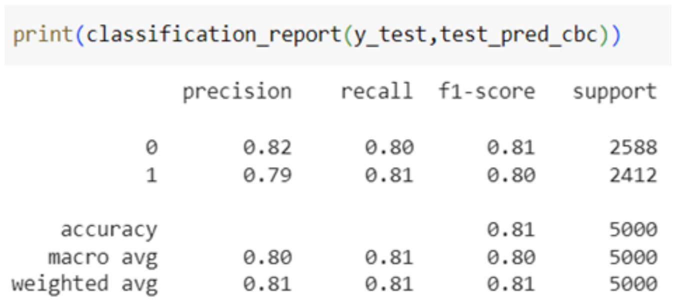
The accuracy of the model is 82%, pretty good for the two-class image classification problem.

5.3 CatBoost

This is the result on the original data:



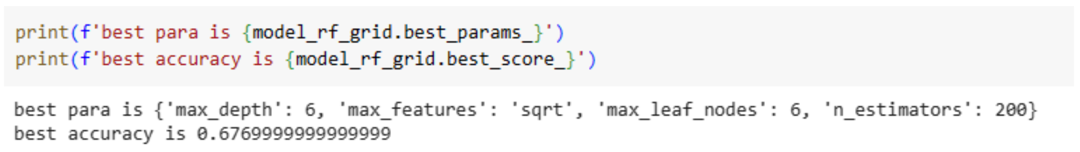
```
print(classification_report(y_train,train_pred_cbc))

              precision    recall  f1-score   support

           0       1.00      1.00      1.00      9912
           1       1.00      1.00      1.00     10088

    accuracy                           1.00     20000
   macro avg       1.00      1.00      1.00     20000
weighted avg       1.00      1.00      1.00     20000
```



```
print(classification_report(y_test,test_pred_cbc))

              precision    recall  f1-score   support

           0       0.82      0.80      0.81      2588
           1       0.79      0.81      0.80      2412

    accuracy                           0.81      5000
   macro avg       0.80      0.81      0.80      5000
weighted avg       0.81      0.81      0.81      5000
```

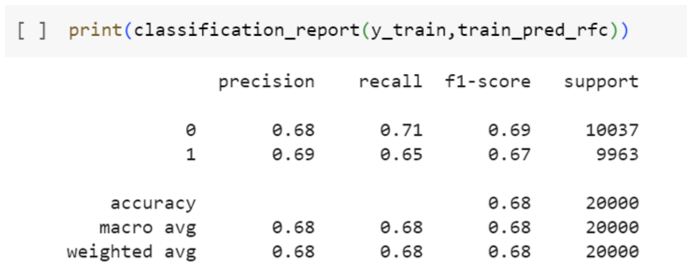
5.4 Random Forest

After tuning by Grid Search, we print out best hyperparameter set and best score(accuracy):

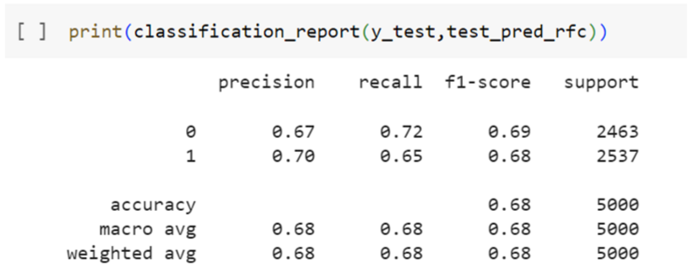


```
print(f'best para is {model_rf_grid.best_params_}')
print(f'best accuracy is {model_rf_grid.best_score_}')

best para is {'max_depth': 6, 'max_features': 'sqrt', 'max_leaf_nodes': 6, 'n_estimators': 200}
best accuracy is 0.6769999999999999
```

Finally, we fit and predict on original data with best hyperparameter that has just been found and print out result of evaluation functions:



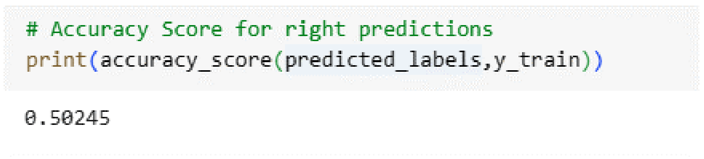
```
[ ]  print(classification_report(y_train,train_pred_rfc))

              precision    recall  f1-score   support

           0       0.68      0.71      0.69     10037
           1       0.69      0.65      0.67      9963

    accuracy                           0.68     20000
   macro avg       0.68      0.68      0.68     20000
weighted avg       0.68      0.68      0.68     20000
```



```
[ ]  print(classification_report(y_test,test_pred_rfc))

              precision    recall  f1-score   support

           0       0.67      0.72      0.69      2463
           1       0.70      0.65      0.68      2537

    accuracy                           0.68      5000
   macro avg       0.68      0.68      0.68      5000
weighted avg       0.68      0.68      0.68      5000
```

5.5 K-means Clustering

Here is our first try with the full dataset and k = 2:



```
# Accuracy Score for right predictions
print(accuracy_score(predicted_labels,y_train))
```

```
0.50245
```

That's very low regards that we only have 2 classes. Now we will use our small dataset for elbow method to choose the appropriate k.

```
Accuracy for k = 2:  0.5225
Accuracy for k = 16:  0.626
Accuracy for k = 64:  0.6725
Accuracy for k = 100:  0.6555
Accuracy for k = 256:  0.6855
```

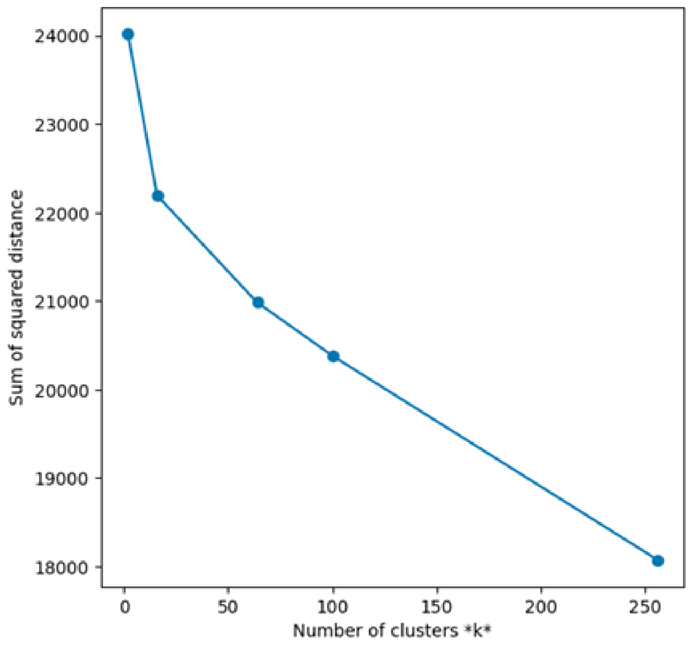
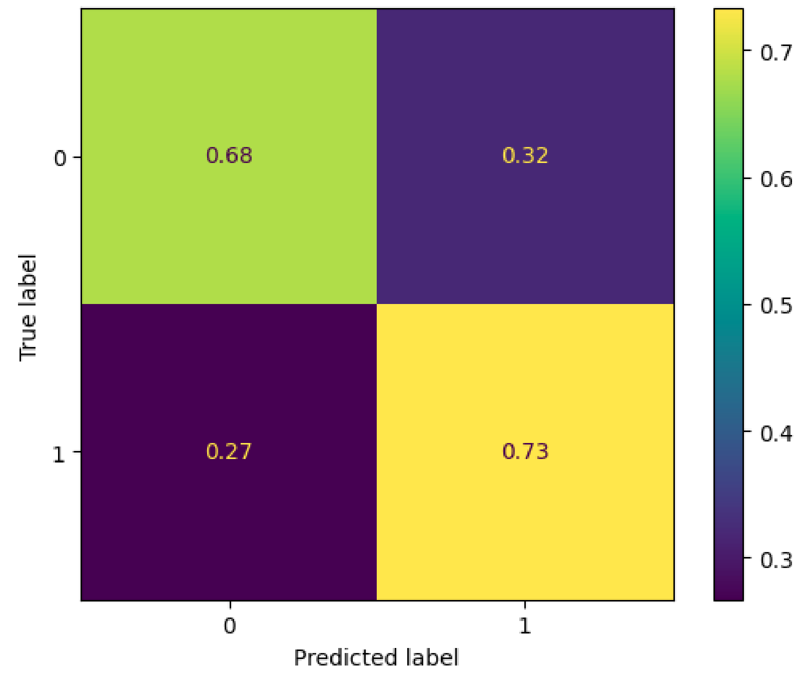


Figure 18. Elbow method

As you can see, the sum of squared distance still drops drastically even if k = 256. So we choose k = 256. And then after training with k = 256 and the full dataset, we got the result:

```
              precision    recall  f1-score   support

         cat      0.719     0.679     0.699     10037
         dog      0.694     0.733     0.713      9963

    accuracy                          0.706     20000
   macro avg      0.707     0.706     0.706     20000
weighted avg      0.707     0.706     0.706     20000
```



5.6 CNN

We also use a smaller dataset for tuning parameters.
We have 2 models: the first is model0. It is the same as our previous model, but the BatchNormalization layers and the Dropout layers are removed. And the second is our model as I mentioned.
Both models are trained with the small dataset and we obtain a surprised result. The model0 performs better than our old model.

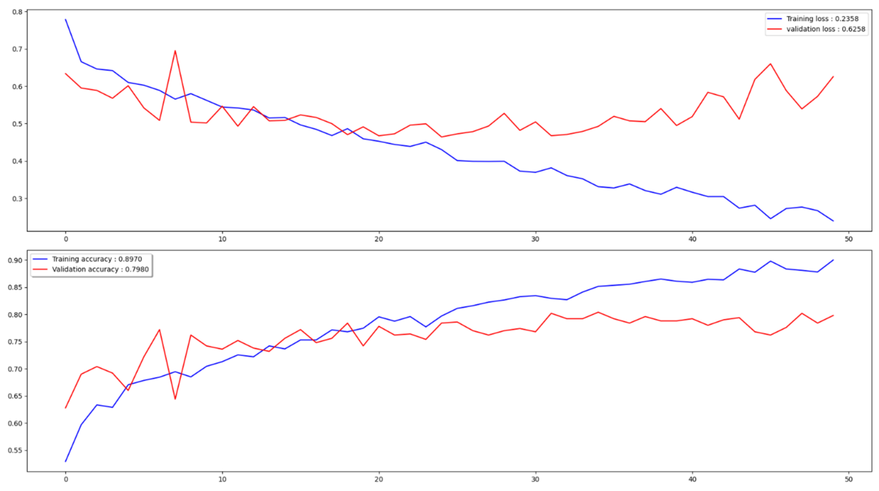


Figure 19. Model0 (without BatchNormalization layer and Dropout layer)

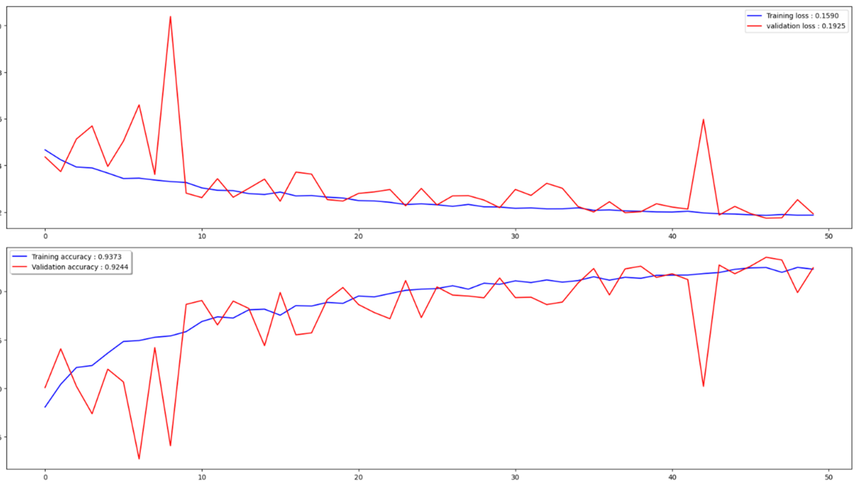


Figure 20. Our old model

But as theory, the BatchNormalization layer and Dropout layer are supposed to make our model better. Explanation for this can be:

- Insufficient data: Deep learning models, including CNNs, typically require a large amount of data to generalize well. With a small dataset, the model may not have enough examples to learn meaningful patterns and relationships. In this case, the additional regularization provided by BatchNormalization and Dropout layers can potentially hinder the learning process further by reducing the effective sample size.
- Over-regularization: BatchNormalization and Dropout layers are regularization techniques that help prevent overfitting. However, if the dataset is already small, applying too much regularization can lead to underfitting. The model may become overly constrained, resulting in poor performance and limited ability to capture the complexity of the data.

So even if it performs worse on our small dataset, we still decided to use the old model because in theory, with a larger dataset, these 2 layers can have many good impacts on our model. Here's our result:

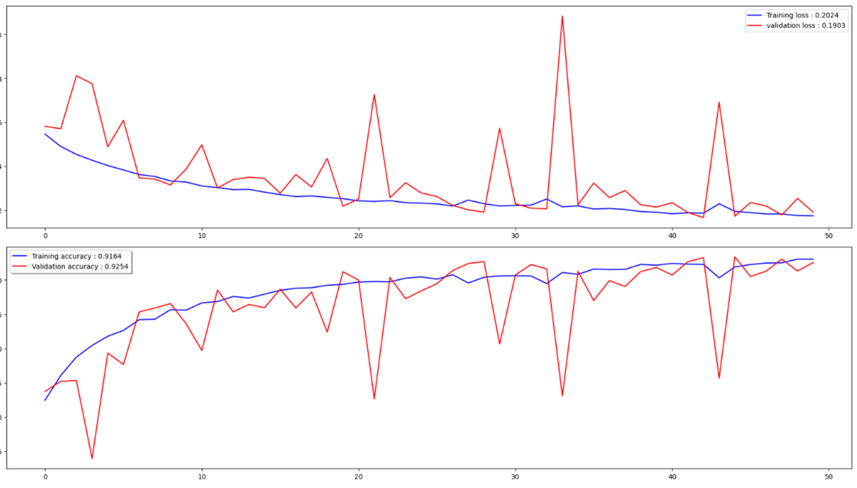
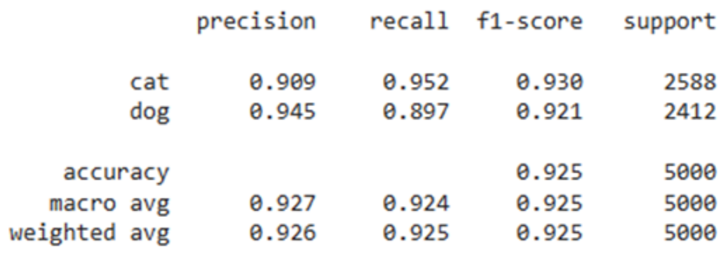
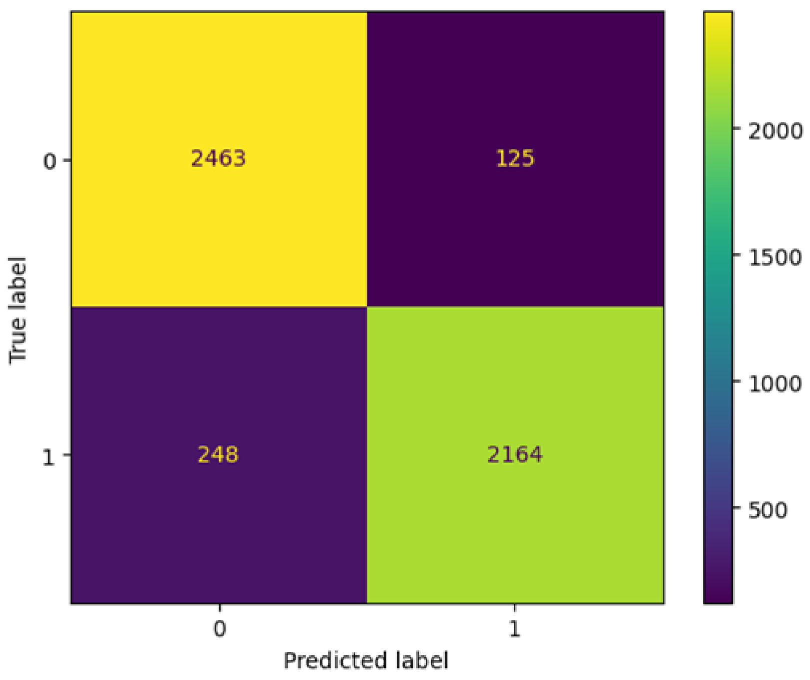


Figure 21. Old model with full data



```
              precision    recall  f1-score   support

         cat      0.909     0.952     0.930      2588
         dog      0.945     0.897     0.921      2412

    accuracy                          0.925      5000
   macro avg      0.927     0.924     0.925      5000
weighted avg      0.926     0.925     0.925      5000
```



5.7 Total Comparison

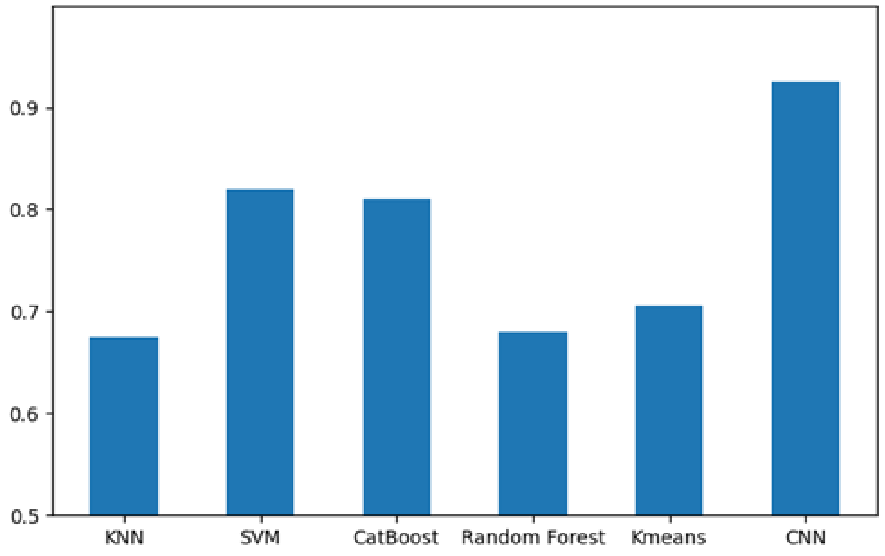


Figure 18. Comparison between models

6 Conclusion and improvements

As you can see in the total comparison, our models, except for CNN, have a result of 70% - 80% accuracy. In our opinion this is an acceptable result because images classification is a very difficult task for these kinds of models. We get a few bad results at first when we haven't tried parameter tuning yet, but after doing that we get much better results. The CNN model executes well with 93% accuracy.
What we have done here is not completed yet, there is still a lot of room for improvement. We will try some other way to make our model better like further tuning or transfer learning.

7 Reference

- Dataset: Dogs vs. cats
- Libraries: scikit-learn