ML-475 Final Project: Face Masking Recognition

A monitoring strategy based on machine learning algorithm in the COVID-19 era

Instructed by: Aggelos K Katsaggelos

Member： Jiaqi Guo: JGR9647;   Manzhu Wang: MWU2863;   Xinyi Su: XSY0714;   Chenxi Liu: CLS9757

YouTube Access: https://youtu.be/r6gWZx_7GY4

GitHub Access: https://github.com/GuoJiaqi-1020/EE-475-ML-Final-Project

Project

December 6, 2021

1 EE475 Group Project : Machine Learning Based Face Mask Recognition

JiaqiGuo: JGR9647; ManzhuWang: MWU2863; XinyiSu: XSY0714; ChenxiLiu: CLS9757

As the COVID-19 has brought great disaster to human beings, personal protection has become particularly important. For the purpose of controlling the spread of the epidemic, every one of us has the obligation and responsibility to wear masks. The objective of our project is to proposed a system that can monitor people's mask wearing status (Correct, Incorrect and No mask). In our project, we reduce the dimension of input data space by using pre-processing methods: convert into gray image, Histogram of Oriented Gradients algorithm and Canny edge detector algorithm. We firstly implement linear model from scratch, then implement SVM, Decision Tree and Random Forest using scikit learn library.

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import sklearn.svm as svm
import sklearn.tree as tree
import sklearn.ensemble as ensemble
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn import metrics
from lib_fun import *
from skimage.feature import hog
```

[ ]:
```
# Load the original face mask data with pixel of 20*20, 50*50 and 100*100
# We visulaize these image of Correct, Incorrect and No mask with each␣
↪resolution.
data20, labels20 = load('../Data/Pixel20/')
data50, labels50 = load('../Data/Pixel50/')
data100, labels100 = load('../Data/Pixel100/')
VisualizeRGB(data20, data50, data100)
```

```
X shape: (4559, 20, 20, 3), Y shape: (4559,)
X shape: (4559, 50, 50, 3), Y shape: (4559,)
X shape: (4559, 100, 100, 3), Y shape: (4559,)
```

[ ]:
```
# In order to reduce the input dimension and keep critical information, we transfer the RGB image to gray image
data20_gray = RGBtoGray(data20)
data50_gray = RGBtoGray(data50)
data100_gray = RGBtoGray(data100)
VisualizeGray(data20_gray, data50_gray, data100_gray)
```

[ ]:
```
# Using Histogram of Oriented Gradients to extract features from RGB image
# Compute 8 direction in each 2*2 pixel
data20_hog = RGBtoHOG(data20)
data50_hog = RGBtoHOG(data50)
data100_hog = RGBtoHOG(data100)
VisualizeGray(data20_hog, data50_hog, data100_hog)
```

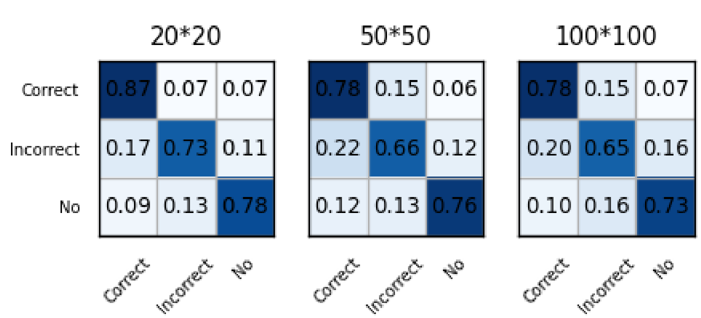
[ ]:
```
# Using Canny edge detector to extract features from gray image
data20_edge = GRAYtoEDGE(data20_gray, sigma=1)
data50_edge = GRAYtoEDGE(data50_gray, sigma=3)
data100_edge = GRAYtoEDGE(data100_gray, sigma=5)
VisualizeGray(data20_edge, data50_edge, data100_edge)
```

[ ]:
```
# Training process using different model. Split the data into training : test = 4 : 1.
# Flatten the image as the input data of model
def train_model(model, data, labels):
    x = Flatten(data)
    x_train, x_test , y_train, y_test = train_test_split(x, labels, test_size = 0.2, random_state=1)
    clf = make_pipeline( StandardScaler(), model)
    clf.fit(x_train, y_train)
    pred = clf.predict(x_test)
    print('accuracy: ', metrics.accuracy_score(y_test, pred))
    confusion = metrics.confusion_matrix(y_test, pred)
    return confusion
```

Firstly We implement one vs rest SVM model. SVM maps training examples to points in space so as to maximise the width of the gap between the two categories. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall. As we are using a soft margin SVM, we set the penalty term C as 1. A low C makes the decision surface smooth, while a high C aims at classifying all training examples correctly. And we are using a kernel function to map the input features to a higher dimension space. In this experiment, we compare the linear, polynomial and rbf kernel. We choose gray and hog data as out training data in this experiment.

[ ]:
```
# SVM with linear kernel, using gray images as data.
confusion20 = train_model(svm.SVC(C=1, kernel='linear', cache_size=4000), data20_gray, labels20)
confusion50 = train_model(svm.SVC(C=1, kernel='linear', cache_size=4000), data50_gray, labels50)
confusion100 = train_model(svm.SVC(C=1, kernel='linear', cache_size=4000), data100_gray, labels100)
plot_confusion_matrix([confusion20, confusion50, confusion100])
```

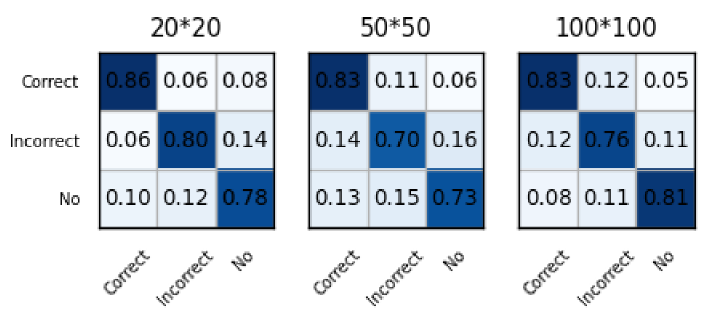
```
accuracy:  0.7905701754385965
accuracy:  0.7324561403508771
accuracy:  0.7214912280701754
```



It is not difficult to find that with the increase of image size, the accuracy of our model even decreases, which is very counterintuitive.

```
[ ]: # SVM with linear kernel, using hog images as data.
confusion20 = train_model(svm.SVC(C=1, kernel='linear', cache_size=4000),␣
 ↪data20_hog, labels20)
confusion50 = train_model(svm.SVC(C=1, kernel='linear', cache_size=4000),␣
 ↪data50_hog, labels50)
confusion100 = train_model(svm.SVC(C=1, kernel='linear', cache_size=4000),␣
 ↪data100_hog, labels100)
plot_confusion_matrix([confusion20, confusion50, confusion100])
```

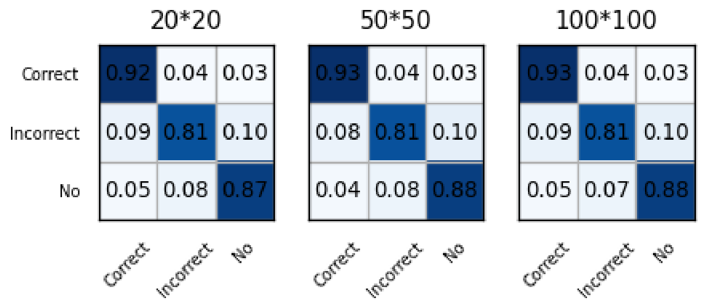
```
accuracy:  0.8125
accuracy:  0.7521929824561403
accuracy:  0.8004385964912281
```



Although the classification accuracy was slightly improved after the HoG feature was used, the overall performance of the model was still unsatisfactory, which indicates that Linear kernel has reached its performance limit, and we must find an alternative methodology to replace it.

```
[ ]: # SVM with 3 degree polynomial kernel, using gray images as data.
confusion20 = train_model(svm.SVC(C=1, kernel='poly', degree=3,␣
 ↪cache_size=4000), data20_gray, labels20)
confusion50 = train_model(svm.SVC(C=1, kernel='poly', degree=3,␣
 ↪cache_size=4000), data50_gray, labels50)
confusion100 = train_model(svm.SVC(C=1, kernel='poly', degree=3,␣
 ↪cache_size=4000), data100_gray, labels100)
plot_confusion_matrix([confusion20, confusion50, confusion100])
```

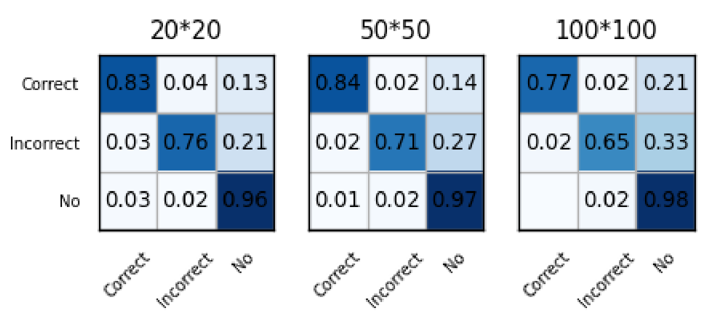
```
accuracy:  0.868421052631579
accuracy:  0.875
accuracy:  0.8739035087719298
```



Here we introduced the 3 degree class polynomial kernel. As can be seen from the figure below, the classification accuracy of all categories has been significantly improved.

```
[ ]: # SVM with 3 degree polynomial kernel, using hog images as data.
confusion20 = train_model(svm.SVC(C=1, kernel='poly', degree=3,␣
 ↪cache_size=4000), data20_hog, labels20)
confusion50 = train_model(svm.SVC(C=1, kernel='poly', degree=3,␣
 ↪cache_size=4000), data50_hog, labels50)
confusion100 = train_model(svm.SVC(C=1, kernel='poly', degree=3,␣
 ↪cache_size=4000), data100_hog, labels100)
plot_confusion_matrix([confusion20, confusion50, confusion100])
```

```
accuracy:  0.8530701754385965
accuracy:  0.8464912280701754
accuracy:  0.8037280701754386
```

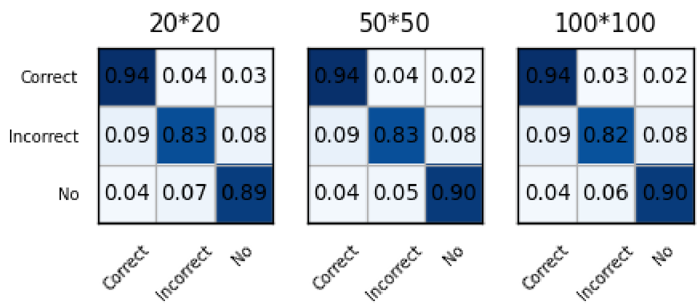


Different from the previous control groups, when we replaced gray graphics with Hog features, the classification accuracy did not significantly improve

```
[ ]: # SVM with rbf kernel, using gray images as data. Beta is set to 1 / n_features.
confusion20 = train_model(svm.SVC(C=1, kernel='rbf', gamma='auto',␣
 ↪cache_size=4000), data20_gray, labels20)
```

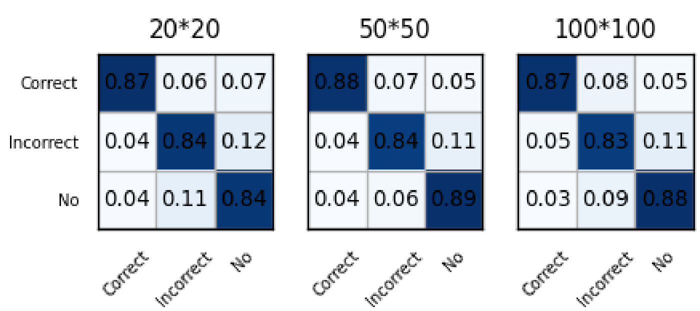
```
confusion50 = train_model(svm.SVC(C=1, kernel='rbf', gamma='auto',␣
↪cache_size=4000), data50_gray, labels50)
confusion100 = train_model(svm.SVC(C=1, kernel='rbf', gamma='auto',␣
↪cache_size=4000), data100_gray, labels100)
plot_confusion_matrix([confusion20, confusion50, confusion100])
```

```
accuracy:  0.8859649122807017
accuracy:  0.8914473684210527
accuracy:  0.8903508771929824
```



[ ]:
```
# SVM with rbf kernel, using hog images as data. Beta is set to 1 / n_features.
confusion20 = train_model(svm.SVC(C=1, kernel='rbf', gamma='auto',␣
↪cache_size=4000), data20_hog, labels20)
confusion50 = train_model(svm.SVC(C=1, kernel='rbf', gamma='auto',␣
↪cache_size=4000), data50_hog, labels50)
confusion100 = train_model(svm.SVC(C=1, kernel='rbf', gamma='auto',␣
↪cache_size=4000), data100_hog, labels100)
plot_confusion_matrix([confusion20, confusion50, confusion100])
```

```
accuracy:  0.8530701754385965
accuracy:  0.8728070175438597
accuracy:  0.8607456140350878
```



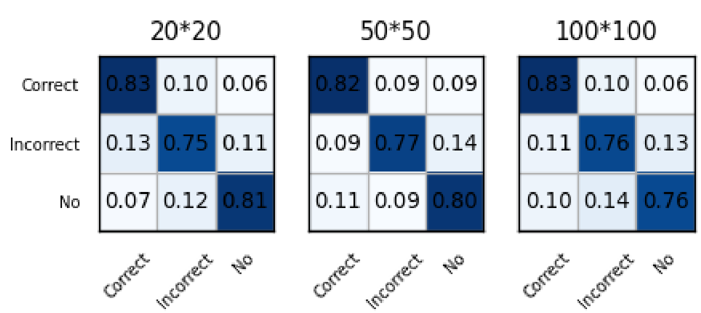
Finally, we test the effect of RBF kernel. Compared with linear kernel, both RBF kernel and POLY kernel can significantly improve the effect of the model. Meanwhile, when gray Scale image(50×50) was used as the input, we obtained the highest accuracy of the SVM model of 89.1%

In this experiment, we can see that the result of rbf kernel is better than polynomial, which is better than linear kernel. The hog method lead to worse results using each kernel function. The reason is that models need more information other than edge information to make right decision.The most interesting thing we can see is that, the highest resolution of image doesn't produce the best results. The reason may be that it is harder for SVM to split high dimension data space.

The next model we choose is decision tree. Decision Trees (DTs) are a non-parametric supervised learning method used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. A tree can be seen as a piecewise constant approximation. In our setting, the maximum depth of the tree is nor specified, nodes are expanded until all leaves are pure or until all leaves contain less than 2 samples. And we compare the results of using gray, hog and canny dataset.

[ ]:
```
# Decision Tree on gray image.
confusion20 = train_model(tree.DecisionTreeClassifier(), data20_gray, labels20)
confusion50 = train_model(tree.DecisionTreeClassifier(), data50_gray, labels50)
confusion100 = train_model(tree.DecisionTreeClassifier(), data100_gray,␣
↪labels100)
plot_confusion_matrix([confusion20, confusion50, confusion100])
```

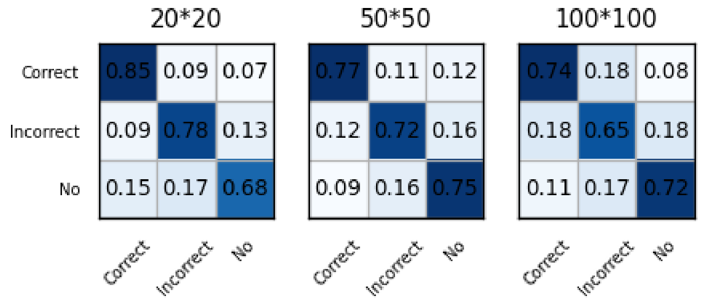
```
accuracy:  0.7993421052631579
accuracy:  0.793859649122807
accuracy:  0.7828947368421053
```



[ ]:
```
# Decision Tree on hog image.
confusion20 = train_model(tree.DecisionTreeClassifier(), data20_hog, labels20)
confusion50 = train_model(tree.DecisionTreeClassifier(), data50_hog, labels50)
confusion100 = train_model(tree.DecisionTreeClassifier(), data100_hog,␣
↪labels100)
plot_confusion_matrix([confusion20, confusion50, confusion100])
```

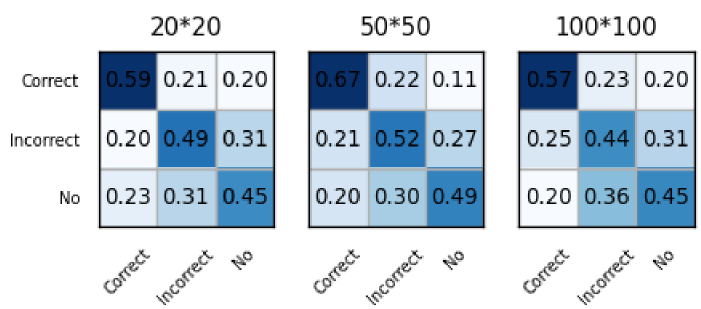
```
accuracy:  0.7675438596491229
accuracy:  0.7467105263157895
```

```
accuracy:  0.7039473684210527
```




```
[ ]: # Decision Tree on canny image.
confusion20 = train_model(tree.DecisionTreeClassifier(), data20_edge, labels20)
confusion50 = train_model(tree.DecisionTreeClassifier(), data50_edge, labels50)
confusion100 = train_model(tree.DecisionTreeClassifier(), data100_edge,
 ↪labels100)
plot_confusion_matrix([confusion20, confusion50, confusion100])
```

```
accuracy:  0.5098684210526315
accuracy:  0.5603070175438597
accuracy:  0.4857456140350877
```

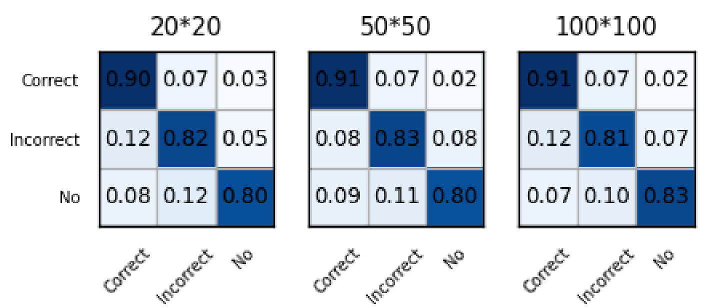


We can see that hog extractor has bad influence on the results, while canny extractor has very bad influence on the results. And the accuarcy of a single tree is worse than SVM with rbf kernel.

Then, we try the random forest algorithm. The Random forest is a special estimator that fits a certain number of single trees on various sub-samples of the dataset and uses averaging to improve the classification accuracy and control the situation of over-fitting. In our setting, The maximum depth of the tree is nor specified, nodes are expanded until all leaves are pure or until all leaves contain less than 2 samples. And we compare the results of 5, 10 and 50 subtrees on only gray dataset.

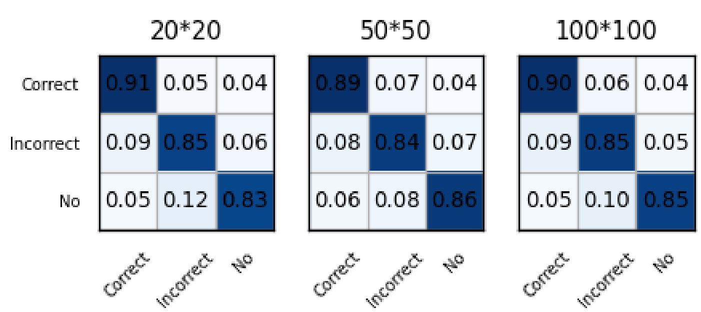
[ ]:
```
# Random Forest using 5 subtrees on gray image.
confusion20 = train_model(ensemble.RandomForestClassifier(n_estimators= 5),␣
↪data20_gray, labels20)
confusion50 = train_model(ensemble.RandomForestClassifier(n_estimators= 5),␣
↪data50_gray, labels50)
confusion100 = train_model(ensemble.RandomForestClassifier(n_estimators= 5),␣
↪data100_gray, labels100)
plot_confusion_matrix([confusion20, confusion50, confusion100])
```

```
accuracy:  0.8410087719298246
accuracy:  0.8475877192982456
accuracy:  0.8464912280701754
```



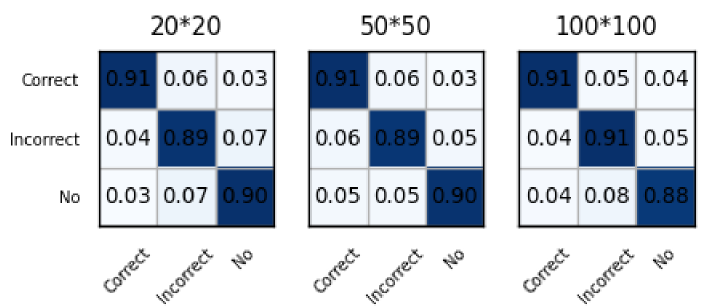
[ ]:
```
# Random Forest using 10 subtrees on gray image.
confusion20 = train_model(ensemble.RandomForestClassifier(n_estimators= 10),␣
↪data20_gray, labels20)
confusion50 = train_model(ensemble.RandomForestClassifier(n_estimators= 10),␣
↪data50_gray, labels50)
confusion100 = train_model(ensemble.RandomForestClassifier(n_estimators= 10),␣
↪data100_gray, labels100)
plot_confusion_matrix([confusion20, confusion50, confusion100])
```

```
accuracy:  0.8640350877192983
accuracy:  0.8662280701754386
accuracy:  0.868421052631579
```



```
[ ]: # Random Forest using 50 subtrees on gray image.
confusion20 = train_model(ensemble.RandomForestClassifier(n_estimators= 50),␣
 ↪data20_gray, labels20)
confusion50 = train_model(ensemble.RandomForestClassifier(n_estimators= 50),␣
 ↪data50_gray, labels50)
confusion100 = train_model(ensemble.RandomForestClassifier(n_estimators= 50),␣
 ↪data100_gray, labels100)
plot_confusion_matrix([confusion20, confusion50, confusion100])
```

```
accuracy:  0.9002192982456141
accuracy:  0.8991228070175439
accuracy:  0.9013157894736842
```



Since the growth of each tree in the random forest has grow to the maximum extent and has a certain degree of randomness, the performance and robustness of the overall model are largely improved when we integrate them together. We can see that random forest has higher accuarcy than a single tree, and the more subtress we use, the higher accuarcy we can gain.

Finally, instead of using scikit learn package, we implement a linear model from scratch. Linear classifier is a very common and efficient classification algorithm in machine learning. In this project we will compare its results as a baseline with the other three classification algorithms. Its result can be illustrated as below:

● Linear Classification Result Summation（Baseline）

➢ Parameter setting:

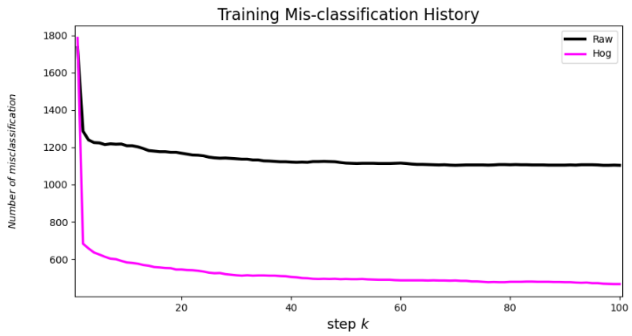
<u>Study Rate</u>: diminishing α, decay from 0.02 (α = 0.02/iteration)
<u>Optimization Method</u>: gradient decent
<u>Cost Function</u>: Multiclass SoftMax

➢ Problems & Solutions

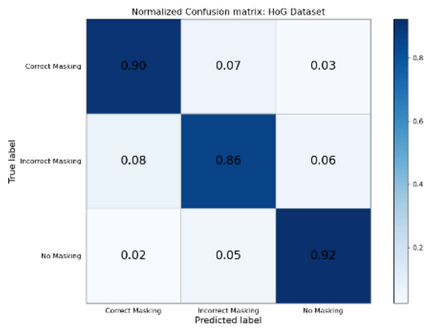
1. When we choose a fixed learning rate, the cost of training will show significant fluctuations, indicating that our choice of learning rate is too large. However, when we choose a smaller learning rate, the model convergence speed will slow down, that will result in a low training efficiency.
   Solution： Using diminishing study rate
2. The appearance and shape of a face cannot be well represented by using the original image data, so the classification accuracy of the model trained by using the original image is low.
   Solution： Using HoG algorithm to extract the image feature

➢ Result

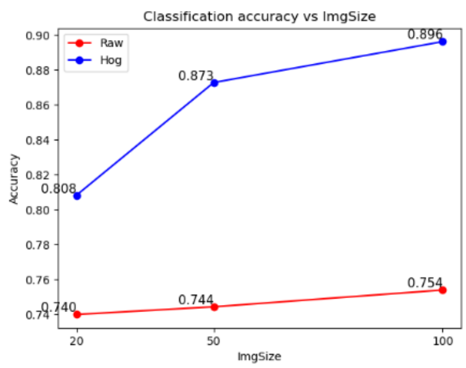
1. ***Significant Improvement in Accuracy:*** By using the image features extracted from the Hog algorithm as the model input, we improved the accuracy by at least 15 (about 800 less misclass samples) percentage points, which is quite remarkable .
   <u>*(*Input size 4500×100×100×3)*</u>



2. ***Classification Accuracy of Different Categories:*** Among the three categories, Incorrect masking has the lowest identification accuracy of only 86%, while No masking has the highest accuracy of 92%. As the data of the three types are almost equal in this project, we do not need to calculate balanced accuracy.



3. ***Sensitive to Change in Image Sharpness:*** As the input image size(sharpness) increases, the classification accuracy of the model increases. As can be seen from the following figure, compared with the Raw dataset, the Hog dataset is more sensitive to the improvement of image sharpness and its accuracy rate increases by nearly 10 percentage points, while the Raw control group has almost no change.



● Appendix

➢ Random Forest.py

```python
import sys
from matplotlib import pyplot as plt
from lib.nonlinear_superlearn_library.recursive_tree_lib.ClassificationTree import ClassificationStump
from lib import edge_extract
import autograd.numpy as np
import copy

sys.path.append('..')



class Tree:
    def __init__(self):
        self.split = None
        self.node = None
        self.left = None
        self.right = None
        self.left_leaf = None
        self.right_leaf = None

        self.number_mis_class_left = 0
        self.number_mis_class_right = 0
        self.all_miss = 0



class Random_Forest_Algorithm:
    def __init__(self, data_name, depth, train_portion, img_size):
        self.gray = []
        self.red = []
        self.blue = []
        self.green = []
        file_path = "../Data/Pixel" + str(img_size[0]) + "/"
        x, y = self.fetchData(file_path, data_name, img_size)
        self.feature_extraction(x, img_size)
        self.x = self.hog_extractor(np.array(self.gray).T)
        print("feature extraction finished")

        self.y = y
        self.depth = depth
        self.colors = ['salmon', 'cornflowerblue', 'lime', 'bisque', 'mediumaquamarine', 'b', 'm', 'g']
        self.plot_colors = ['lime', 'violet', 'orange', 'lightcoral', 'chartreuse', 'aqua', 'deeppink']
```

```python
        self.make_train_val_split(train_portion)

        # build root regression stump
        self.tree = Tree()
        stump = ClassificationStump.Stump(self.x_train, self.y_train)

        # build remainder of tree
        self.build_tree(stump, self.tree, depth)

        # compute train / valid errors
        self.compute_train_val_accuracies()
        self.best_depth = np.argmax(self.valid_accuracies)

    @staticmethod
    def fetchData(file_path, data_name, img_size):
        y = []
        x = np.empty(shape=(0, img_size[0], img_size[1], 3))
        tag = 0
        number = 0
        for name in data_name:
            data_subset = np.load(file_path + name)
            x = np.append(x, data_subset, axis=0)
            y.extend(np.shape(data_subset)[0] * [tag])
            number += np.shape(data_subset)[0]
            tag += 1
        return x, np.reshape(np.array(y), (1, number))

    def feature_extraction(self, data, img_size):
        for i in range(np.shape(data)[0]):
            self.gray_image(data[i, :], img_size)

    def gray_image(self, img, img_size):
        blue = []
        green = []
        red = []
        for i in range(img_size[0]):
            for j in range(img_size[1]):
                red.append(img[i][j][0])
                green.append(img[i][j][1])
                blue.append(img[i][j][2])
        gray = list(0.07 * np.array(blue) + 0.72 * np.array(green) + 0.21 * np.array(red))
        self.gray.append(gray)
        self.red.append(red)
```

```
        self.green.append(green)
        self.blue.append(blue)

    @staticmethod
    def hog_extractor(x):
        kernels = np.array([
            [[-1, -1, -1],
             [0, 0, 0],
             [1, 1, 1]],
            [[-1, -1, 0],
             [-1, 0, 1],
             [0, 1, 1]],
            [[-1, 0, 1],
             [-1, 0, 1],
             [-1, 0, 1]],
            [[0, 1, 1],
             [-1, 0, 1],
             [-1, -1, 0]],
            [[1, 0, -1],
             [1, 0, -1],
             [1, 0, -1]],
            [[0, -1, -1],
             [1, 0, -1],
             [1, 1, 0]],
            [[1, 1, 1],
             [0, 0, 0],
             [-1, -1, -1]],
            [[1, 1, 0],
             [1, 0, -1],
             [0, -1, -1]]])
        extractor = edge_extract.tensor_conv_layer()
        x_transformed = extractor.conv_layer(x.T, kernels).T
        return x_transformed

    def make_train_val_split(self, train_portion):
        self.train_portion = train_portion
        r = np.random.permutation(self.x.shape[1])
        train_num = int(np.round(train_portion * len(r)))
        self.train_inds = r[:train_num]
        self.valid_inds = r[train_num:]
        self.x_train = self.x[:, self.train_inds]
        self.x_valid = self.x[:, self.valid_inds]
        self.y_train = self.y[:, self.train_inds]
        self.y_valid = self.y[:, self.valid_inds]
```

```
    def build_subtree(self, stump):
        # get params from input stump
        best_split = stump.split
        best_dim = stump.dim
        left_x = stump.left_x
        right_x = stump.right_x
        left_y = stump.left_y
        right_y = stump.right_y

        left_stump = stump
        right_stump = stump

        if np.size(np.unique(left_y)) > 1:
            left_stump = ClassificationStump.Stump(left_x, left_y)
        if np.size(np.unique(right_y)) > 1:
            right_stump = ClassificationStump.Stump(right_x, right_y)
        return left_stump, right_stump

    def build_tree(self, stump, node, depth):
        if depth > 1:
            node.split = stump.split
            node.dim = stump.dim
            node.left_leaf = stump.left_leaf
            node.right_leaf = stump.right_leaf
            node.step = stump.step
            left_stump, right_stump = self.build_subtree(stump)

            node.left = Tree()
            node.right = Tree()
            depth -= 1
            return self.build_tree(left_stump, node.left, depth), self.build_tree(right_stump,
node.right, depth)
        else:
            node.split = stump.split
            node.dim = stump.dim
            node.left_leaf = stump.left_leaf
            node.right_leaf = stump.right_leaf
            node.step = stump.step

    def compute_train_val_accuracies(self):
        self.train_accuracies = []
        self.valid_accuracies = []
        for j in range(self.depth):
```

```python
            # compute training error
            train_evals = np.array([self.predict(v[:, np.newaxis], depth=j) for v in self.x_train.T]).T
            valid_evals = np.array([self.predict(v[:, np.newaxis], depth=j) for v in self.x_valid.T]).T

            # compute cost
            train_miss = 0
            if self.y_train.size > 0:
                train_miss = 1 - len(np.argwhere(train_evals != self.y_train)) / self.y_train.size
            valid_miss = 0
            if self.y_valid.size > 0:
                valid_miss = 1 - len(np.argwhere(valid_evals != self.y_valid)) / self.y_valid.size

            self.train_accuracies.append(train_miss)
            self.valid_accuracies.append(valid_miss)

    def predict(self, val, **kwargs):
        depth = self.depth
        if 'depth' in kwargs:
            depth = kwargs['depth']

        # search tree
        tree = copy.deepcopy(self.tree)
        d = 0
        while d < depth:
            split = tree.split
            dim = tree.dim
            if val[dim, :] <= split:
                tree = tree.left
            else:
                tree = tree.right
            d += 1

        # get final leaf value
        split = tree.split
        dim = tree.dim
        if val[dim, :] <= split:
            tree = tree.left_leaf
        else:
            tree = tree.right_leaf

        # return evaluation
        return tree(val)

    def evaluate_tree(self, val, depth):
```

```
        if depth > self.depth:
            return ('desired depth greater than depth of tree')
        tree = copy.deepcopy(self.tree)
        d = 0
        while d < depth:
            split = tree.split
            dim = tree.dim
            if val[dim, :] <= split:
                tree = tree.left
            else:
                tree = tree.right
            d += 1

        # get final leaf value
        split = tree.split
        dim = tree.dim
        if val[dim, :] <= split:
            tree = tree.left_leaf
        else:
            tree = tree.right_leaf
        return tree(val)

    def draw_fused_model(self, runs):
        # get visual boundary
        xmin1 = np.min(self.x[0, :])
        xmax1 = np.max(self.x[0, :])
        xgap1 = (xmax1 - xmin1) * 0.05
        xmin1 -= xgap1
        xmax1 += xgap1
        xmin2 = np.min(self.x[1, :])
        xmax2 = np.max(self.x[1, :])
        xgap2 = (xmax2 - xmin2) * 0.05
        xmin2 -= xgap2
        xmax2 += xgap2
        ind0 = np.argwhere(self.y == +1)
        ind0 = [v[1] for v in ind0]
        plt.scatter(self.x[0, ind0], self.x[1, ind0], s=60, color=self.colors[0], edgecolor='k',
linewidth=1, zorder=3)
        ind1 = np.argwhere(self.y == -1)
        ind1 = [v[1] for v in ind1]
        plt.scatter(self.x[0, ind1], self.x[1, ind1], s=60, color=self.colors[1], edgecolor='k',
linewidth=1, zorder=3)
        plt.xlim([xmin1, xmax1])
        plt.ylim([xmin2, xmax2])
```

```
        plt.title("Final Model of " + str(num_trees) + " Bagged Models")
        plt.xlabel(r'$x_1$', fontsize=14)
        plt.ylabel(r'$x_2$', rotation=0, fontsize=14, labelpad=10)
        s1 = np.linspace(xmin1, xmax1, 50)
        s2 = np.linspace(xmin2, xmax2, 50)
        a, b = np.meshgrid(s1, s2)
        a = np.reshape(a, (np.size(a), 1))
        b = np.reshape(b, (np.size(b), 1))
        h = np.concatenate((a, b), axis=1)
        a.shape = (np.size(s1), np.size(s2))
        b.shape = (np.size(s1), np.size(s2))
        t_ave = []
        for k in range(len(runs)):
            tree = runs[k]
            depth = tree.best_depth
            t = []
            for val in h:
                val = val[:, np.newaxis]
                out = tree.evaluate_tree(val, depth)
                t.append(out)
            t = np.array(t)
            t.shape = (np.size(s1), np.size(s2))
            col = np.random.rand(1, 3)
            plt.contour(s1, s2, t, linewidths=2.5, levels=[0], colors=self.plot_colors[k], zorder=2,
alpha=0.4)
            t_ave.append(t)
        t_ave = np.array(t_ave)
        t_ave1 = np.median(t_ave, axis=0)
        plt.contour(s1, s2, t_ave1, linewidths=3.5, levels=[0], colors='k', zorder=4, alpha=1)
        plt.show()


def plot(y, label):
    x = range(1, len(y[1]) + 1)
    colors = ['dimgray', 'coral', 'aquamarine', 'crimson', 'blueviolet', 'chartreuse']
    plt.title(label)
    for i in range(len(y)):
        plt.plot(x, y[i], marker='o', color=colors[i])
    plt.xticks(x, rotation=0)
    plt.xlabel("Depth of Decision Tree")
    plt.ylabel("Accuracy")
    plt.show()
```

```python
if __name__ == "__main__":
    # xx = np.load('testdata.npy')
    data_name = ['Correct.npy', 'Incorrect.npy', 'NoMask.npy', ]
    trees = []
    train_acc = []
    valid_acc = []
    num_trees = 5
    depth = 7
    train_portion = 0.67
    for i in range(num_trees):
        print("training fold: " + str(i))
        tree = Random_Forest_Algorithm(data_name, depth, train_portion=train_portion, img_size=[20, 20])
        trees.append(tree)
        train_acc.append(tree.train_accuracies)
        valid_acc.append(tree.valid_accuracies)
    # Compare the acc of training_set and validation_set
    plot(train_acc, label='Training set accuracy')
    plot(valid_acc, label='Validation set accuracy')
    # Draw 5+1 models all in one
    tree = Random_Forest_Algorithm(data_name, depth, train_portion=1, img_size=[20, 20])
    tree.draw_fused_model(runs=trees)
```

- **ClassificationStump.py**

```python
from autograd import numpy as np
import copy


left_his = []
right_his = []


# class for building regression stump
class Stump:
    def __init__(self, x, y):
        # globals
        self.x = x
        self.y = y
        # find best stump given input data
        self.make_stump()
```

```python
def counter(self, step, x, y):
    # compute predictions
    y_hat = step(x)[np.newaxis, :]

    # compute total counts
    vals, counts = np.unique(y, return_counts=True)

    # compute misclass on each class, compute balanced accuracy
    balanced = 0
    for i in range(len(vals)):
        v = vals[i]
        c = counts[i]
        ind = np.argwhere(y == v)
        miss_val = 1
        if ind.size > 0:
            ind = [a[1] for a in ind]
            miss = np.argwhere(y_hat[:, ind] != y[:, ind])
            if miss.size > 0:
                miss = len([a[1] for a in miss])
                miss_val = (1 - miss / c)
        balanced += miss_val
    balanced = balanced / len(vals)
    return balanced

### create prototype steps ###
def make_stump(self):
    # important constants: dimension of input N and total number of points P
    N = np.shape(self.x)[0]
    P = np.size(self.y)

    # begin outer loop - loop over each dimension of the input - create split points and dimensions
    acc_matrix_right = [0] * N
    acc_matrix_left = [0] * N
    best_split = np.inf
    best_dim = np.inf
    best_val = -np.inf
    best_left_leaf = []
    best_right_leaf = []
    best_left_ave = []
    best_right_ave = []
    best_step = []
    c_vals, c_counts = np.unique(self.y, return_counts=True)
    self.c_counts = c_counts
```

```python
for n in range(N):
    # make a copy of the n^th dimension of the input data (we will sort after this)
    x_n = copy.deepcopy(self.x[n, :])
    y_n = copy.deepcopy(self.y)

    # sort x_n and y_n according to ascending order in x_n
    sorted_inds = np.argsort(x_n, axis=0)
    # 将元素从小到大排列，提取对应得 index
    x_n = x_n[sorted_inds]
    y_n = y_n[:, sorted_inds]
    # loop over points and create stump in between each
    # in dimension n
    for p in range(P - 1):
        if y_n[:, p] != y_n[:, p + 1] and x_n[p] != x_n[p + 1]:
            # compute split point
            split = (x_n[p] + x_n[p + 1]) / float(2)

            ## determine most common label relative to proportion of each class present ##
            # compute various counts and decide on levels
            y_n_left = y_n[:, :p + 1]
            y_n_right = y_n[:, p + 1:]
            c_left_vals, c_left_counts = np.unique(y_n_left, return_counts=True)
            c_right_vals, c_right_counts = np.unique(y_n_right, return_counts=True)

            prop_left = []
            prop_right = []
            for i in range(np.size(c_vals)):
                val = c_vals[i]
                count = c_counts[i]

                val_ind = np.argwhere(c_left_vals == val)
                val_count = 0
                if np.size(val_ind) > 0:
                    val_count = c_left_counts[val_ind][0][0]
                prop_left.append(val_count / count)

                # check right side
                val_ind = np.argwhere(c_right_vals == val)
                val_count = 0
                if np.size(val_ind) > 0:
                    val_count = c_right_counts[val_ind][0][0]
                prop_right.append(val_count / count)

            # array it
```

```
                prop_left = np.array(prop_left)
                best_left = np.argmax(prop_left)
                left_ave = c_vals[best_left]
                best_acc_left = prop_left[best_left]
                # left = y_n_left.size / y_n.size

                prop_right = np.array(prop_right)
                best_right = np.argmax(prop_right)
                right_ave = c_vals[best_right]
                best_acc_right = prop_right[best_right]
                # right = y_n_right.size / y_n.size
                val = (best_acc_left + best_acc_right) / 2

                # define leaves
                left_leaf = lambda x, left_ave=left_ave, dim=n: np.array([left_ave for v in x[dim, :]])
                right_leaf = lambda x, right_ave=right_ave, dim=n: np.array([right_ave for v in
x[dim, :]])

                # create stump
                step = lambda x, split=split, left_ave=left_ave, right_ave=right_ave, dim=n: np.array(
                   [(left_ave if v <= split else right_ave) for v in x[dim, :]])

                # compute cost value on step
                # val = self.counter(step,self.x,self.y)

                if val > best_val:
                    acc_matrix_right = prop_right
                    acc_matrix_left = prop_left
                    best_left_leaf = copy.deepcopy(left_leaf)
                    best_right_leaf = copy.deepcopy(right_leaf)

                    best_dim = copy.deepcopy(n)
                    best_split = copy.deepcopy(split)
                    best_val = copy.deepcopy(val)
                    best_left_ave = copy.deepcopy(left_ave)
                    best_right_ave = copy.deepcopy(right_ave)
                    best_step = copy.deepcopy(step)

        # define globals
        self.step = best_step
        self.left_leaf = best_left_leaf
        self.right_leaf = best_right_leaf
        self.dim = best_dim
        self.split = best_split
```

```python
        # sort x_n and y_n according to ascending order in x_n
        sorted_inds = np.argsort(self.x[best_dim, :], axis=0)
        self.x = self.x[:, sorted_inds]
        self.y = self.y[:, sorted_inds]

        # cull out points on each leaf
        left_inds = np.argwhere(self.x[best_dim, :] <= best_split).flatten()
        right_inds = np.argwhere(self.x[best_dim, :] > best_split).flatten()

        self.left_x = self.x[:, left_inds]
        self.right_x = self.x[:, right_inds]
        self.left_y = self.y[:, left_inds]
        self.right_y = self.y[:, right_inds]
        self.number_mis_class_left = self.caculate_mis_class(acc_matrix_right, acc_matrix_left)[0]
        self.number_mis_class_right = self.caculate_mis_class(acc_matrix_right, acc_matrix_left)[1]
        right_his.append(self.number_mis_class_right)
        left_his.append(self.number_mis_class_left)


    def caculate_mis_class(self, prop_right, prop_left):
        leaf_label_ind_left = np.argmax(prop_left)
        left_label_count = self.c_counts[leaf_label_ind_left]
        leaf_label_ind_right = np.argmax(prop_right)
        right_label_count = self.c_counts[leaf_label_ind_right]
        mis_class_left = self.left_x.shape[1] - round(left_label_count * prop_left[leaf_label_ind_left])
        mis_class_right = self.right_x.shape[1] - round(right_label_count *
prop_right[leaf_label_ind_right])
        return mis_class_left, mis_class_right
```

- **Conventional_ML_Method.py**

```python
import sys
from lib.math_optimization_library import static_plotter
import autograd.numpy as np
from autograd.misc.flatten import flatten_func
from autograd import grad as gradient
from lib import edge_extract
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
```

```python
sys.path.append('..')

plotter = static_plotter.Visualizer()


def linear_model(x, w):
    a = w[0] + np.dot(x.T, w[1:])
    return a.T


def multiclass_softmax(w, x, y, iter):
    x_p = x[:, iter]
    y_p = y[:, iter]
    all_evals = linear_model(x_p, w)
    a = np.log(np.sum(np.exp(all_evals), axis=0))
    b = all_evals[y_p.astype(int).flatten(), np.arange(np.size(y_p))]
    cost = np.sum(a - b)
    return cost / float(np.size(y_p))


class FaceMask_Classification(object):
    def __init__(self, data_name, img_size):
        self.gray = []
        self.red = []
        self.blue = []
        self.green = []
        self.mismatching_his = []
        file_path = "../Data/Pixel" + str(img_size[0]) + "/"
        x, y = self.fetchData(file_path,data_name, img_size)
        self.feature_extraction(x, img_size)
        self.y = y
        self.x = np.array(self.gray).T
        self.shuffle_data(n_sample=4500, x=self.x, y=self.y)
        self.standard_normalizer(self.x_rand.T)
        self.x_rand = self.normalizer(self.x_rand.T).T
        self.x_edge = self.hog_extractor(self.x_rand)

        self.cost_function = multiclass_softmax

    @staticmethod
    def fetchData(file_path, data_name, img_size):
        y = []
        x = np.empty(shape=(0, img_size[0], img_size[1], 3))
        tag = 0
```

```python
    number = 0
    for name in data_name:
        data_subset = np.load(file_path+name)
        x = np.append(x, data_subset, axis=0)
        y.extend(np.shape(data_subset)[0] * [tag])
        number += np.shape(data_subset)[0]
        tag += 1
    return x, np.reshape(np.array(y), (1, number))

def feature_extraction(self, data, img_size):
    for i in range(np.shape(data)[0]):
        self.gray_image(data[i, :], img_size)

def gray_image(self, img, img_size):
    blue = []
    green = []
    red = []
    for i in range(img_size[0]):
        for j in range(img_size[1]):
            red.append(img[i][j][0])
            green.append(img[i][j][1])
            blue.append(img[i][j][2])
    gray = list(0.07 * np.array(blue) + 0.72 * np.array(green) + 0.21 * np.array(red))
    self.gray.append(gray)
    self.red.append(red)
    self.green.append(green)
    self.blue.append(blue)

def shuffle_data(self, n_sample, x, y):
    inds = np.random.permutation(y.shape[1])[:n_sample]
    self.x_rand = np.array(x)[:, inds]
    self.y_rand = y[:, inds]

def standard_normalizer(self, x):
    x_ave = np.nanmean(x, axis=1)[:, np.newaxis]
    x_std = np.nanstd(x, axis=1)[:, np.newaxis]
    self.normalizer = lambda data: (data - x_ave) / x_std

@staticmethod
def hog_extractor(x):
    kernels = np.array([
        [[-1, -1, -1],
         [0, 0, 0],
         [1, 1, 1]],
```

```
        [[-1, -1, 0],
         [-1, 0, 1],
         [0, 1, 1]],
        [[-1, 0, 1],
         [-1, 0, 1],
         [-1, 0, 1]],
        [[0, 1, 1],
         [-1, 0, 1],
         [-1, -1, 0]],
        [[1, 0, -1],
         [1, 0, -1],
         [1, 0, -1]],
        [[0, -1, -1],
         [1, 0, -1],
         [1, 1, 0]],
        [[1, 1, 1],
         [0, 0, 0],
         [-1, -1, -1]],
        [[1, 1, 0],
         [1, 0, -1],
         [0, -1, -1]]]])
    extractor = edge_extract.tensor_conv_layer()
    x_transformed = extractor.conv_layer(x.T, kernels).T
    return x_transformed

def gradient_descent(self, loss_fun, w, x_train, y_train, alpha, max_its, batch_size):
    g_flat, unflatten, w = flatten_func(loss_fun, w)
    grad = gradient(g_flat)
    num_train = y_train.size
    w_hist = [unflatten(w)]
    train_hist = [g_flat(w, x_train, y_train, np.arange(num_train))]
    num_batches = int(np.ceil(np.divide(num_train, batch_size)))
    for k in range(max_its):
        for b in range(num_batches):
            batch_inds = np.arange(b * batch_size, min((b + 1) * batch_size, num_train))
            grad_eval = grad(w, x_train, y_train, batch_inds)
            grad_eval.shape = np.shape(w)
            w = w - (alpha / (k + 1)) * grad_eval
        train_cost = g_flat(w, x_train, y_train, np.arange(num_train))
        w_hist.append(unflatten(w))
        train_hist.append(train_cost)
    return w_hist, train_hist

def misclass_counting(self, x, y, weight_his):
```

```
        mis_his = []
        for w in weight_his:
            all_evals = linear_model(x, w)
            y_predict = (np.argmax(all_evals, axis=0))[np.newaxis, :]
            count = np.shape(np.argwhere(y != y_predict))[0]
            mis_his.append(count)
        return mis_his

    # Plotting
    def confusion_matrix(self, mis_history, x, y, weight_his, labels, normalize=False, title='Confusion
Matrix',
                         precision="%0.f"):
        ind = np.argmin(mis_history)
        w_p = weight_his[ind]
        tick_marks = np.array(range(len(labels))) + 0.5
        all_evals = linear_model(x, w_p)
        y_predict = np.argmax(all_evals, axis=0)
        count = np.shape(np.argwhere(y != y_predict))[0]
        acc = 1 - (count / np.shape(all_evals)[1])
        print("the prediction accuracy is:" + str(acc))
        cm = confusion_matrix(y[0], y_predict)
        if normalize:
            cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
            title = "Normalized " + title
            precision = "%0.2f"
        plt.figure(figsize=(12, 8), dpi=120)
        ind_array = np.arange(len(labels))
        x, y = np.meshgrid(ind_array, ind_array)
        for x_val, y_val in zip(x.flatten(), y.flatten()):
            c = cm[y_val][x_val]
            if c > 0.0:
                plt.text(x_val, y_val, precision % (c,), color='k', fontsize=17, va='center',
ha='center')
        plt.gca().set_xticks(tick_marks, minor=True)
        plt.gca().set_yticks(tick_marks, minor=True)
        plt.gca().xaxis.set_ticks_position('none')
        plt.gca().yaxis.set_ticks_position('none')
        plt.grid(True, which='minor', linestyle='-')
        plt.gcf().subplots_adjust(bottom=0.15)
        plt.imshow(cm, interpolation='nearest', cmap='Blues')
        font = {'size': 13}
        plt.title(title, font)
        plt.colorbar()
        xlocations = np.array(range(len(labels)))
```

```python
        plt.xticks(xlocations, labels, rotation=0)
        plt.yticks(xlocations, labels)
        plt.ylabel('True label', font)
        plt.xlabel('Predicted label', font)
        plt.show()

    @staticmethod
    def weight_normalizer(w):
        w_norm = sum([v ** 2 for v in w[1:]]) ** 0.5
        return [v / w_norm for v in w]


if __name__ == "__main__":
    data_name = ['Correct.npy', 'Incorrect.npy', 'NoMask.npy', ]
    FaceMask = FaceMask_Classification(data_name, img_size=[20, 20])
    N = FaceMask.x_rand.shape[0]
    C = len(np.unique(FaceMask.y_rand))
    w = 0.1 * np.random.randn(N + 1, C)
    weight_his, cost_his = FaceMask.gradient_descent(FaceMask.cost_function, w, FaceMask.x_rand,
FaceMask.y_rand, alpha=0.02,
                                          max_its=100, batch_size=300)
    N = FaceMask.x_edge.shape[0]
    w = 0.1 * np.random.randn(N + 1, C)
    weight_edge_his, cost_edge_his = FaceMask.gradient_descent(FaceMask.cost_function, w,
FaceMask.x_edge, FaceMask.y_rand,
                                                    alpha=0.02,
                                                    max_its=100, batch_size=300)
    mis1 = FaceMask.misclass_counting(FaceMask.x_rand, FaceMask.y_rand, weight_his)
    FaceMask.confusion_matrix(mis1, FaceMask.x_rand, FaceMask.y_rand, weight_his,
                        labels=["Correct Masking", "Incorrect Masking", "No Masking"],
                        normalize=True,
                        title="Confusion matrix: Raw Dataset")

    mis2 = FaceMask.misclass_counting(FaceMask.x_edge, FaceMask.y_rand, weight_edge_his)
    FaceMask.confusion_matrix(mis2, FaceMask.x_edge, FaceMask.y_rand, weight_edge_his,
                        labels=["Correct Masking", "Incorrect Masking", "No Masking"],
                        normalize=True,
                        title="Confusion matrix: HoG Dataset")

    plotter.plot_mismatching_histories(histories=[mis1, mis2], start=1,
                                 labels=['Raw', 'Hog'],
                                 title="Training Mis-classification History")
    plotter.plot_cost_histories(histories=[cost_his, cost_edge_his], start=0,
```

```
                                labels=['Raw', 'Hog'],
                                title="Training Cost History")
```

➢ Static_plotter.py

```
import autograd.numpy as np
import matplotlib.pyplot as plt


def confusion_matrix(cm_collection, labels, precision="%0.f", normalize=True):
    tick_marks = np.array(range(len(labels))) + 0.5
    title = ["Image Size = 20", "Image Size = 50", "Image Size = 100"]
    plt.figure(figsize=(16, 8), dpi=120)
    for i in range(len(cm_collection)):
        cm = cm_collection[i]
        plt.subplot(1, len(cm_collection), i + 1)
        if normalize:
            cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
            title[i] = "Normalized " + title[i]
            precision = "%0.2f"
        ind_array = np.arange(len(labels))
        x, y = np.meshgrid(ind_array, ind_array)
        for x_val, y_val in zip(x.flatten(), y.flatten()):
            c = cm[y_val][x_val]
            if c > 0.0:
                plt.text(x_val, y_val, precision % (c,), color='k', fontsize=11, va='center', ha='center')
        plt.gca().set_xticks(tick_marks, minor=True)
        plt.gca().set_yticks(tick_marks, minor=True)
        plt.gca().xaxis.set_ticks_position('none')
        plt.gca().yaxis.set_ticks_position('none')
        plt.grid(True, which='minor', linestyle='-')
        plt.gcf().subplots_adjust(bottom=0.15)
        plt.imshow(cm, interpolation='nearest', cmap='Blues')
        font = {'size': 9}
        plt.title(title[i], font)
        xlocations = np.array(range(len(labels)))
        plt.xticks(xlocations, labels, rotation=45)
        if i == 0:
            plt.yticks(xlocations, labels)
        else:
            plt.yticks([])


    @staticmethod
def plot_mismatching_histories(histories=list, start=int, title='', **kwargs):
    # plotting colors
```

```
    colors = ['k', 'magenta', 'aqua', 'blueviolet', 'chocolate']

    # initialize figure
    fig = plt.figure(figsize=(10, 5))

    # create subplot with 1 panel
    gs = gridspec.GridSpec(1, 1)
    ax = plt.subplot(gs[0])

    # any labels to add?
    labels = [' ', ' ', ' ']
    if 'labels' in kwargs:
        labels = kwargs['labels']

    # plot points on cost function plot too?
    points = False
    if 'points' in kwargs:
        points = kwargs['points']

    # run through input histories, plotting each beginning at 'start' iteration
    for c in range(len(histories)):
        history = histories[c]
        label = 0
        if c == 0:
            label = labels[0]
        elif c == 1:
            label = labels[1]
        else:
            label = labels[2]

        # check if a label exists, if so add it to the plot
        if np.size(label) == 0:
            ax.plot(np.arange(start, len(history), 1), history[start:], linewidth=3 * 0.8 ** c,
color=colors[c])
        else:
            ax.plot(np.arange(start, len(history), 1), history[start:], linewidth=3 * 0.8 ** c,
color=colors[c],
                    label=label)

            # check if points should be plotted for visualization purposes
        if points:
            ax.scatter(np.arange(start, len(history), 1), history[start:], s=90, color=colors[c],
edgecolor='w',
                       linewidth=2, zorder=3)
```

```python
        # clean up panel
    xlabel = 'step $k$'
    if 'xlabel' in kwargs:
        xlabel = kwargs['xlabel']
    ylabel = '$Number\ of\ misclassification$'
    if 'ylabel' in kwargs:
        ylabel = kwargs['ylabel']
    ax.set_xlabel(xlabel, fontsize=14)
    ax.set_ylabel(ylabel, fontsize=10, rotation=90, labelpad=25)
    if np.size(label) > 0:
        anchor = (1, 1)
        if 'anchor' in kwargs:
            anchor = kwargs['anchor']
        plt.legend(loc='upper right', bbox_to_anchor=anchor)
        # leg = ax.legend(loc='upper left', bbox_to_anchor=(1.02, 1), borderaxespad=0)

    ax.set_xlim([start - 0.5, len(history) - 0.5])
    plt.title(title, fontsize=16)
    plt.show()


@staticmethod
def plot_cost_histories(histories=list, start=int, title='', **kwargs):
    # plotting colors
    colors = ['k', 'magenta', 'aqua', 'blueviolet', 'chocolate']

    # initialize figure
    fig = plt.figure(figsize=(10, 5))

    # create subplot with 1 panel
    gs = gridspec.GridSpec(1, 1)
    ax = plt.subplot(gs[0])

    # any labels to add?
    labels = [' ', ' ', ' ']
    if 'labels' in kwargs:
        labels = kwargs['labels']

    # plot points on cost function plot too?
    points = False
    if 'points' in kwargs:
        points = kwargs['points']
```

```
    # run through input histories, plotting each beginning at 'start' iteration
    for c in range(len(histories)):
        history = histories[c]
        label = 0
        if c == 0:
            label = labels[0]
        elif c == 1:
            label = labels[1]
        else:
            label = labels[2]

        # check if a label exists, if so add it to the plot
        if np.size(label) == 0:
            ax.plot(np.arange(start, len(history), 1), history[start:], linewidth=3 * 0.8 ** c,
color=colors[c])
        else:
            ax.plot(np.arange(start, len(history), 1), history[start:], linewidth=3 * 0.8 ** c,
color=colors[c],
                    label=label)

            # check if points should be plotted for visualization purposes
        if points:
            ax.scatter(np.arange(start, len(history), 1), history[start:], s=90, color=colors[c],
edgecolor='w',
                       linewidth=2, zorder=3)

            # clean up panel
    xlabel = 'step $k$'
    if 'xlabel' in kwargs:
        xlabel = kwargs['xlabel']
    ylabel = r'$g\left(\mathbf{w}^k\right)$'
    if 'ylabel' in kwargs:
        ylabel = kwargs['ylabel']
    ax.set_xlabel(xlabel, fontsize=14)
    ax.set_ylabel(ylabel, fontsize=14, rotation=0, labelpad=25)
    if np.size(label) > 0:
        anchor = (1, 1)
        if 'anchor' in kwargs:
            anchor = kwargs['anchor']
        plt.legend(loc='upper right', bbox_to_anchor=anchor)
        # leg = ax.legend(loc='upper left', bbox_to_anchor=(1.02, 1), borderaxespad=0)

    ax.set_xlim([start - 0.5, len(history) - 0.5])
```

```
    plt.title(title, fontsize=16)
    plt.show()
```

- **lib_fun.py**

```
import numpy as np
import matplotlib.pyplot as plt
import cv2
import sklearn.svm as svm
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn import metrics
from skimage import feature
import os

def RGBtoGray(data):
    new_data = np.zeros((1, data.shape[1], data.shape[2]))
    for im in data:
        new_data = np.append(new_data,
cv2.cvtColor(im.astype('float32'),cv2.COLOR_RGB2GRAY).reshape(1,data.shape[1], data.shape[2]), axis=0)
    return new_data[1:,:,:]

def RGBtoHOG(data):
    new_data = np.zeros((1, data.shape[1], data.shape[2]))
    for im in data:
        fd, hog_image = feature.hog(im, orientations=8, pixels_per_cell=(2, 2),
                    cells_per_block=(1, 1), visualize=True, multichannel=True)
        new_data = np.append(new_data, hog_image.reshape((1, data.shape[1], data.shape[2])), axis=0)
    return new_data[1:,:,:]

def GRAYtoEDGE(data, sigma):
    new_data = np.zeros((1, data.shape[1], data.shape[2]))
    for im in data:
        edges = feature.canny(im, sigma=sigma)
        new_data = np.append(new_data, edges.reshape((1, data.shape[1], data.shape[2])), axis=0)
    return new_data[1:,:,:]

def Flatten(images):
    images = images.reshape(images.shape[0], -1)
```

```python
    return images

def load(data_path):
    for i, set in enumerate(['Correct', 'Incorrect', 'NoMask']):
        if i == 0:
            data = np.load(data_path+set+'.npy')
            labels = np.array([i]*len(data))
            i += 1
        else:
            data_ = np.load(data_path+set+'.npy')
            data = np.append(data,data_, axis=0)
            labels = np.append(labels, np.array([i]*len(data_)), axis=0)
    print('X shape: {}, Y shape: {}'.format(data.shape, np.array(labels).shape))
    return data, labels

def VisualizeRGB(data20, data50, data100):
    f, axes = plt.subplots(1,9, figsize=(18,2))
    axes[0].imshow(data20[0,:,:,:].astype('uint8'))
    axes[1].imshow(data20[2000,:,:,:].astype('uint8'))
    axes[2].imshow(data20[-1,:,:,:].astype('uint8'))
    axes[3].imshow(data50[0,:,:,:].astype('uint8'))
    axes[4].imshow(data50[2000,:,:,:].astype('uint8'))
    axes[5].imshow(data50[-1,:,:,:].astype('uint8'))
    axes[6].imshow(data100[0,:,:,:].astype('uint8'))
    axes[7].imshow(data100[2000,:,:,:].astype('uint8'))
    axes[8].imshow(data100[-1,:,:,:].astype('uint8'))
    for ax in axes:
        ax.set_xticks([])
        ax.set_yticks([])

def VisualizeGray(data20, data50, data100):
    f, axes = plt.subplots(1,9, figsize=(18,2))
    axes[0].imshow(data20[0,:,:].astype('uint8'),cmap ='gray')
    axes[1].imshow(data20[2000,:,:].astype('uint8'),cmap ='gray')
    axes[2].imshow(data20[-1,:,:].astype('uint8'),cmap ='gray')
    axes[3].imshow(data50[0,:,:].astype('uint8'),cmap ='gray')
    axes[4].imshow(data50[2000,:,:].astype('uint8'),cmap ='gray')
    axes[5].imshow(data50[-1,:,:].astype('uint8'),cmap ='gray')
    axes[6].imshow(data100[0,:,:].astype('uint8'),cmap ='gray')
    axes[7].imshow(data100[2000,:,:].astype('uint8'),cmap ='gray')
    axes[8].imshow(data100[-1,:,:].astype('uint8'),cmap ='gray')
    for ax in axes:
        ax.set_xticks([])
        ax.set_yticks([])
```

```python
def plot_confusion_matrix(cm_collection, labels=["Correct", "Incorrect", "No"], precision="%0.f",
normalize=True):
    tick_marks = np.array(range(len(labels))) + 0.5
    title = ["20*20", "50*50", "100*100"]
    plt.figure(figsize=(4, 2), dpi=120)
    for i in range(len(cm_collection)):
        cm = cm_collection[i]
        plt.subplot(1, len(cm_collection), i + 1)
        if normalize:
            cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
            title[i] = title[i]
            precision = "%0.2f"
        ind_array = np.arange(len(labels))
        x, y = np.meshgrid(ind_array, ind_array)
        for x_val, y_val in zip(x.flatten(), y.flatten()):
            c = cm[y_val][x_val]
            if c > 0.0:
                plt.text(x_val, y_val, precision % (c,), color='k', fontsize=8, va='center', ha='center')
        plt.gca().set_xticks(tick_marks, minor=True)
        plt.gca().set_yticks(tick_marks, minor=True)
        plt.gca().xaxis.set_ticks_position('none')
        plt.gca().yaxis.set_ticks_position('none')
        plt.grid(True, which='minor', linestyle='-')
        plt.gcf().subplots_adjust(bottom=0.15)
        plt.imshow(cm, interpolation='nearest', cmap='Blues')
        font = {'size': 9}
        plt.title(title[i], font)
        xlocations = np.array(range(len(labels)))
        plt.xticks(xlocations, labels, rotation=45, fontsize=6)
        if i == 0:
            plt.yticks(xlocations, labels, fontsize=6)
        else:
            plt.yticks([])
    plt.show()
```