MAMAN 12, intro to computer vision,22928

Ayala Shaubi-Mann

200244242

Data sets

Iris

This data sets consists of 3 different types of irises' (Setosa, Versicolour, and Virginica) petal and sepal length and width, stored in a 150x4 numpy.ndarray

The rows being the samples and the columns being: Sepal Length, Sepal Width, Petal Length and Petal Width.

The Predicted attribute is the class of iris plant.

EMNIST

Data files train.csv and test.csv contain gray-scale images of hand-drawn digits, from zero through nine.

Each image is 28 pixels in height and 28 pixels in width, for a total of 784 pixels in total. Each pixel has a single pixel-value associated with it, indicating the lightness or darkness of that pixel, with higher numbers meaning darker. This pixel-value is an integer between 0 and 255, inclusive.

Data composed out of training set, 42K images, with their digit labels, and test set of 28K images without labels.

Figure 1-Sample of images from MNIST training set

KNN

Given a new point (image), find the k closest images from training data.

Each image represented using a combination of d features - a d-dim column vector called a feature vector.

Basic representation of an image will be its gray-scale pixels values vector.

Distance between images is calculated using distance function. I will use the Euclidian (L2) distance function.

As preprocessing, for each training image, I will extract feature vector.

Given a new image do as follows:

- Calc distance between the input image to each of the training images
- Sort training images according to their distance from input image, smallest to largest.
- Pull top K training images
- Among these K neighbors, count the number of data points to each category

- Assign the new data point to the category where you counted the most neighbors

In this assignment, I have tested different values of K, from 1 to 20.

To execute KNN, I will use python package, sklearn, implementation.

KNN classification

Script: knn_cls.py

```
import time

import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.neighbors import KNeighborsClassifier

def knn(trainData, valData, trainLabels, valLabels):
    # initialize the values of k for our k-Nearest Neighbor classifier along with
the
    kVals = range(1, 20, 2)
    # list of accuracies for each value of k
    accuracies = []

    for k in kVals:
        print("k = " + str(k) + " begin ")
        start = time.time()
        # train the k-Nearest Neighbor classifier with the current value of `k`
        model = KNeighborsClassifier(n_neighbors=k)
        model.fit(trainData, trainLabels)
        valPred = model.predict(valData)
        valPred_prob = model.predict_proba(valData)

        # evaluate the model and update the accuracies list
        score = model.score(valData, valLabels)
        accuracies.append(score)
        end = time.time()

        # output performance
        print("classification report:")
        print(classification_report(valLabels, valPred))
        print("confusion matrix:")
        print(confusion_matrix(valLabels, valPred))
        print("k=%d, accuracy=%.2f%%" % (k, score * 100))
        print("Complete time: " + str(end - start) + " Secs.")

    # plot accuracy by K values
    plt.plot(kVals, accuracies)
    plt.xlabel('K value (number of neighbors)')
    plt.ylabel('accuracy of validation set (25%)')
    plt.show()

    # find the value of k that has the largest accuracy
    i = int(np.argmax(accuracies))
    print("k=%d achieved highest accuracy of %.2f%% on validation data" % (kVals[i],
accuracies[i] * 100))
    model = KNeighborsClassifier(n_neighbors=kVals[i])
    model.fit(trainData, trainLabels)
    valPred = model.predict(valData)
    valPred_prob = model.predict_proba(valData)
    return valPred, valPred_prob
```

SVM

The purpose is to maximize the margin around the separating hyperplane.

Decision function fully specified by a subset of training samples, which are the supported vectors.

For a 2-class, linearly separable, problem we will perform next steps:

- Normalize points to be with 0 average.
- Compute the convex hull of the positive points, and convex hull of negative points
- For each pair if points, one on positive hull and the other on the negative hull, compute the margin.
- Choose the largest margin possible (which is the minimum margin received)

If we have noises in our dataset, we can use slack variable, regularization parameter C, for misclassified points. So we can choose the weight to give to misclassified points.

Small C allows constraints to be easily ignored and provide large margin.

Large C makes constraints hard to ignore and will create narrow margin.

SVM has been formulated as a constrained optimization problem over W (hyperplane parameters vector), and ε:

$$\min_{w \in R^d, \varepsilon_i \in R^+} ||w||^2 + C \sum_{1}^{N} \varepsilon_i \;\; subjected\ to\ y_i(w^T x_i + b) \geq 1 - \varepsilon_i\ for\ i = 1 \dots N$$

Or equivalently:

$$\min_{w \in R^d, \varepsilon_i \in R^+} ||w||^2 + C \sum_{1}^{N} \max\big(0, 1 - y_i f(x_i)\big)$$

When $y_i f(x_i) > 1$ means point is outside of the margin, $y_i f(x_i) = 1$ means point is on the margin and $y_i f(x_i) < 1$ means point violates margin constraint and contribution to loss higher than 0.

To handle data which isn't linearly separable, we have multiple optional solutions.

In this case I will test the use of polinomial kernel and a Radial basis function kernel, in addition to linear kernel.

I will test different values for C ("soft margin" parameter) from c=10^(-20) to c=100.

To execute KNN, I will use python package, sklearn, implementation.

As SVM is a binary classifier, to train multi-class svm, I will use one-vs-one scheme (sklearn.svm.svc default).

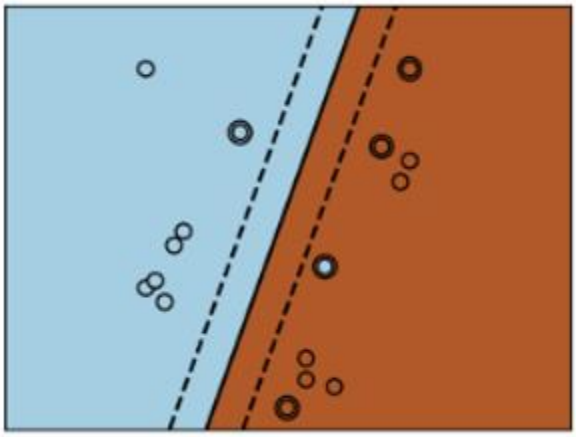


Figure 2- Linear kernel SVM

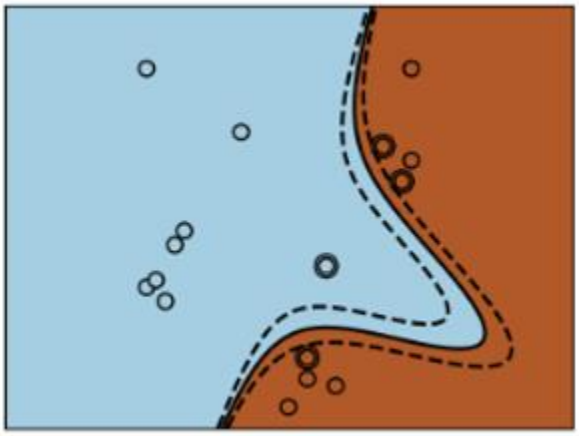


Figure 3- polinomial kernel SVM

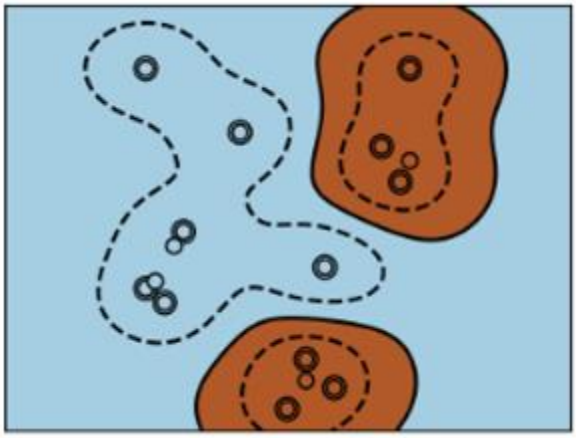


Figure 4- Radial basis function kernel SVM

SVM classification

Script: svm_cls.py

```
import time

import matplotlib.pyplot as plt
import numpy as np
from sklearn import svm
from sklearn.metrics import classification_report, confusion_matrix

from mmn2 import utils

def SVM(trainData, valData, trainLabels, valLabels):
    # initialize the values of c for our SVM classifier
    CVals = [10**(-20),10**(-10),0.001,0.1,1,10,100]
    # list of accuracies for each value of c
    accuracies = []
    best_c = 0
    best_kernel = 'linear'
    best_accuracy = 0

    for fig_num, kernel in enumerate(('poly','linear','rbf')):
        print("kernal = " + kernel + " begin ")
        for c in CVals:
            print("C = " + str(c) + " begin ")
            start = time.time()
            # train the SVM classifier with the current value of `c`
            model = svm.SVC(kernel=kernel, C=c, degree=3, probability=True,gamma=10)
            model.fit(trainData, trainLabels)
            valPred = model.predict(valData)
            valPred_prob = model.predict_proba(valData)

            # evaluate the model and update the accuracies list
            score = model.score(valData, valLabels)
            accuracies.append(score)
            end = time.time()

            # output performance
            print("classification report:")
            print(classification_report(valLabels, valPred))
            print("confusion matrix:")
            print(confusion_matrix(valLabels, valPred))
            print("c=%d, accuracy=%.2f%%" % (c, score * 100))
            print("Complete time: " + str(end - start) + " Secs.")
        # plot accuracy by c values
        plt.figure(fig_num)
        plt.clf()
        plt.plot(CVals, accuracies)
        plt.xlabel('c value (penalty)')
        plt.ylabel('accuracy of validation set (25%)')
        plt.title(kernel)

        i = int(np.argmax(accuracies))
        print("c=%.2f%% achieved highest accuracy of %.2f%% on validation data" %
                (CVals[i], accuracies[i] * 100))
        if accuracies[i] > best_accuracy:
            best_c = CVals[i]
            best_kernel = kernel
            best_accuracy = accuracies[i]
        accuracies = []
    plt.show()
```

```
    # find the value of c & kernel that has the largest accuracy
    model = svm.SVC(kernel=best_kernel, C=best_c, degree=3, probability=True)
    model.fit(trainData, trainLabels)
    valPred = model.predict(valData)
    valPred_prob = model.predict_proba(valData)
    return valPred, valPred_prob
```

Tested features

Image pixels

Use pixels vector as features upon the Euclidian distance will operate (in KNN or SVM).

HOG

The technique counts occurrences of gradient orientation in localized portions of an image. This method is similar to that of edge orientation histograms, scale-invariant feature transform descriptors, and shape contexts, but differs in that it is computed on a dense grid of uniformly spaced cells and uses overlapping local contrast normalization for improved accuracy.
the first step of calculation is the computation of the gradient values where the most common method is to apply the 1-D centered, point discrete derivative mask in one or both of the horizontal and vertical directions.

The second step of calculation is creating the cell histograms. Each pixel within the cell casts a weighted vote for an orientation-based histogram channel based on the values found in the gradient computation. The cells themselves can either be rectangular or radial in shape, and the histogram channels are evenly spread over 0 to 180 degrees

To account for changes in illumination and contrast, the gradient strengths must be locally normalized, which requires grouping the cells together into larger, spatially connected blocks. The HOG descriptor is then the concatenated vector of the components of the normalized cell histograms from all of the block regions.

To test this features with KNN and SVM classifiers, I used 3 orientation bins, cell size of 2*2, 4*4 cells per block with L2 normalization.

Tested on MNIST dataset.

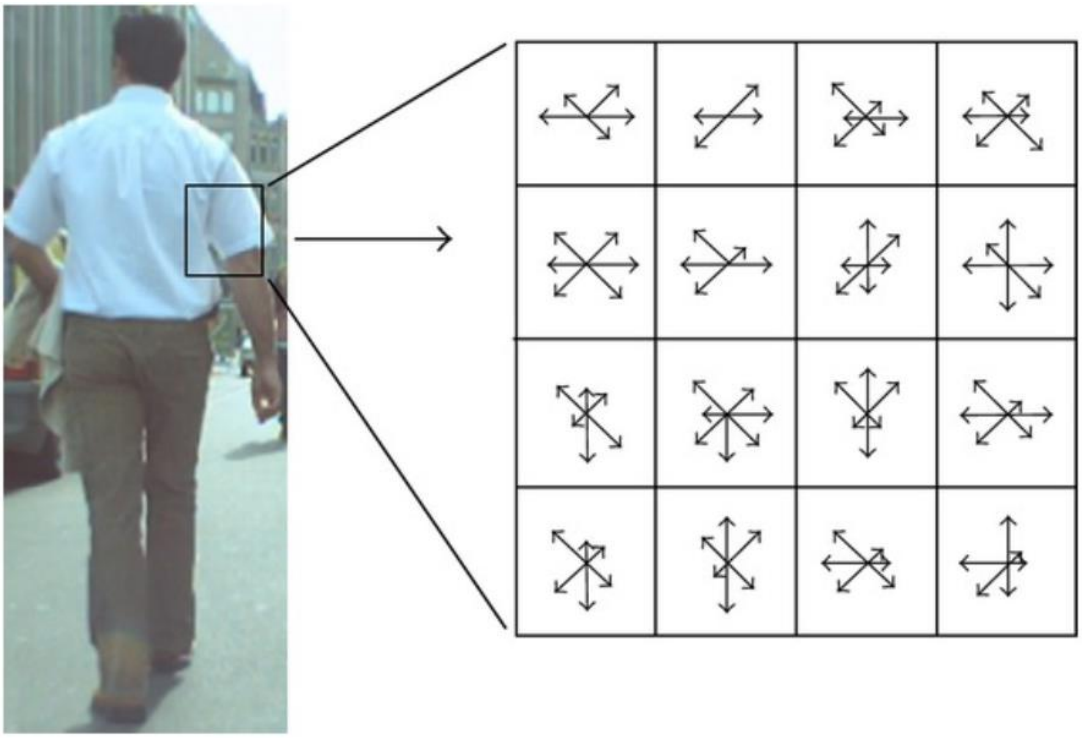


Figure 5-Extraction of HOG features

Get hog features

Python file: features.py

```
import cv2
import numpy as np
import pandas as pd
import scipy.misc
from skimage import feature

def hog_representation(image, orientations, pixelsPerCell, cellsPerBlock, block_norm):
    hist = feature.hog(image, orientations=orientations,
                       pixels_per_cell=pixelsPerCell,
                       cells_per_block=cellsPerBlock,
                       block_norm=block_norm)
    return hist

def hog_batch_representation(images, orientations, pixelsPerCell, cellsPerBlock,
          block_norm):
    result = []
    for image in images:
        # describe the image and update the data matrix
        hist = hog_representation(image, orientations, pixelsPerCell, cellsPerBlock,
               block_norm)
        result.append(hist)
    return result
```

Utils

Script: utils.py	Multiclass ROC curve
imports	

```
from itertools import cycle

import matplotlib.pyplot as plt
import numpy as np
from scipy import interp
from sklearn.decomposition import PCA
from sklearn.metrics import roc_curve, auc
```

Confusion matrix plot

```
def plot_confusion(confusion_matrix):
    plt.figure()
    plt.matshow(confusion_matrix)
    plt.title('Confusion Matrix for Validation Data')
    plt.colorbar()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.show()
```

Multiclass ROC plot

```
def plot_roc(test_y, tesy_y_prob, title=''):
    n_classes = np.unique(test_y).shape[0]
    fpr = dict()
    tpr = dict()
    roc_auc = dict()
    for i in range(n_classes):
        fpr[i], tpr[i], _ = roc_curve(np.where(test_y == i, 1, 0), tesy_y_prob[:, i])
        roc_auc[i] = auc(fpr[i], tpr[i])
    # Compute micro-average ROC curve and ROC area

    test_y_multi = tesy_y_prob
    for i in range(n_classes):
        test_y_multi[:, i] = np.where(test_y == i, 1, 0)
    fpr["micro"], tpr["micro"], _ = roc_curve(test_y_multi.ravel(), tesy_y_prob.ravel
    roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])

    # Compute macro-average ROC curve and ROC area

    # First aggregate all false positive rates
    all_fpr = np.unique(np.concatenate([fpr[i] for i in range(n_classes)]))

    # Then interpolate all ROC curves at this points
    mean_tpr = np.zeros_like(all_fpr)
    for i in range(n_classes):
        mean_tpr += interp(all_fpr, fpr[i], tpr[i])

    # Finally average it and compute AUC
    mean_tpr /= n_classes

    fpr["macro"] = all_fpr
    tpr["macro"] = mean_tpr
    roc_auc["macro"] = auc(fpr["macro"], tpr["macro"])

    # Plot all ROC curves
    lw = 2
    plt.figure()
    plt.plot(fpr["micro"], tpr["micro"],
             label='micro-average ROC curve (area = {0:0.2f})'
```

```
                  ''.format(roc_auc["micro"]),
             color='deeppink', linestyle=':', linewidth=4)

    plt.plot(fpr["macro"], tpr["macro"],
             label='macro-average ROC curve (area = {0:0.2f})'
                  ''.format(roc_auc["macro"]),
             color='navy', linestyle=':', linewidth=4)

    colors = cycle(['aqua', 'darkorange', 'cornflowerblue'])
    for i, color in zip(range(n_classes), colors):
        plt.plot(fpr[i], tpr[i], color=color, lw=lw,
                 label='ROC curve of class %d (area = %.4f%%)' % (i, roc_auc[i]))

    plt.plot([0, 1], [0, 1], 'k--', lw=lw)
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('ROC multi-class')
    plt.legend(loc="lower right")
    plt.title(title)
    plt.show()
```

Iris classification

To classify Iris data set, I have used given features of Sepal Length, Sepal Width, Petal Length and Petal Width, 4d feature vector, with L2 distance function.

Different configurations tested for knn and SVM.

Iris classification

Script: iris_classification.py

```
i
mport argparse

import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

from mmn2 import knn_cls
from mmn2 import svm_cls
from mmn2 import utils

# 1. load data set
iris = load_iris()
X, y = iris.data, iris.target
np.random.seed(10)
n_sample = len(X)
order = np.random.permutation(n_sample)
X = X[order]
y = y[order]

# 2. split train ans test sets
(trainData, valData, trainLabels, valLabels) = train_test_split(X,
                                                                y, test_size=0.3,
                                                                random_state=42)
```

```
ap = argparse.ArgumentParser()
ap.add_argument("-m", "--model", required=True,
                help="name of classification model- one of knn or svm")
args = vars(ap.parse_args())

# 3. classify according to selected model
print('execute classification using {model}'.format(model=args["model"]))
if args["model"] == 'knn':
    valPred, valPred_prob = knn_cls.knn(trainData, valData, trainLabels, valLabels)
else:
    valPred, valPred_prob = svm_cls.SVM(trainData, valData, trainLabels, valLabels)

# 4. plot ROC,AUC
utils.plot_roc(valLabels, valPred_prob, title="Iris {model} ROC".format(model=args["model"]))

pd.DataFrame(trainData).to_csv("Iris\\trainData.csv")
pd.DataFrame(valData).to_csv("Iris\\valData.csv")
pd.DataFrame(trainLabels).to_csv("Iris\\trainLabels.csv")
pd.DataFrame(valLabels).to_csv("Iris\\valLabels.csv")
pd.DataFrame(valPred).to_csv("Iris\\valPred_{model}.csv".format(model=args["model"]))
pd.DataFrame(valPred_prob).to_csv("Iris\\valPred_prob_{model}.csv".format(model=args["model"]))
```

KNN

Execute Iris classification using knn
iris_classification.py –m knn

The execution include test of multiple K values for knn classification, as seen in figure 6.

With K=5, we can reach up to **97.78%** accuracy.

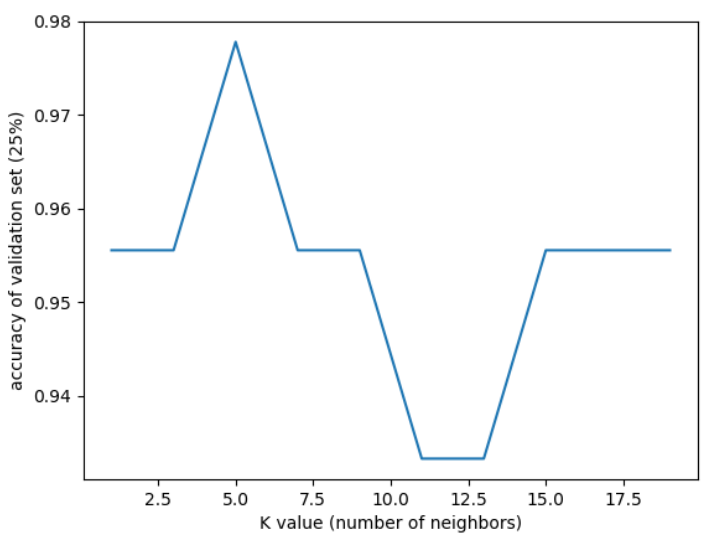


Figure 6- accuracy of validation set with different K values

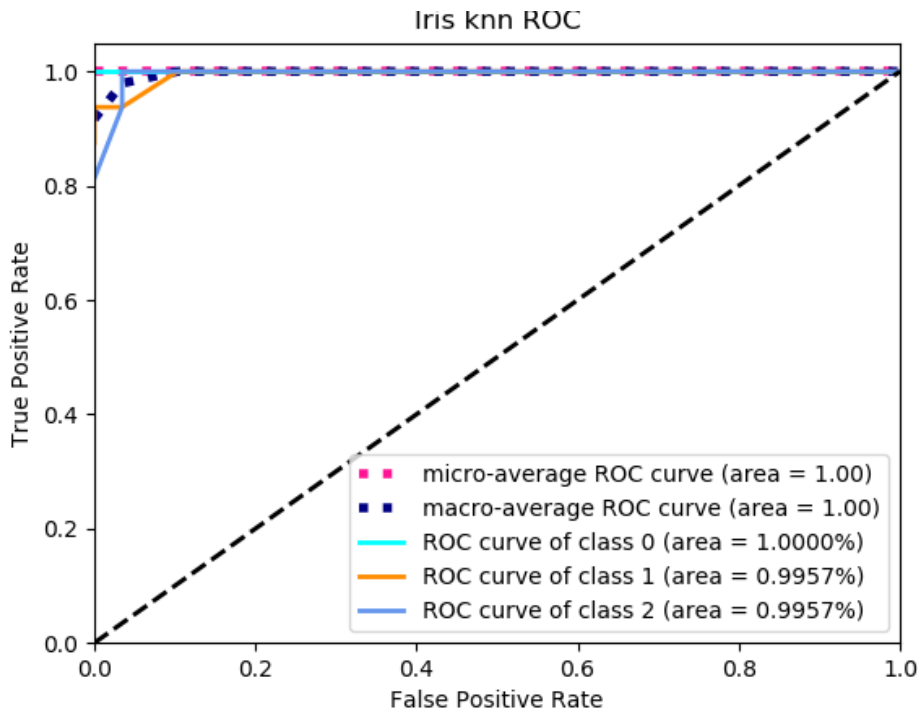


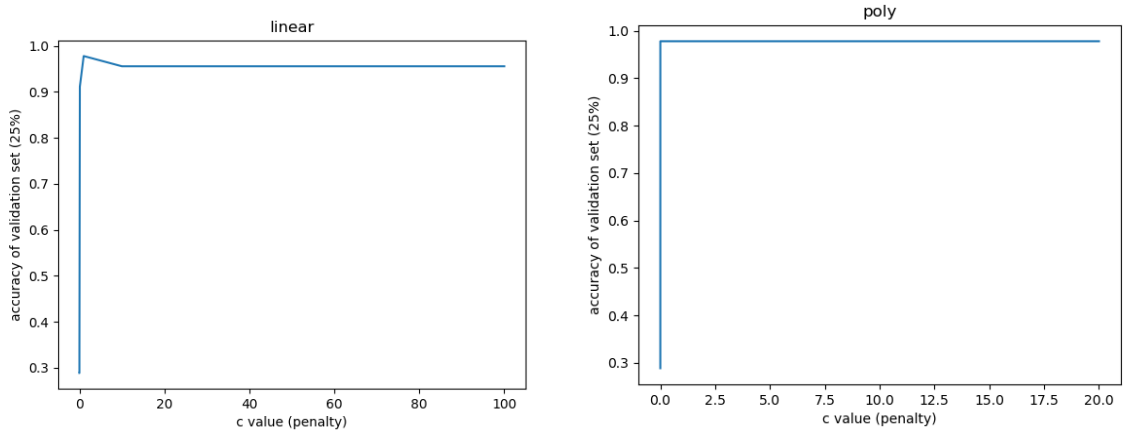
Figure 7- Iris ROC, using knn classification

SVM

Execute Iris classification using svm
iris_classification.py –m svm

The execution include test of multiple C values for svm classification and different kernels types, as seen in figure 8.

The best penalty value achieved accuracy of **97.78%** is C=1, using Linear kernel.



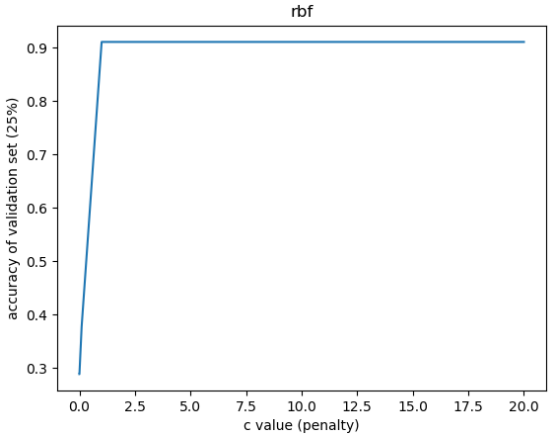


Figure 8- Accuracy per C value for different kernels

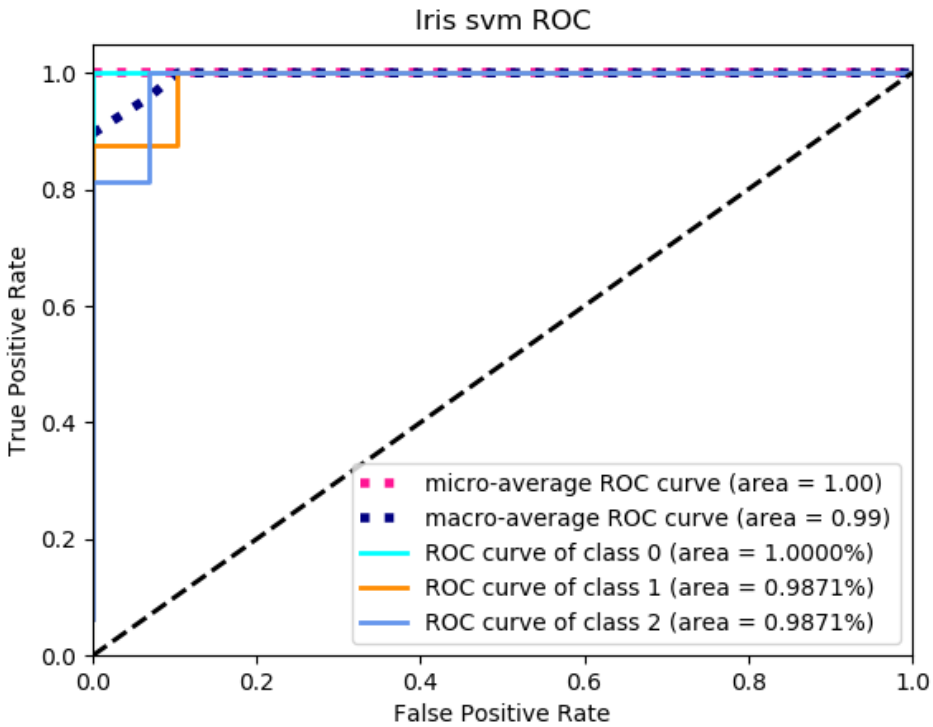


Figure 9- Iris ROC, using svm classification

Confusion matrix:

13	0	0
0	15	1
0	0	16

Classification report:

label	precision	recall	f1-score	support
0	1	1	1	13
1	1	0.94	0.97	16
2	0.94	1	0.97	15
avg / total	0.98	0.98	0.98	45

MNIST classification

To classify MNIST data set, I tested multiple features:

Starting from original image pixels and following with hog features on top of it.

Different configurations tested for knn and SVM.

Classifying EMNIST dataset

Script: MNIST_classification.py

```
import argparse

import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

from mmn2 import features
from mmn2 import knn_cls
from mmn2 import svm_cls
from mmn2 import utils

mnist_dir = "MNIST\\"
test = True
test_size = 20000 # to shorten execution time, sample from complete db

# 1. load data set
train_data = pd.read_csv(mnist_dir + "train.csv")
train_data.reset_index()

np.random.seed(10)
n_sample = len(train_data)
order = np.random.permutation(n_sample)
train_data = train_data.iloc[order]

if test:
    train_data = train_data.head(test_size)

X = train_data.drop("label", 1)
y = train_data['label']

# get features
# construct the argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-m", "--model", required=True,
                help="name of classification model- one of knn or svm")
ap.add_argument("-f", "--feature", required=True,
                help="name of used feature- one of baseline, hog or sift")
args = vars(ap.parse_args())

# split train ans test sets according to selected feature
if args["feature"] == 'baseline':
```

```
    (trainData, valData, trainLabels, valLabels) = train_test_split(X,
                                                                    y, test_size=0.3,
random_state=42)
elif args["feature"] == 'hog':
    digits = np.asarray(X).reshape((X.shape[0], 28, 28))
    hog_rep = features.hog_batch_representation(digits, orientations=3, pixelsPerCell=(2, 2),
                                                cellsPerBlock=(4, 4), block_norm='L2-Hys')
    (trainData, valData, trainLabels, valLabels) = train_test_split(np.array(hog_rep),
                                                                    y, test_size=0.3,
random_state=42)
elif args["feature"] == 'sift':
    images = X
    X_sift = features.sift_batch_representation(images)
    (trainData, valData, trainLabels, valLabels) = train_test_split(X_sift,
                                                                    y, test_size=0.3,
random_state=42)
else:
    digits = np.asarray(X).reshape((X.shape[0], 28, 28))
    hog_rep = features.hog_batch_representation(digits, orientations=3, pixelsPerCell=(2, 2),
                                                cellsPerBlock=(4, 4), block_norm='L2-Hys')
    images = X
    X_sift = features.sift_batch_representation(images)
    comb = np.concatenate((hog_rep, X_sift), axis=1)
    (trainData, valData, trainLabels, valLabels) = train_test_split(comb,
                                                                    y, test_size=0.3,
random_state=42)

# classify according to selected model
print('execute classification using {model}'.format(model=args["model"]))
if args["model"] == 'knn':
    valPred, valPred_prob = knn_cls.knn(trainData, valData, trainLabels, valLabels)
else:
    valPred, valPred_prob = svm_cls.SVM(trainData, valData, trainLabels, valLabels)
###ROC,AUC
utils.plot_roc(valLabels, valPred_prob, title="MNIST {model} ROC".format(model=args["model"]))
```

KNN

Baseline

As a baseline, I have utilized **raw pixel intensities** as inputs to KNN, using different K values.

Execute MNIST classification using knn, baseline
MNIST_classification.py –m knn –f baseline

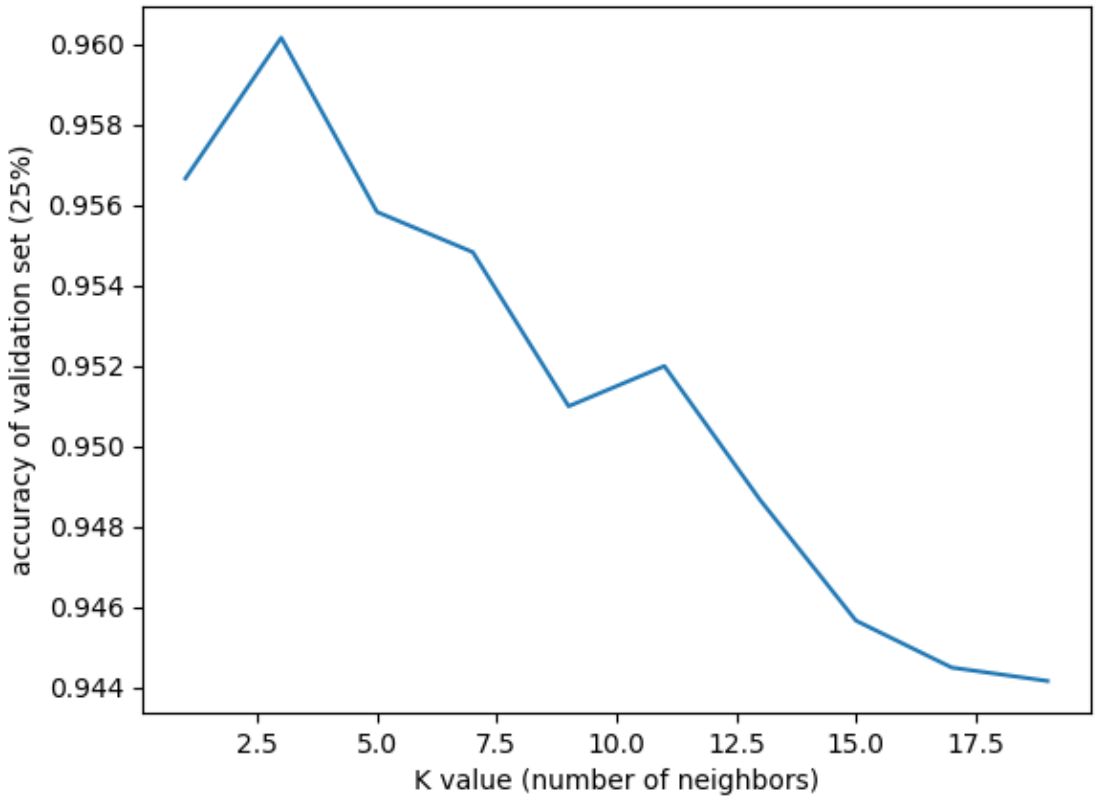


Figure 10- knn accuracy according to different k values, MNIST

k=3 achieved highest accuracy of **96.02%** on validation data.

Classification report:

label	precision	recall	f1-score	support
0	0.97	0.99	0.98	587
1	0.94	1	0.97	651
2	0.97	0.95	0.96	609
3	0.94	0.95	0.95	642
4	0.97	0.97	0.97	550
5	0.97	0.95	0.96	550
6	0.98	0.98	0.98	613
7	0.95	0.97	0.96	641
8	0.97	0.89	0.93	575
9	0.94	0.94	0.95	582
avg / total	0.96	0.96	0.96	6000

Confusion matrix:

579	1	0	0	0	0	5	1	0	1
0	650	0	0	0	0	0	1	0	0
3	10	581	0	0	0	0	9	5	1
1	2	7	613	0	6	0	3	7	3
0	3	0	0	536	0	1	0	0	10
1	3	0	10	2	522	6	0	2	4
3	2	0	0	2	2	603	0	1	0
1	7	0	0	4	0	0	622	0	7
5	11	7	20	4	8	2	2	510	6
4	2	2	7	6	1	0	14	1	545

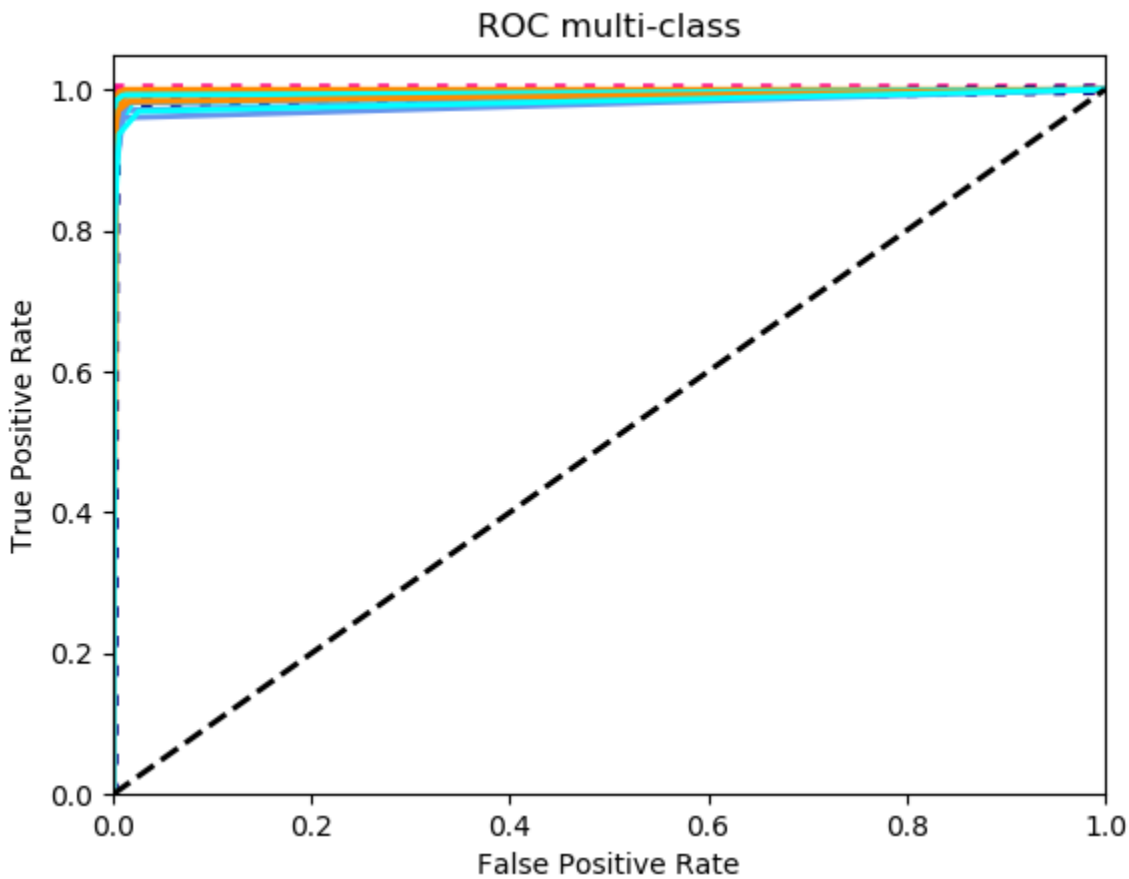


Figure 11- Mnist ROC, using knn classification

Hog

Execute MNIST classification using knn, hog feature

```
MNIST_classification.py –m knn –f hog
```

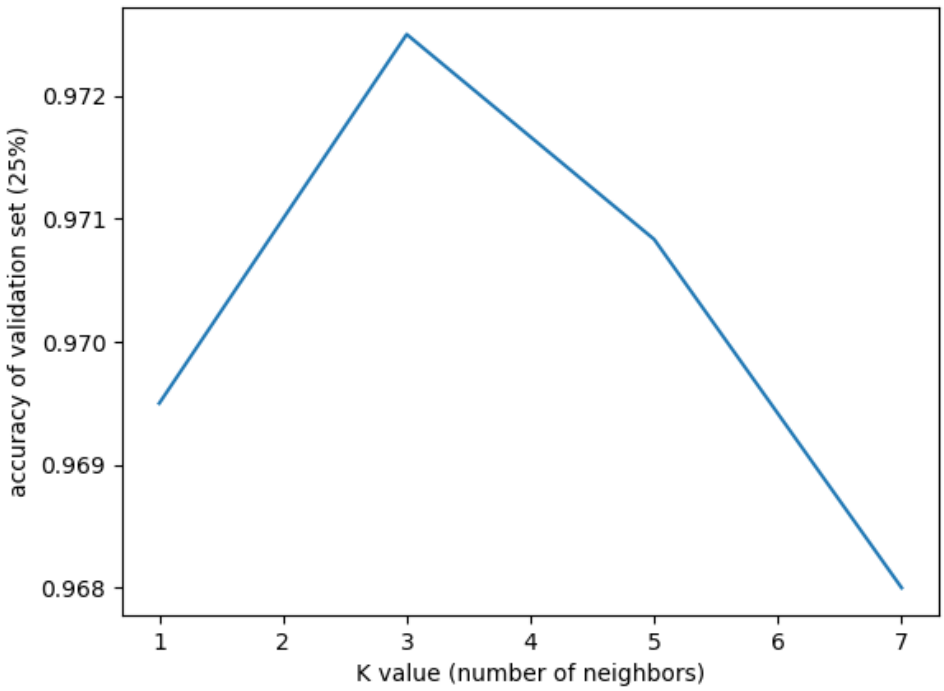


Figure 12- knn accuracy according to different k values, MNIST, hog features

k=3 achieved highest accuracy of **97.25%** on validation data.

Classification report:

label	precision	recall	f1-score	support
0	0.97	0.99	0.98	587
1	0.97	0.99	0.98	651
2	0.98	0.98	0.98	609
3	0.99	0.97	0.98	642
4	0.99	0.96	0.97	550
5	0.98	0.97	0.98	550
6	0.98	0.99	0.99	613
7	0.97	0.96	0.97	641
8	0.97	0.95	0.96	575
9	0.92	0.96	0.94	582
avg / total	0.96	0.96	0.96	6000

Confusion matrix:

583	1	0	0	0	0	3	0	0	1
1	645	2	0	2	0	0	1	0	0

2	4	594	0	0	0	0	5	4	0
1	0	4	622	0	1	0	4	7	3
0	2	2	0	528	0	1	1	0	16
3	2	0	4	1	533	5	0	1	1
1	1	0	0	1	2	608	0	0	0
1	4	1	0	0	0	0	615	1	19
10	7	1	1	0	3	1	0	546	6
1	3	0	1	4	4	1	6	1	561

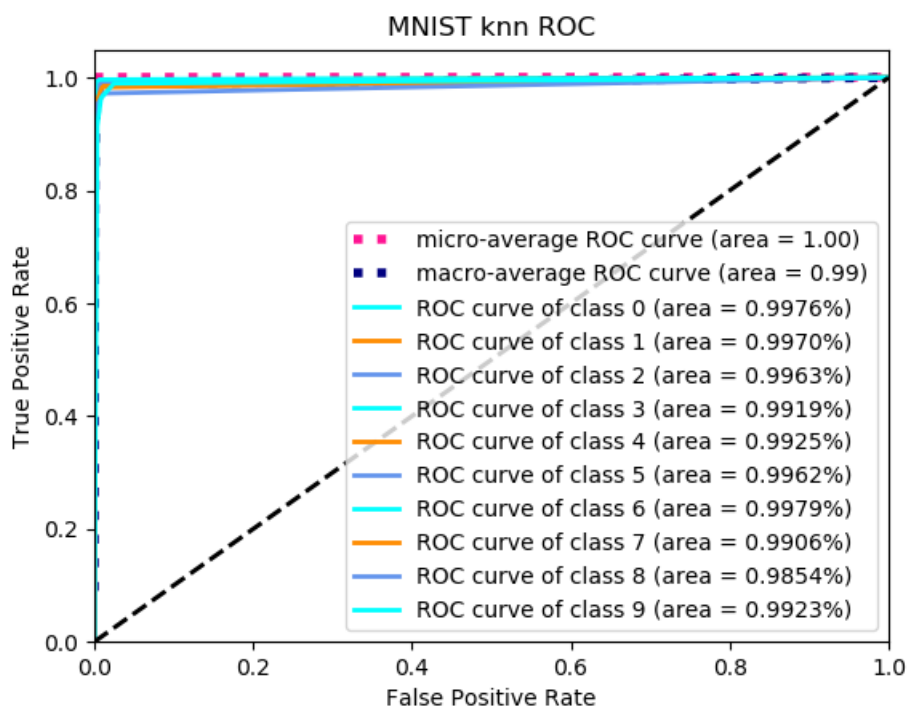


Figure 13- Mnist ROC, using knn classification, hog features

SVM

Baseline

As a baseline, I have utilized **raw pixel intensities** as inputs to SVM, testing different c values and three different kernel types (Linear, polynomial and rbf).

Execute MNIST classification using svm, baseline
`MNIST_classification.py –m svm –f baseline`

c=0.1 achieved highest accuracy of **96.07%** on validation data, using polynomial kernel with 3 degrees.

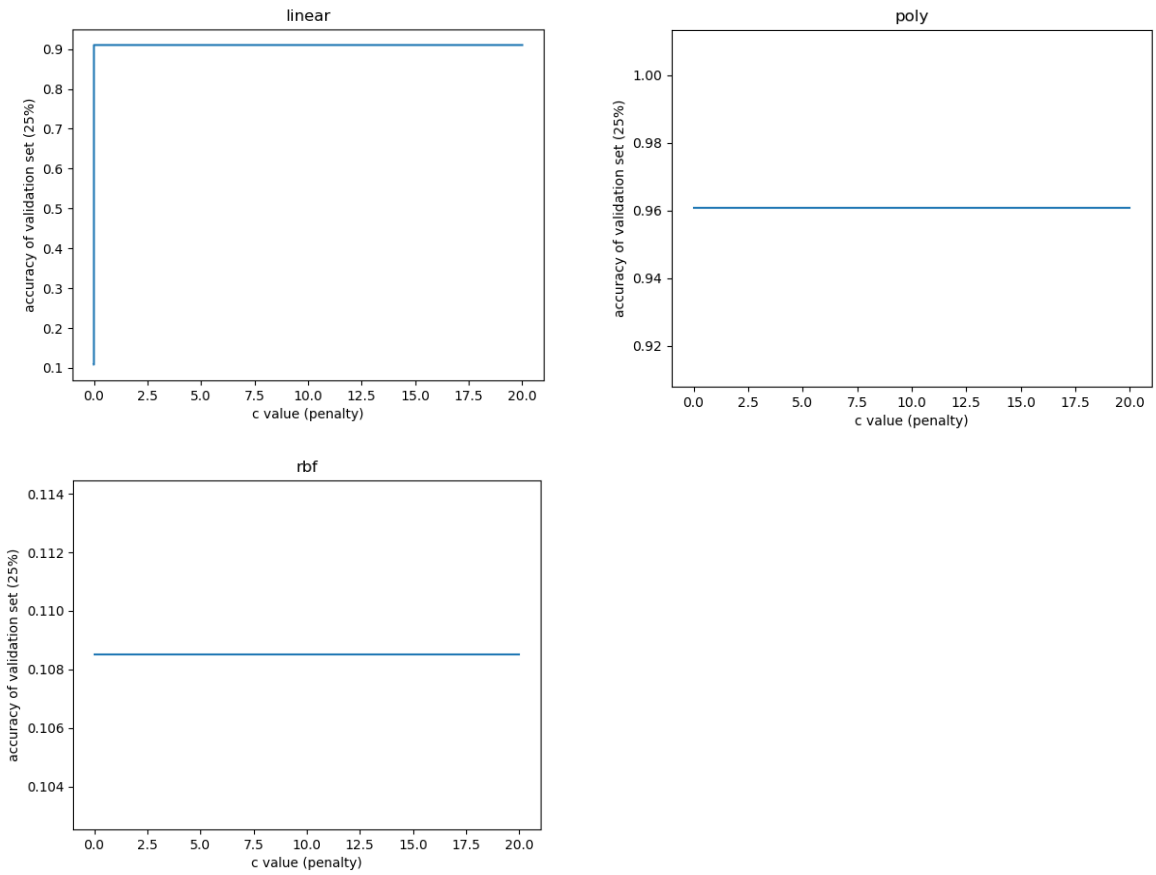


Figure 14- MNIST, Accuracy per C value for different kernels

Classification report:

label	precision	recall	f1-score	support
0	0.97	0.97	0.97	587
1	0.97	0.99	0.98	651
2	0.94	0.98	0.96	609
3	0.96	0.93	0.95	642
4	0.97	0.97	0.97	550
5	0.96	0.95	0.96	550
6	0.97	0.97	0.97	613
7	0.96	0.97	0.97	641
8	0.94	0.93	0.94	575
9	0.94	0.94	0.95	582
avg / total	0.96	0.94	0.95	6000

Confusion matrix:

570	0	3	0	1	4	4	0	4	1

0	643	1	1	1	0	0	2	3	0
1	2	595	1	1	1	2	3	3	0
2	3	18	597	0	7	2	5	6	2
1	0	3	0	536	0	2	1	1	6
2	3	2	9	0	523	3	0	6	2
6	1	1	0	2	3	596	0	4	0
0	5	2	0	1	1	0	622	1	9
5	8	7	7	1	5	4	2	533	3
3	1	0	4	8	1	1	11	4	549

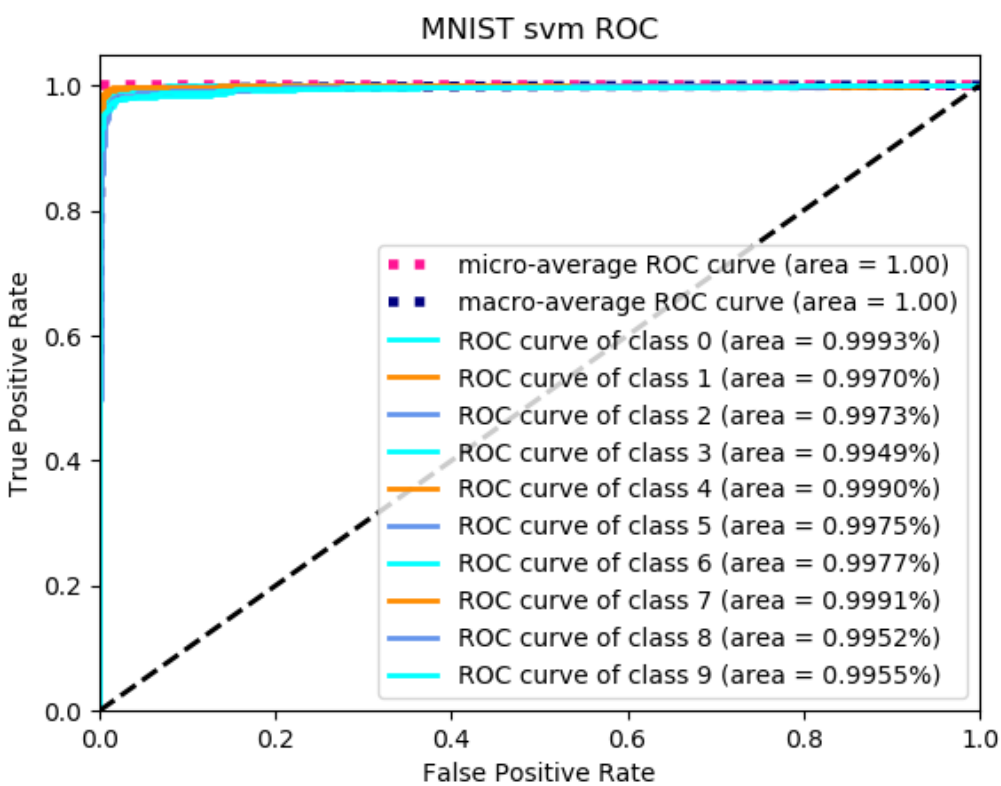
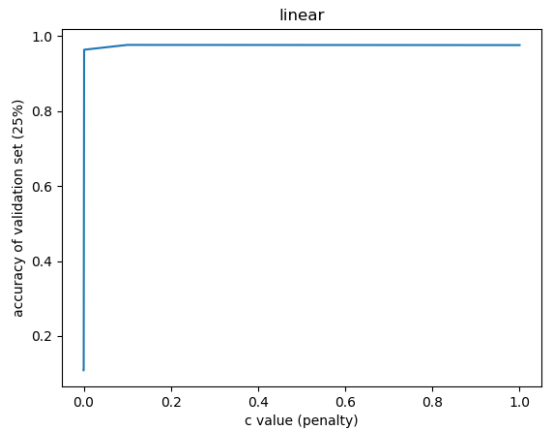


Figure 15- Mnist ROC, using svm classification

Hog

Execute MNIST classification using svm, hog
MNIST_classification.py –m svm –f hog

Using polynomial kernel with 3 degrees, with c=0.001, achieved highest accuracy of **98.57%**



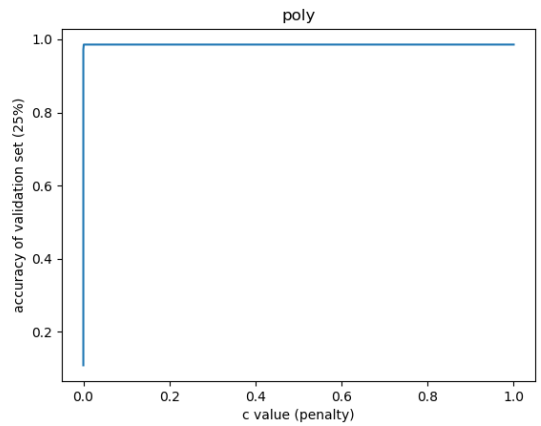


Figure 16- MNIST, Accuracy per C value for different kernels, hog features

Best accuracy of 98.57% achieved when using SVM classification, polynomial kernel, with 3 degrees.

Classification report:

label	precision	recall	f1-score	support
0	0.99	0.99	0.99	587
1	0.99	1	0.99	651
2	0.98	0.99	0.99	609
3	0.99	0.98	0.98	642
4	0.99	0.98	0.99	550
5	0.99	0.99	0.99	550
6	0.99	0.99	0.99	613
7	0.98	0.99	0.99	641
8	0.98	0.98	0.98	575
9	0.98	0.98	0.98	582
avg / total	0.99	0.99	0.99	6000

Confusion matrix:

581	0	1	0	0	0	3	0	1	1
1	648	0	0	1	0	0	1	0	0
0	2	604	0	0	0	0	1	2	0
0	0	4	630	0	0	0	2	**5**	1
0	2	2	0	541	0	1	0	0	4
0	0	0	3	0	543	2	0	1	1
0	0	1	0	1	**5**	605	0	1	0
0	1	1	0	1	0	0	632	2	4

2	1	4	3	1	1	0	1	562	0
1	1	0	2	3	1	0	5	1	568

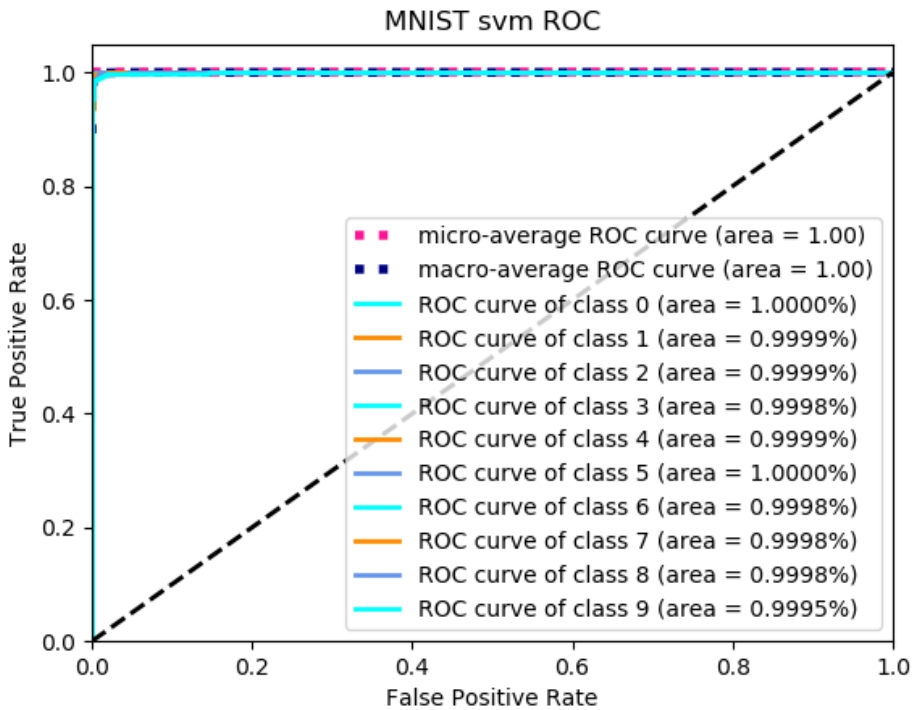


Figure 17- Mnist ROC, using svm classification, hog features

Summary & Conclusions

In this assignment, I've experience using KNN and SVM classification on Iris and MNIST data set.

During classification, different parameters has been tested: for KNN, different K values and for SVM, different C values and kernel types.

In addition, I've checked the use in HOG features for MNIST dataset.

Knn and SVM for **Iris classification** achieved same accuracy of 97.78%. Looking at ROC curve, it seems as KNN bring higher auc value for Iris classes.

For **MNIST classification**, best accuracy of 98.57% achieved when using SVM classification, polynomial kernel, with 3 degrees, on HOG features.

Looking at best classifier, at figure 18, we can see that the highest confusion is between 5 & 6 and 3 & 8. When written in handwrite, they are indeed visually similar.

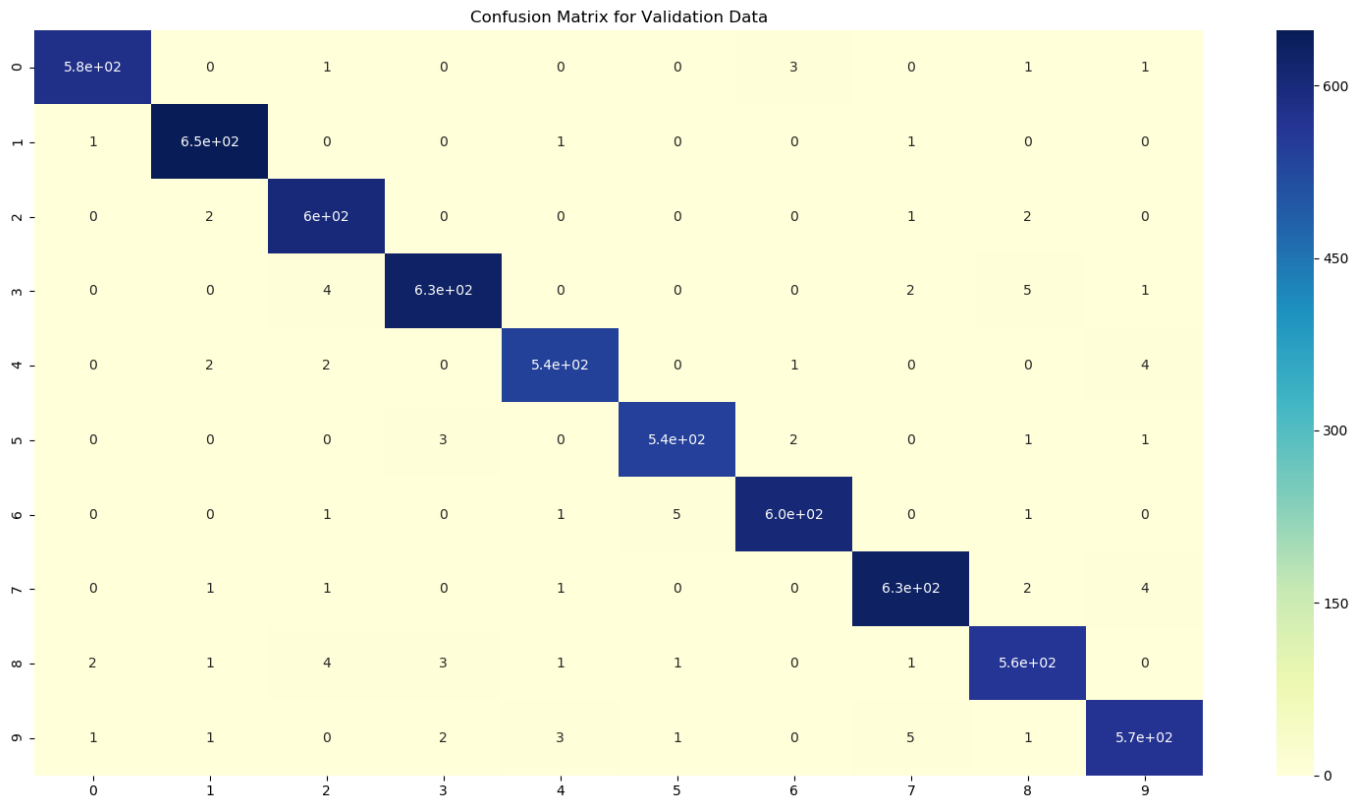


Figure 18- confusion matrix, svm polynomial classifier, hog features