hw3nguyen

October 17, 2020

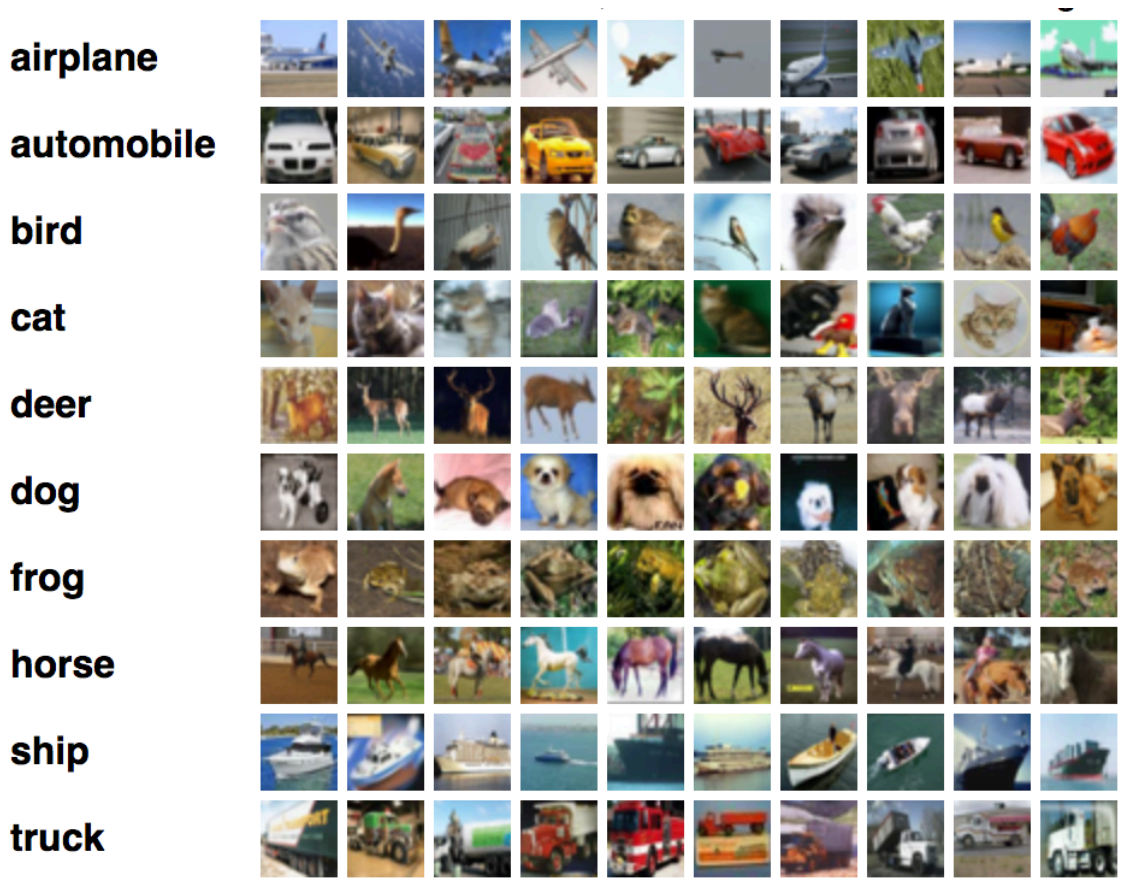
0.1 Assignment overview

In this assignment, you will perform image classification. You will experiment with support vector machines (using both linear and nonlinear kernels), as well as decision trees.

The data for this assignment comes from CIFAR-10, an image dataset created at the University of Toronto. Images are labeled with one of 10 classes (see examples below). Some of the classes are similar (e.g., "automobile" and "truck"), though in this dataset, they are defined to be mutually exclusive.

[1]:
```
from IPython.display import Image
Image('http://cmci.colorado.edu/classes/INFO-4604/data/hw3_example.png',␣
 ↪width=500)
```

[1]:



The images are tiny: 32x32 pixels. This makes it a difficult classification problem, because there is not a lot of detail in the images, but this is actually convenient for the purpose of this assignment, because the small size makes the data easier to work with computationally.

In general, image classification is a challenging machine learning problem, as well as a computationally intensive one. How to define and extract good features is an open challenge – you'll see a little of that here. The CIFAR-10 data contains 50,000 images for training and 10,000 for testing, but for this assignment, you will work with only a sample of 1,000 training images and 1,000 test images. Your classifiers won't be as accurate as classifiers trained on all 50,000 instances, but it will be a lot more manageable to work with a small sample.

0.1.1 What to hand in

You will submit the assignment on Canvas. Submit a single Jupyter notebook named hw3lastname.ipynb, where lastname is replaced with your last name. **Please also submit a .html or .pdf version of your assignment to Canvas**.

If you have any output that is not part of your notebook, you may submit that as a separate document, in a single PDF. For example, this assignment requires you to create plots. You could do it directly with python using matplotlib, but if you wanted to create them using other software, that's acceptable as long as you put all of the figures in a single document and you clearly label them with the corresponding deliverable number.

When writing code in this notebook, you are encouraged to create additional cells in whatever way makes the presentation more organized and easy to follow. You are allowed to import additional Python libraries.

0.1.2 Submission policies

- **Errors:** We should be able to run Kernel → Restart & Run All and run your submission without any errors. If your code does not run, we will not grade your assignment and you will not receive credit.
- **Collaboration:** You are allowed to work with a partner on this assignment. You are still expected to write up your own solution. Each individual must turn in their own submission, and list your collaborator after your name.
- **Late submissions:** We allow each student to use up to 5 late days over the semester. You have late days, not late hours. This means that if your submission is late by any amount of time past the deadline, then this will use up a late day. If it is late by any amount beyond 24 hours past the deadline, then this will use a second late, and so on. Once you have used up all late days, late assignments will be given at most 80% credit after one day and 60% credit after two days.

0.2 Getting started

Begin by downloading and loading the data, stored in CSV format. The first column in each CSV file is the class label (the y value), and the remaining columns are the feature values. The StandardScaler class is used to standardize the feature values using z-score normalization.

It may take a minute to run the cell below. In general, you'll notice that code in this assignment will take longer to execute than in HW2, even though the HW2 data had a similar number of instances and number of features. Why? The reason is that the image features are *dense*, meaning that all feature values need to be stored and used in the algorithm. In the tweets in HW2, the features were *sparse*, meaning that most of the feature values were simply 0, and therefore didn't need to be stored in memory and in many cases could be skipped during computation in the learning algorithms.

[2]:
```
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler

scalar = StandardScaler()

print('Loading training data...')

# 'http://cmci.colorado.edu/classes/INFO-4604/data/cifar_raw_train.csv
df = pd.read_csv('cifar_raw_train.csv', header=None)

rawY = df.iloc[0:, 0].values
rawX = df.iloc[0:, 1:].values
scalar.fit(rawX)
rawX = scalar.transform(rawX)

print('...done.')
print('Loading test data...')

# 'http://cmci.colorado.edu/classes/INFO-4604/data/cifar_raw_test.csv'
df = pd.read_csv('cifar_raw_test.csv', header=None)

rawY_test = df.iloc[0:, 0].values
rawX_test = df.iloc[0:, 1:].values
scalar.fit(rawX_test)
rawX_test = scalar.transform(rawX_test)

print('...done.')
```

```
Loading training data…
…done.
Loading test data…
…done.
```

[3]:
```
# investigate data
rawX_test.shape, 3072/3
```

[3]:
```
((1000, 3072), 1024.0)
```

0.2.1 What do the columns mean?

The columns in the above CSV files correspond to color values of each pixel of the image (after the first column, which is the label). A 32x32 image has 1,024 pixels. The first 1,024 features are the red channel values of each pixel (values from 0 to 255), the second 1,024 features are the green channel values, and the last 1,024 features are the blue channel values, for a total of 3,072 features.

We refer to this as the ``raw'' data, because it simply encodes the image itself. You will load a different version of the dataset with more abstract features later on in this notebook.

0.2.2 Establishing a baseline

Before experimenting with building classifiers, it can be good to come up with a ``baseline'' accuracy that you expect your classifiers to outperform. This is helpful for contextualizing how good your classifiers are doing.

sklearn provides a class called DummyClassifier which implements a few different naive baselines that don't learn from features. In the code below, we use the most_frequent strategy, which simply creates a classifier that always predicts the majority class in the training data.

The ten classes in this dataset are roughly evenly distributed, so the majority class baseline only has an accuracy of around 10%. If you get an accuracy of 10% or lower in your experiments later, you'll know that your classifier is not learning anything useful.

[4]:
```
from sklearn.dummy import DummyClassifier
from sklearn.metrics import accuracy_score

classifier = DummyClassifier(strategy='most_frequent')
classifier.fit(rawX, rawY)

print("Training accuracy: %0.6f" % accuracy_score(rawY, classifier.
 ↪predict(rawX)))
print("Testing accuracy: %0.6f" % accuracy_score(rawY_test, classifier.
 ↪predict(rawX_test)))
```

```
Training accuracy: 0.112000
Testing accuracy: 0.089000
```

0.3 Problem 1: Support vector machines

We will begin by trying to classify the data using support vector machines (SVMs). For this problem, you will use sklearn's SVC class, which implements a variety of SVM kernels. To handle multiple classes, SVC uses the ``all pairs'' algorithm (called ``one-vs-one'' in the sklearn documentation) for multiclass classification. (In contrast, the SGDClassifier you used in HW2 uses ``one-vs-rest'' by default, so you may get different results if you try to compare the two.)

You can define the kernel with the kernel keyword argument. Additional keyword arguments can specify various hyperparameters specific to the kernels. You should try four different kernels:

- linear
- poly with degree=2 (a quadratic kernel)
- poly with degree=3 (a 3rd-degree polynomial kernel)
- rbf (the radial basis function or Gaussian kernel)

You should also set the hyperparameter C with the keyword argument C, which controls the tradeoff between regularization (increasing the margin) and training performance (decreasing the loss). C is similar to alpha in SGDClassifier, but the inverse: increasing alpha increases regularization, while increasing C decreases regularization.

0.3.1 Classifying the raw data [2 points]

Let's start by using the data described above, which contains 3,072 features representing the RGB color values of the 1,024 pixels in the 32x32 images.

While the data was already downsampled to 1,000 training instances, it still takes a minute to train an SVM on this data.

Kernel	C	Test accuracy
Linear	0.0001	0.323
Poly-2	1.0	0.277
Poly-3	1.0	0.271
RBF	1.0	0.353

Deliverable 1.1: For each of the four kernels listed above, calculate the training and test accuracy when setting C to each of $[0.0001, 0.01, 1.0, 100.0]$. Fill out the table below, where for each kernel you give the value of C that gave the best test accuracy, as well as the test accuracy itself at that value of C. If you want, it's OK to write code to generate a table like this, instead of filling it out by hand. Just label the table clearly as deliverable 1.1.

[5]:
```
from sklearn.svm import SVC

# code for 1.1 here
```

```
# Note the random_state argument. Like in HW2, you should keep this argument
# in every classifier constructor so that you get consistent results.

C = [0.0001,0.01,1.0,100.0]
kernel_type = ["linear", "poly-2", "poly-3", "rbf"]

def printAccuracy(c_ = 1.0, ker ="linear", deg = 3):
    classifier = SVC(C = c_, kernel = ker, degree = deg, random_state=123)
    classifier.fit(rawX, rawY)
    print("Training accuracy, {}, {}, {}: %0.6f".format(ker, c_, deg) %␣
 ↪accuracy_score(rawY, classifier.predict(rawX)))
    print("Testing accuracy, {}, {}, {}: %0.6f".format(ker, c_, deg) %␣
 ↪accuracy_score(rawY_test, classifier.predict(rawX_test)))
    return classifier

for i in kernel_type:
    max_score = 0
    for j in C:
        deg = 3;
        ker = i
        if (i == "poly-2"):
            deg = 2
            ker = "poly"
        elif (i == "poly-3"):
            ker = 'poly'

        classifier = printAccuracy(j, ker, deg)
        curr_score = accuracy_score(rawY_test, classifier.predict(rawX_test))

        if (curr_score > max_score):
            max_score = curr_score
            max_c = j

    print("max", i, max_c, ':', max_score)
```

```
Training accuracy, linear, 0.0001, 3: 0.416000
Testing accuracy, linear, 0.0001, 3: 0.323000
Training accuracy, linear, 0.01, 3: 0.998000
Testing accuracy, linear, 0.01, 3: 0.301000
Training accuracy, linear, 1.0, 3: 1.000000
Testing accuracy, linear, 1.0, 3: 0.294000
Training accuracy, linear, 100.0, 3: 1.000000
Testing accuracy, linear, 100.0, 3: 0.294000
max linear 0.0001 : 0.323
Training accuracy, poly, 0.0001, 2: 0.112000
```

```
Testing accuracy, poly, 0.0001, 2: 0.089000
Training accuracy, poly, 0.01, 2: 0.112000
Testing accuracy, poly, 0.01, 2: 0.089000
Training accuracy, poly, 1.0, 2: 0.722000
Testing accuracy, poly, 1.0, 2: 0.277000
Training accuracy, poly, 100.0, 2: 1.000000
Testing accuracy, poly, 100.0, 2: 0.264000
max poly-2 1.0 : 0.277
Training accuracy, poly, 0.0001, 3: 0.112000
Testing accuracy, poly, 0.0001, 3: 0.089000
Training accuracy, poly, 0.01, 3: 0.120000
Testing accuracy, poly, 0.01, 3: 0.098000
Training accuracy, poly, 1.0, 3: 0.771000
Testing accuracy, poly, 1.0, 3: 0.271000
Training accuracy, poly, 100.0, 3: 0.999000
Testing accuracy, poly, 100.0, 3: 0.254000
max poly-3 1.0 : 0.271
Training accuracy, rbf, 0.0001, 3: 0.112000
Testing accuracy, rbf, 0.0001, 3: 0.089000
Training accuracy, rbf, 0.01, 3: 0.112000
Testing accuracy, rbf, 0.01, 3: 0.089000
Training accuracy, rbf, 1.0, 3: 0.781000
Testing accuracy, rbf, 1.0, 3: 0.353000
Training accuracy, rbf, 100.0, 3: 1.000000
Testing accuracy, rbf, 100.0, 3: 0.350000
max rbf 1.0 : 0.353
```

0.3.2 Using better features [4 points]

Using the color values of each individual pixel typically does not yield good image classification results. The exact color value of a specific pixel is not likely to be associated with a specific class in a way that generalizes well. Typically, more general and abstract features are extracted from images before using them in learning algorithms.

For the remainder of this assignment, we will use a different, preprocessed version of the dataset containing various image features rather than the original pixel representation. The data has two broad types of features:

- *DAISY* is an algorithm for extracting ``descriptors'' of different parts of an image, such as edges. These features were extracted using scikit-image.
- *Color histograms* provide counts of the number of times ranges of colors appear within an image (or section of an image), providing a summary of the color distribution rather than specifying the colors of individual pixels. For this data, color histogram features are extracted from the entire image as well as each quadrant of the image. These features were extracted using the calcHist function of the OpenCV computer vision library.

To load the new data, run the block of code below:

[6]:
```
scalar = StandardScaler()

print('Loading training data...')

df = pd.read_csv('cifar_features_train.csv', header=None)

Y = df.iloc[0:, 0].values
X = df.iloc[0:, 1:].values
scalar.fit(X)
X = scalar.transform(X)

print('...done.')
print('Loading test data...')

df = pd.read_csv('cifar_features_test.csv', header=None)

Y_test = df.iloc[0:, 0].values
X_test = df.iloc[0:, 1:].values
scalar.fit(X_test)
X_test = scalar.transform(X_test)

print('...done.')
```

```
Loading training data…
…done.
Loading test data…
…done.
```

[7]:
```
C = [0.0001,0.01,1.0,100.0]
kernel_type = ["linear", "poly-2", "poly-3", "rbf"]

def printAccuracy(c_ = 1.0, ker ="linear", deg = 3):
    classifier = SVC(C = c_, kernel = ker, degree = deg, random_state=123)
    classifier.fit(X, Y)
    return classifier

for i in kernel_type:
    max_score = 0
    for j in C:
        deg = 3;
        ker = i
        if (i == "poly-2"):
            deg = 2
            ker = "poly"
        elif (i == "poly-3"):
            ker = 'poly'
```

```
        classifier = printAccuracy(j, ker, deg)
        curr_score = accuracy_score(Y_test, classifier.predict(X_test))

        if (curr_score > max_score):
            max_score = curr_score
            max_c = j

    print(i, max_c, ':', max_score)
```

```
linear 0.01 : 0.415
poly-2 100.0 : 0.385
poly-3 100.0 : 0.28
rbf 100.0 : 0.477
```

Kernel	C	Test accuracy
Linear	0.01	0.415
Poly-2	100.0	0.385
Poly-3	100.0	0.28
RBF	100.0	0.477

Deliverable 1.2: Redo the experiments you did for 1.1 using the data with new features. Fill in the table below, once again with the best test accuracies and the C values that gave those accuracies:

Deliverable 1.3: Compare the results from 1.2 to the results from 1.1. Comment on three aspects: (i) Overall, which version (raw data in 1.1 or preprocessed data in 1.2) resulted in better performance? (ii) Was the relative performance of the four kernels (e.g., `rbf` outperforms `linear`) similar across the two versions? (iii) Were the optimal C values similar across the two versions?

i) The results from the preproccessed data resulted in higher testing accuracy
ii) The relative performance appears to be similar across versions. For both versions, the kernel accuracy from highest to lowest goes as follows: RBF, Linear, Poly-2, Poly-3.
iii) The optimal C values were different across both versions. The C for the linear kernel changed from 0.0001 to 0.01 in the preprocessed version. The C for the Poly-2, Poly-3, and RBF changed from 1.0 to 100.0 in the preproccessed version.

0.3.3 Understanding kernels [8 points]

For these experiments, let's focus on the rbf kernel (which you should have found outperformed the others) and the linear kernel. Set C to 0.001 for the linear kernel and 10.0 for the RBF kernel. Use the preprocessed data rather than the raw data (i.e., X and Y instead of rawX and rawY).

Deliverable 1.4: Train a linear and RBF kernel classifier using the first N training instances (where $N = 1000$ in the earlier problems) when N is each of $[20, 50, 100, 200, 400, 800, 1000]$ and calculate the test accuracy. Create a line plot where the x-axis is N and the y-axis is test accuracy. There should be two lines, one for each kernel. `[your solution should either be plotted below, or included in a separate PDF]`

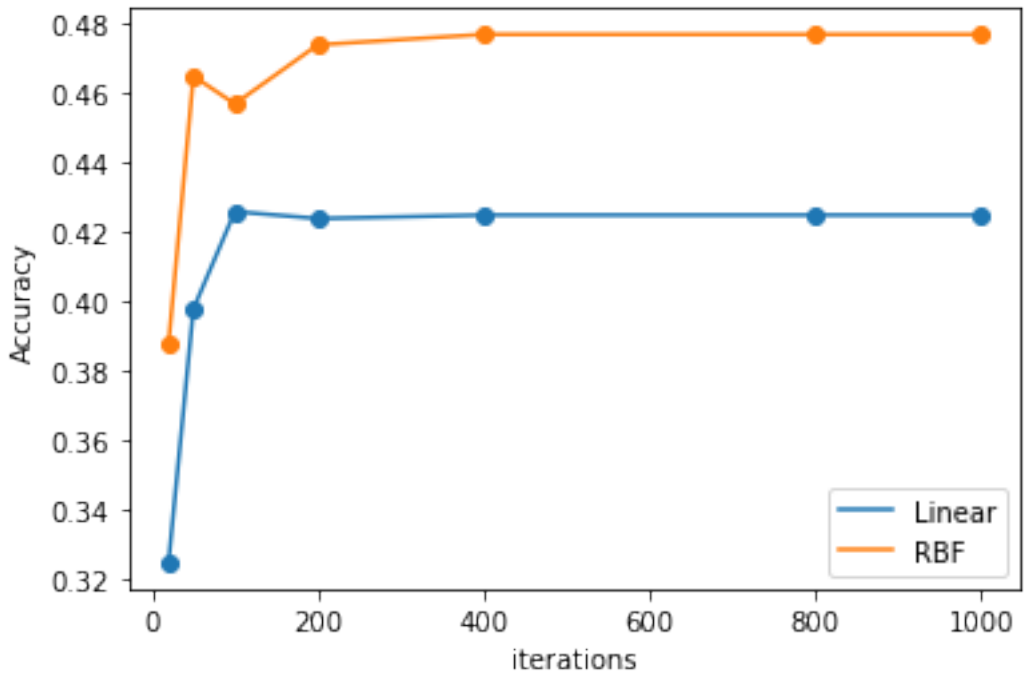
Deliverable 1.5: Nonlinear classifiers can learn more complex patterns than linear models, but they are at higher risk of overfitting because they have more parameters to learn. Keeping this in mind, what do you observe in your plot from 1.4? What might you conclude about the advantage of nonlinear classification versus linear classification based on the size of the training data? `At each iteration, the testing accuracy of the RBF kernel is higher than the accuracy for the linear model. For large datasets, non-linear classification is less prone to overfitting and therefore, is preferred to linear models beacuse of higher accuracy.`

[8]:
```
# code for 1.4 here
import matplotlib.pyplot as plt

n = [20,50,100,200,400,800,1000]

acc_linear = []
acc_rbf = []

for i in n:
    linear = SVC(kernel="linear", C=0.001, random_state=123, max_iter = i)
    linear.fit(X, Y)
    acc_linear.append(accuracy_score(Y_test, linear.predict(X_test)))

    rbf = SVC(kernel="rbf", C=10.0, random_state=123, gamma="scale", max_iter =␣
 ↪i)
    rbf.fit(X, Y)
    acc_rbf.append(accuracy_score(Y_test, rbf.predict(X_test)))

plt.plot(n, acc_linear, label = 'Linear')
plt.scatter(n, acc_linear)
plt.plot(n, acc_rbf, label = 'RBF')
plt.scatter(n, acc_rbf)
plt.ylabel("Accuracy")

plt.xlabel("iterations")
plt.legend()
```

```
/Users/nguyenkm13/opt/anaconda3/lib/python3.8/site-
packages/sklearn/svm/_base.py:246: ConvergenceWarning: Solver terminated early
(max_iter=20).  Consider pre-processing your data with StandardScaler or
MinMaxScaler.
  warnings.warn('Solver terminated early (max_iter=%i).'
```

```
/Users/nguyenkm13/opt/anaconda3/lib/python3.8/site-
packages/sklearn/svm/_base.py:246: ConvergenceWarning: Solver terminated early
(max_iter=20).  Consider pre-processing your data with StandardScaler or
MinMaxScaler.
  warnings.warn('Solver terminated early (max_iter=%i).'
/Users/nguyenkm13/opt/anaconda3/lib/python3.8/site-
packages/sklearn/svm/_base.py:246: ConvergenceWarning: Solver terminated early
(max_iter=50).  Consider pre-processing your data with StandardScaler or
MinMaxScaler.
  warnings.warn('Solver terminated early (max_iter=%i).'
/Users/nguyenkm13/opt/anaconda3/lib/python3.8/site-
packages/sklearn/svm/_base.py:246: ConvergenceWarning: Solver terminated early
(max_iter=50).  Consider pre-processing your data with StandardScaler or
MinMaxScaler.
  warnings.warn('Solver terminated early (max_iter=%i).'
/Users/nguyenkm13/opt/anaconda3/lib/python3.8/site-
packages/sklearn/svm/_base.py:246: ConvergenceWarning: Solver terminated early
(max_iter=100).  Consider pre-processing your data with StandardScaler or
MinMaxScaler.
  warnings.warn('Solver terminated early (max_iter=%i).'
/Users/nguyenkm13/opt/anaconda3/lib/python3.8/site-
packages/sklearn/svm/_base.py:246: ConvergenceWarning: Solver terminated early
(max_iter=100).  Consider pre-processing your data with StandardScaler or
MinMaxScaler.
  warnings.warn('Solver terminated early (max_iter=%i).'
/Users/nguyenkm13/opt/anaconda3/lib/python3.8/site-
packages/sklearn/svm/_base.py:246: ConvergenceWarning: Solver terminated early
(max_iter=200).  Consider pre-processing your data with StandardScaler or
MinMaxScaler.
  warnings.warn('Solver terminated early (max_iter=%i).'
/Users/nguyenkm13/opt/anaconda3/lib/python3.8/site-
packages/sklearn/svm/_base.py:246: ConvergenceWarning: Solver terminated early
(max_iter=200).  Consider pre-processing your data with StandardScaler or
MinMaxScaler.
  warnings.warn('Solver terminated early (max_iter=%i).'
/Users/nguyenkm13/opt/anaconda3/lib/python3.8/site-
packages/sklearn/svm/_base.py:246: ConvergenceWarning: Solver terminated early
(max_iter=400).  Consider pre-processing your data with StandardScaler or
MinMaxScaler.
  warnings.warn('Solver terminated early (max_iter=%i).'
/Users/nguyenkm13/opt/anaconda3/lib/python3.8/site-
packages/sklearn/svm/_base.py:246: ConvergenceWarning: Solver terminated early
(max_iter=400).  Consider pre-processing your data with StandardScaler or
MinMaxScaler.
  warnings.warn('Solver terminated early (max_iter=%i).'
```

[8]: <matplotlib.legend.Legend at 0x7fb99f28b820>



In class, we learned that the RBF kernel has a hyperparameter γ which can affect overfitting. You can set the value of γ with the keyword argument gamma.

For these experiments with gamma, it is preferred that you use all 1000 training instances.

gamma	Training accuracy	Test accuracy
1e-2	1.000000	0.47700
1e-3	1.000000	0.47700
1e-4	1.000000	0.47700
1e-5	1.000000	0.47700
1e-6	1.000000	0.47700
1e-7	1.000000	0.47700

Deliverable 1.6: Train an RBF SVM (with $C = 10.0$) where gamma is each of: [1e-2, 1e-3, 1e-4, 1e-5, 1e-6, 1e-7], where 1eM is scientific notation meaning 1×10^M. For each value of gamma, compute both the training accuracy and test accuracy. Record the values in the table below:

[9]:
```
# code for 1.6 here
gam =  [1e-2, 1e-3, 1e-4, 1e-5, 1e-6, 1e-7]
for i in gam:
    rbf = SVC(kernel="rbf", C=10.0, random_state=123, gamma= i)
    rbf.fit(X, Y)
```

```
    print("Training accuracy, gamma = {}: %0.6f".format(i) % accuracy_score(Y,␣
↪classifier.predict(X)))
    print("Testing accuracy, gamma = {}: %0.6f".format(i) %␣
↪accuracy_score(Y_test, classifier.predict(X_test)))
```

```
Training accuracy, gamma = 0.01: 1.000000
Testing accuracy, gamma = 0.01: 0.477000
Training accuracy, gamma = 0.001: 1.000000
Testing accuracy, gamma = 0.001: 0.477000
Training accuracy, gamma = 0.0001: 1.000000
Testing accuracy, gamma = 0.0001: 0.477000
Training accuracy, gamma = 1e-05: 1.000000
Testing accuracy, gamma = 1e-05: 0.477000
Training accuracy, gamma = 1e-06: 1.000000
Testing accuracy, gamma = 1e-06: 0.477000
Training accuracy, gamma = 1e-07: 1.000000
Testing accuracy, gamma = 1e-07: 0.477000
```

Deliverable 1.7: Describe what you observe in the table for 1.6, describing the effect of gamma in terms of the bias/variance tradeoff. The accuracy of RBF is unaffected by the different gamma.

0.3.4 Understanding features [5604: 4 points; 4604: +2 EC]

As stated earlier, there are two types of features in the preprocessed data: DAISY features and color histogram features.

For this last SVM problem, create two additional copies of X and X_test: one that only contains the DAISY features (the first 1664 columns of X), and one that only contains the histogram features (the last 320 columns of X). Give the copies new variable names, because the rest of this notebook will continue to use the previous definitions of X and X_test.

Experiment with these two different feature sets. For these experiments, use a linear SVM with $C = 0.001$ and an RBF SVM with $C = 10.0$ and a default setting of gamma.

Deliverable 1.8: Compute the test accuracy when using only the DAISY features (for both linear and RBF kernels), and compute the test accuracy when using only histogram features (for both kernels). Describe what you observe. How do the two feature sets compare to each other? How do the two feature sets compare to using both types of features combined? Both the training and testing accuracy was higher for the Daisy data than the histogram data across both models. The testing accuracy of the combined data was higher for both the linear and rbf models when compared to the daisy and histogram data.

[10]:
```
# code for 1.8 here
X_Daisy = X[0:, 0:1664]
```

```
X_Daisy_test = X_test[0:, 0:1664]
X_histogram = X[0:, 1984 - 320:]
X_histogram_test = X_test[0:, 1984 - 320:]

linear = SVC(kernel="linear", C=0.001, random_state=123)
rbf = SVC(kernel="rbf", C=10.0, random_state=123)


print("linear")
linear.fit(X_Daisy, Y)
print("train acc daisy: ", accuracy_score(Y, linear.predict(X_Daisy)))
print("test acc daisy: ", accuracy_score(Y_test, linear.predict(X_Daisy_test)))
linear.fit(X_histogram, Y)
print("train acc histogram: ", accuracy_score(Y, linear.predict(X_histogram)))
print("test acc histogram: ", accuracy_score(Y_test, linear.
 ↪predict(X_histogram_test)))

print("rbf")
rbf.fit(X_Daisy, Y)
print("train acc daisy: ", accuracy_score(Y, rbf.predict(X_Daisy)))
print("test acc daisy: ", accuracy_score(Y_test, rbf.predict(X_Daisy_test)))
rbf.fit(X_histogram, Y)
print("train acc histogram: ", accuracy_score(Y, rbf.predict(X_histogram)))
print("test acc histogram: ", accuracy_score(Y_test, rbf.
 ↪predict(X_histogram_test)))
```

```
linear
train acc daisy:  0.665
test acc daisy:  0.403
train acc histogram:  0.357
test acc histogram:  0.251
rbf
train acc daisy:  1.0
test acc daisy:  0.47
train acc histogram:  0.879
test acc histogram:  0.302
```

0.4 Problem 2: Decision Trees [6 points]

Let's now experiment with decision tree classification on the preprocessed data using sklearn's DecisionTreeClassifier class.

There are a variety of hyperparameters that can be adjusted to control the bias/variance tradeoff. We will experiment with two:

- max_depth is the maximum depth of the decision tree, as we learned about in class.

- `min_samples_leaf` is the minimum number of training instances that must be remaining at a leaf node to make a final prediction. If the algorithm tries to split a node and fewer than this minimum instances would end up in the leaf, then the algorithm will stop expanding down that path.

As with SVMs, a bit of code is in the cell below to help you get started.

Deliverable 2.1: Experiment with setting `max_depth` to each of $[3, 8, 14, 100]$ and setting `min_samples_leaf` to each of $[1, 5, 10, 15, 50]$. For each combination of these two hyperparameters, record the training and test accuracies in the large table below. It will take a few minutes to do all of these combinations. See output below.

max_depth	min_samples_leaf	Training accuracy	Test accuracy
3	1		
3	5		
3	10		
3	15		
3	50		
8	1		
8	5		
8	10		
8	15		
8	50		
14	1		
14	5		
14	10		
14	15		
14	50		
100	1		
100	5		
100	10		
100	15		
100	50		

Deliverable 2.2: Based on your observations, as well as what you know about these two hyperparameters, describe the effects of `max_depth` and `min_samples_leaf` on the classifier performance and bias/variance tradeoff, including anything you observe about particular combinations of the two hyperparameters. For a fixed max_depth, as min_samples_leaf increased, the training accuracy decreased. For a fixed min_samples_leaf, the training accuracy increased from depths 3 to 14 but did not change from 14 to 100. The testing accuracy, however, saw the highest accuracy at min_samples_leaf of 10 across all max_depths.

It seems decision trees do not perform especially well on this dataset. In class, we learned that *random forests* can combine a number of different decision trees to improve performance. This is easy to run in sklearn with the RandomForestClassifier class. It uses most of the arguments that

DecisionTreeClassifier supports. The argument n_estimators specifies the number of decision trees to train.

n_estimators	Test accuracy
10	0.33
50	0.374
100	0.373
300	0.372

Deliverable 2.3: Train a random forest using the best settings of max_depth and min_samples_leaf from 2.1 (best setting by test accuracy) and record the test accuracies in the table below. Set n_estimators to each of $[10, 50, 100, 300]$. You should see a substantial increase in accuracy over using an individual decision tree.

[11]:
```
from sklearn.tree import DecisionTreeClassifier

# code for 2.1 here

depth = [3,8,14,100]
leaf = [1,5,10,15,50]
print("depth , leaf : Train | Test")

print("---------------------------------")
best_acc = 0
for i in depth:
    for j in leaf:
        clf = DecisionTreeClassifier(max_depth=i, min_samples_leaf=j,␣
↪random_state=123)
        clf.fit(X, Y)
        curr_acc = accuracy_score(Y_test, clf.predict(X_test))
        if (curr_acc > best_acc):
            best_acc = curr_acc
            best_depth = i
            best_leaf = j
        print("   {}   ,   {}   :".format(i, j), accuracy_score(Y, clf.
↪predict(X)),'|', accuracy_score(Y_test, clf.predict(X_test)))

print("\nbest_acc:", best_acc)
print("best_depth:", best_depth)
print("best_leaf:", best_leaf)
```

```
depth , leaf : Train | Test
---------------------------------
   3   ,   1   : 0.305 | 0.248
   3   ,   5   : 0.305 | 0.248
   3   ,   10   : 0.304 | 0.248
   3   ,   15   : 0.301 | 0.25
```

```
   3   ,   50   : 0.301 | 0.241
   8   ,   1   : 0.645 | 0.263
   8   ,   5   : 0.589 | 0.261
   8   ,   10   : 0.553 | 0.282
   8   ,   15   : 0.532 | 0.278
   8   ,   50   : 0.397 | 0.283
   14   ,   1   : 0.968 | 0.258
   14   ,   5   : 0.782 | 0.25
   14   ,   10   : 0.63 | 0.282
   14   ,   15   : 0.552 | 0.273
   14   ,   50   : 0.397 | 0.283
   100   ,   1   : 1.0 | 0.242
   100   ,   5   : 0.784 | 0.254
   100   ,   10   : 0.63 | 0.282
   100   ,   15   : 0.552 | 0.273
   100   ,   50   : 0.397 | 0.283

best_acc: 0.283
best_depth: 8
best_leaf: 50
```

[12]:
```
from sklearn.ensemble import RandomForestClassifier

# code for 2.3 here
estimators = [10,50,100,300]
print("n_estimators | Test Accuracy")
print("----------------------------")

for i in estimators:
    clf = RandomForestClassifier(n_estimators=i, max_depth=8,␣
 ↪min_samples_leaf=50, random_state=123)
    clf.fit(X, Y)
    print("      {}      |      {}      ".format(i, accuracy_score(Y_test, clf.
 ↪predict(X_test))))
```

```
n_estimators | Test Accuracy
----------------------------
      10      |      0.33
      50      |      0.374
      100      |      0.373
      300      |      0.372
```

[ ]:
```

```