1 Data Augmentation

Lab Objective: *Explore different methods of extending data sets to create more robust classifiers.*

Data Augmentation

It is not hard to find amusing examples of deep neural networks or other machine learning systems that are brittle and respond poorly to inputs that are only slightly different than the data that the systems were trained on. One way to address brittleness in machine learning systems is to train them on a much wider range of examples. Adult humans, for example, have seen many images of stop signs in a wide variety of settings. If a machine learning system had seen as many different images of stop signs in as many different settings, it would be much more robust. But all this requires large amounts of data, and good labeled data is hard to come by.

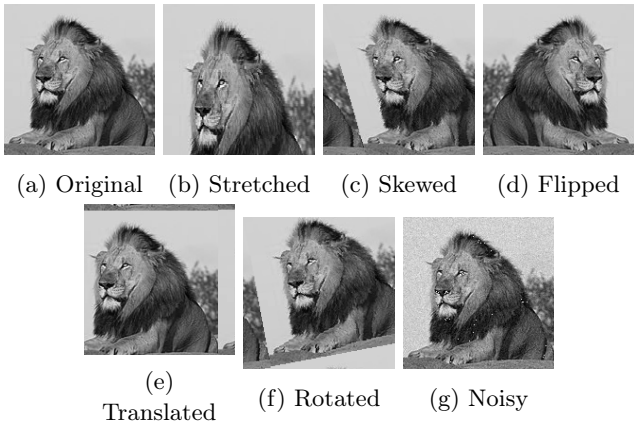
One common approach for generating new data is *data augmentation*, that is, generating new data from old data by applying various transformations that we know should not change the label. For example, an image of a stop sign can be slightly translated, rotated, skewed, or cropped and still be a legitimate image of a stop sign. It may even be blurred or partially obscured, so a classifier for identifying a stop sign must be robust in the face of this sort of interference. Images that don't contain text can often be flipped horizontally, and depending on the image, can also sometimes be flipped vertically. We can use these techniques to generate a larger data set and create more robust classifiers.

As shown in Figure 1 below, these transformations still accurately depict a lion while providing slight modifications to the original image. Image transformations are comprised of three steps. First, create a $2 \times (d1 * d2)$ coordinate representation of the image ($d1$ and $d2$ are the dimensions of the image). For example, if the image is 3 pixels by 3 pixels, the coordinate representation would be

$$C = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 2 & 2 & 2 \\ 0 & 1 & 2 & 0 & 1 & 2 & 0 & 1 & 2 \end{bmatrix}$$

where each column represents the coordinate point of a pixel. Next, perform a linear transformation on the coordinate matrix. Note that some transformations will return float values; however, they need to be integers because the matrix represents coordinate points so you will need to round the values. Finally, use the transformed coordinates to create a new image. (The function `np.take()` may be useful for this.)

Visualizing the code below would give you the second image in Figure 1.



(a) Original (b) Stretched (c) Skewed (d) Flipped

(e) Translated (f) Rotated (g) Noisy

Figure 1.1: Different image transformations

```
>>> import numpy as np
>>> from imageio import imread
>>> import matplotlib.pyplot as plt

>>> lion = imread('sq_lion.png')
>>> d1, d2 = lion.shape
>>> # Get the coordinate points for each pixel in the image
>>> coords = np.mgrid[0:d1, 0:d2].reshape((2,d1*d2))
>>> # Create a linear transformation matrix. (This one will stretch the matrix)
>>> stretch_matrix = np.array([[1, 0], [0, .7]])
>>> # apply the linear transformation to the coordinate matrix
>>> new_coords = stretch_matrix@coords
>>> # some transformations will return entries as floats, but we need them to ↩
    be integers because they are coordinates
>>> new_coords = new_coords.astype(int)
>>> # the next two steps apply the transformation to the image
>>> x, y = new_coords.reshape((2, d1, d2), order='F')
>>> stretched_lion = np.take(lion, x+d1*y, mode='wrap').reshape((d1, d2))
```

When augmenting a data set, it is also important to consider what types of augmentation would provide useful samples. For example, if training a dataset to recognize lowercase letters, flipping an image upside down may not be a useful transformation (consider the letters b and p). It is important to ensure that the transformed data is still an reasonable example of the object it is classified as.

Problem 1. Code from scratch the following simple black-and-white image augmenters that take as inputs the data X (a $d_1 \times d_2$ array that contains an images) and parameters controlling the transformation. It should return the transformed data $f(X)$. Note that each image should receive its own random treatment; for example, if the images are being translated, then each image should be translated by a different (randomly drawn) amount. Your functions should have the the following names and perform the corresponding transform:

1. `translate(X,A,B)`, with parameters A, B. Returns an image translated by a random amount (a, b), where $a \sim unif(-A, A)$, and $b \sim unif(-B, B)$. The resulting image should be cropped to be of size $d_1 \times d_2$. Note that this translation will leave a border on two sides of the image. Fill the empty border with the parts that were cropped off the opposite sides.

2. `rotate(X,T)`, with parameter Θ. Returns each image rotated by a random amount $\theta \sim unif(-\Theta, \Theta)$. The resulting image should be cropped to be the same size as the original, and any blank parts should be filled with one of the parts cropped off the other side. HINT: The rotation matrix is:

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}.$$

3. `skew(X,A)`, with parameter A. Returns each image with the linear transformation $\left[\begin{smallmatrix} 1 & a \\ 0 & 1 \end{smallmatrix}\right]$ applied, where $a \sim unif(0, A)$. Crop parts that go outside the image boundaries and fill missing areas with the appropriate cropped piece.

4. `flip_horizontal(X)`. Returns a horizontally-flipped version of each image.

5. `gauss_noise(X,s)`, with parameter σ^2. For each image draw $d_1 \times d_2$ random noise values from Normal$(0, \sigma^2)$ and add those to the original image.

Show that each transformation works by displaying a transformed version of lion.png.

Problem 2. Create a function called `image_augment` that will augment your data set using each of the transformations created in problem 1. This function should accept the parameters X (the images), Y (labels), and a list of the parameters for each transformation. The function should return an augmented data set with 6 times the number of images N and an array containing the appropriate label for each image.

Take the sklearn digits dataset, make an 80-20 train-test split, and then apply each of your transformations to the entire training set using `image_augment`. You must decide good values of each of the parameters to use. This should give you a larger (augmented) training set with roughly 8,600 training points. Fit a random forest to the augmented training set and to the original training set and score it using the test set. Return the average score for both classifiers (be sure to label the scores).

Your augmented score should be better than your original score.

Audio Augmentation

For audio data, adding various forms of noise is still a reasonable augmentation choice, but many of the other augmentation methods used for images aren't really suitable. Some useful methods include dropping data at certain time steps, blocking certain frequencies, and changing the pitch or speed.

When adding noise, it may be useful to consider the types of noise most likely to be encountered when the method is in use. For example, if the method will be used to identify voice commands in an outdoor environment, then adding in typical outdoor noises would probably be more useful than

adding white noise, which may not occur much in everyday life. Be thoughtful in choosing how to augment your data, as some types of manipulations may change or obscure the data to the point where it is no longer recognizable.

Audio Packages

There are many different python packages that provide different analysis and audio manipulation tools. Some common packages include `pyAudioAnalysis`, `PYO`, and `ffmpeg-python`. For the purposes of this lab, we will be using `LibROSA`. `LibROSA` is a python package that allows users to read in, write, analyze, and alter .wav files. It can be used to augment an audio dataset and extract features that can be used to distinguish between different sounds or types of music. (`LibROSA` is not included in Anaconda, so you may need to install it with `pip install librosa`.)

```
>>> import matplotlib.pyplot as plt
>>> import librosa
>>> import librosa.display  #this needs to be imported separately,
                            #but is included in the librosa package
>>> import numpy as np
# load the audio time series and sampling rate
>>> chopin, sample_rate = librosa.load('chopin.wav', sr = 22050)
>>> plt.figure(figsize = (15,5))
>>> librosa.display.waveplot(chopin,sample_rate)            #generate plot
>>> plt.show()
```

The `LibROSA` library heavily relies on NumPy arrays. If you already have a NumPy array and it's sample rate, you can skip the `librosa.load()`.

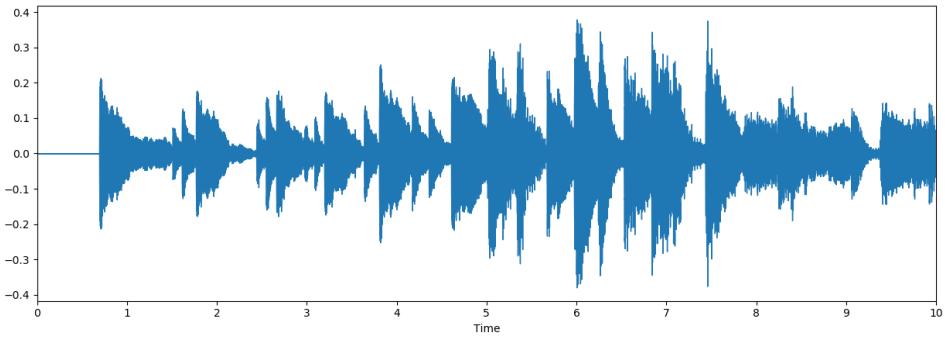


Figure 1.2: Visualization of an audio file

When you load in a `.wav` file, `LibROSA` returns the audio as an `ndarray` and a sample rate.

Depending on the type of audio you are analyzing, different functions may provide better distinguishing characteristics than others. It is important to take these characteristics into account when augmenting data. One possible feature that could be used for classifying music is the "predominant local pulse estimation" or PLP, as shown below. (PLP essentially takes the pulse of the music, just like you can take your own pulse in your wrist.)

```
>>> pulse = librosa.beat.plp(chopin)
>>> plt.figure(figsize = (15,5))
>>> plt.plot(np.linspace(0,10,len(pulse)),pulse)
>>> plt.show()
```

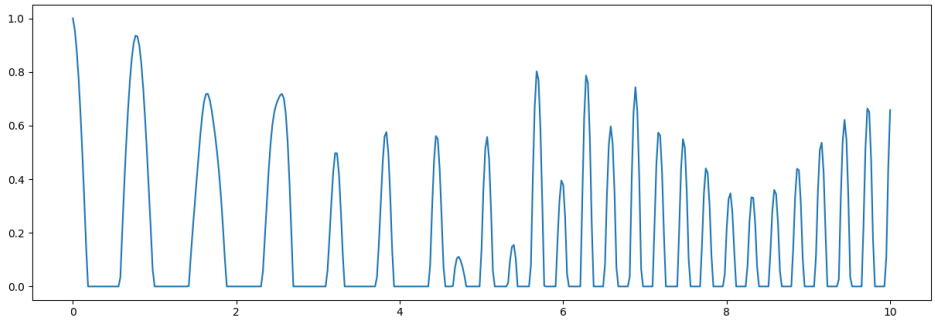


Figure 1.3: Predominant local pulse estimation of audio from figure 1.2

The `LibROSA` package contains several different functions that can be used to manipulate audio data, several of which are described in the table below.

Function	Returns
`time_stretch()`	slows or speeds up audio series by a fixed rate
`pitch_shift()`	shifts the pitch by n_steps semitones
`harmonic()`	extracts the harmonic elements from an audio time-series
`percusive()`	extracts percussive elements from an audio time-series
`split()`	splits an interval into non-silent intervals
`remix()`	re-orders time intervals

Table 1.1: These descriptions were taken directly from `librosa.org/librosa/effects.html`

Problem 3. The file `music.npy` contains the audio time series data of 10 second clips from 150 different songs, with `styles.npy` describing the associated style of ballroom dance. The styles included are Chacha, Foxtrot, Jive, Samba, Rumba, and Waltz. Use `train_test_split` from `sklearn.model_selection` with `test_size=.5` to create train and test sets.

Create two training sets by augmenting this original training set. Each new augmented training set will include the original data and the augmented data. For the first, add ambient noise from the file `restaurant-ambience.wav`. For the second, use `time_stretch`.

HINT: Since the ambient noise clip is much longer than the other music clips, you will have to select a sample of the ambient noise to add to the other clips. It may also benefit you to randomize which ambient noise sample you add to each clip, you can do this by choosing a random index to start from, and sampling starting at that index.

Problem 4. Do the following steps 5 times:

- Use the original data set and the augmented data sets to fit three RandomForestClassifiers, one only on the original data, one on the original data and the data with ambient noise added, and one on the original data and the time stretched data.
- Score each classifier.

Print the mean score for each of the classifiers and print the standard deviation for the scores.

HINT: Use the PLP as a feature you use to fit and classify. This example may be helpful for printing your results nicely.

```
print('\t\t Mean \t STD')
print('Original','\t',np.round(orig.mean(),3),'\t',np.round(orig.std(),3))
print('Ambient Noise','\t',np.round(amb.mean(),3),'\t',np.round(amb.std()↩
    ,3))
print('Time Stretch:','\t',np.round(time.mean(),3),'\t', np.round(time.std↩
    (),3))
```

Synthetic Minority Oversampling

Another situation where generating data can be helpful is in a classification problem where one class is rare compared to the others. For example the problem of identifying glioblastoma (a rare malignant brain tumor) is difficult in part because this cancer only occurs in 3 out of every 100,000 people. A classifier that predicts "no cancer" in every case performs very well (99.997% accurate).

If the training set has 100,000 total cases, only three of which are positive, then undersampling (taking the same number of negatives as positives) gives a dataset with only six total instances, and this is not enough to make a good classifier. Naïve oversampling (repeatedly drawing, with replacement, from the three positive cases to get the same number of positives as negatives) works better than undersampling the negatives, but does not perform very well, because the oversampled dataset just has (roughly) 33,332 repeated instances of each of the three positve instances, and the resulting classifier is likely to be overfit on those three instances.

In the special case where the features are all continuous, we can partially address this class-imbalance problem by synthetically generating new positive instances from the minority class samples (in this case the three positive cases). The *synthetic minority oversampling technique (SMOTE)*[1] works by randomly choosing points along the line segments connecting this point to each (or some) of its k nearest minority neighbors.

SMOTE tends to work better on low-dimensional data than on high-dimensional data. For example if the minority class training examples are images (one dimension per pixel, so, high dimensional) of the subject (say possible tumor cells) that are not centered and not uniformized to be of similar size, then many points along the line connecting two of these images could look nothing like the two endpoints. In such cases SMOTE is not usually very helpful.

[1] See `https://jair.org/index.php/jair/article/view/10302`

The Algorithm

The purpose of this section is to provide a high-level understanding of how Synthetic Minority Oversampling works and will heavily reference the SMOTE paper mentioned earlier.

The goal in creating synthetic observations is to increase the accuracy of the classifier. This means that the synthetic data generated needs to be similar to the minority class data, while creating moderate variation. To do this, select a member of the minority class and find its k nearest neighbors. Randomly select one. Now, for each feature select a random point on the line between the points.

We will demonstrate this using data with two features, represented by x and y coordinates. Consider the points $(0, 0)$, $(1, 3)$, $(2, 1)$, and $(3, 2)$, as shown in figure 1.4.

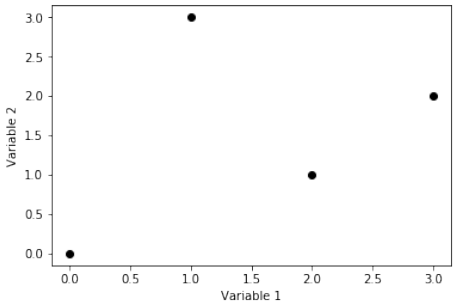


Figure 1.4: Data before SMOTE

To keep things simple, we will use $k = 1$. The nearest neighbor for $(0, 0)$ is $(2, 1)$. Choose a random point between the x values (shown in red) and a random point between the y values (shown in blue). For data with n features, a random point between the two feature values would be chosen for *each* feature. The intersection of these lines gives us the coordinate for the synthetic point (purple).

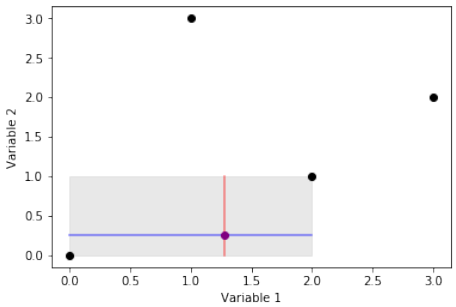
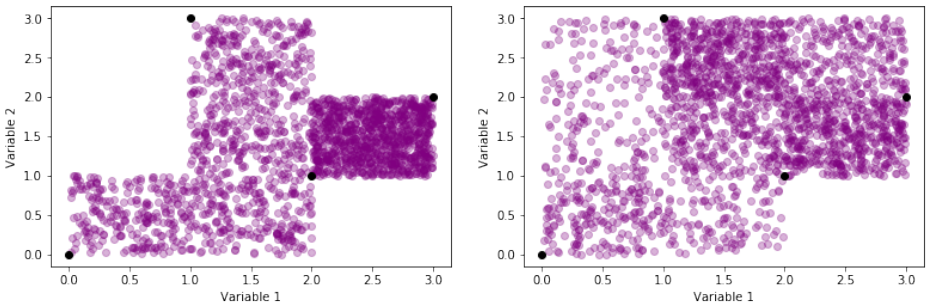


Figure 1.5: SMOTE process

Running this algorithm 500 times per original point, with $k = 1$, returns a graph like figure 1.6a. Running the algorithm 500 times per original point and increasing k to 2, returns the graph like figure 1.6b.



(a) $k = 1$ (b) $k = 2$

Figure 1.6: After SMOTE

Problem 5. Write a function that uses the synthetic minority oversampling technique to augment an imbalanced data set. Your function should:

- Take arguments: `X` a matrix of minority class samples, `N` the number of samples to generate per original point, and `k` the number of nearest neighbors.
- For each original point in the sample, randomly pick one of the `k` nearest neighbors and randomly generate a new point that lies between the two original values. You may use `sklearn.neighbors.KDTree` to find the k nearest neighbors.
- Return an array containing the synthetic samples.

Problem 6. The dataset found in `creditcard.npy` contains information about credit card purchases made over a two day period. Of the approximately 285,000 observations, 492 are fraudulent purchases. The last column indicates if the purchase was valid (0) or fraudulent (1).

Do the following steps 10 times:

- Create a training and test set from the data using `train_test_split` from `sklearn.model_selection` with `test_size=.7`.
- Use `smote` with $N = 500$ and $k = 2$ to augment the training set.
- Create two Gaussian Naïve Bayes classifiers (from `sklearn.naive_bayes.GaussianNB`), one which wil be trained on only the original data and the other on the SMOTE augmented data and the original data.
- Fit each classifier and find the recall and accuracy of each model.

Print the mean recall and mean accuracy of each model and and describe the findings. HINT: Recall $= \frac{tp}{tp+fn}$. This example may be helpful for printing your results nicely.

```
>>> print('\t\t Recall \t Accuracy')
```

```
>>> print('Original','\t',np.round(mean_orig_recall,5),'\t', np.round(↩
    mean_orig_score,5))
>>> print('SMOTE', '\t\t', np.round(mean_smote_recall,5), '\t', np.round(↩
    mean_smote_score,5))
```