

9

Classification – Music Genre Classification

So far, we have had the luxury that every training data instance could easily be described by a vector of feature values. In the Iris dataset, for example, the flowers are represented by vectors containing values for length and width of certain aspects of a flower. In the text-based examples, we could transform the text into a bag of word representations and manually craft our own features that captured certain aspects of the texts.

It will be different in this chapter, when we try to classify songs by their genre. Or, how would we, for instance, represent a three-minute-long song? Should we take the individual bits of its MP3 representation? Probably not, since treating it like a text and creating something like a "bag of sound bites" would certainly be way too complex. Somehow, we will, nevertheless, have to convert a song into a series of values that describe it sufficiently.


Sketching our roadmap

This chapter will show how we can come up with a decent classifier in a domain that is outside our comfort zone. For one, we will have to use sound-based features, which are much more complex than the text-based ones we have used before. And then we will learn how to deal with multiple classes, whereas we have only encountered binary classification problems up to now. In addition, we will get to know new ways of measuring classification performance.

Let us assume a scenario in which, for some reason, we find a bunch of randomly named MP3 files on our hard disk, which are assumed to contain music. Our task is to sort them according to the music genre into different folders such as jazz, classical, country, pop, rock, and metal.


Fetching the music data

We will use the GTZAN dataset, which is frequently used to benchmark music genre classification tasks. It is organized into 10 distinct genres, of which we will use only 6 for the sake of simplicity: Classical, Jazz, Country, Pop, Rock, and Metal. The dataset contains the first 30 seconds of 100 songs per genre. We can download the dataset from <http://opihl.cs.uvic.ca/sound/genres.tar.gz>.

 The tracks are recorded at 22,050 Hz (22,050 readings per second) mono in the WAV format.

Converting into a WAV format

Sure enough, if we would want to test our classifier later on our private MP3 collection, we would not be able to extract much meaning. This is because MP3 is a lossy music compression format that cuts out parts that the human ear cannot perceive. This is nice for storing because with MP3 you can fit 10 times as many songs on your device. For our endeavor, however, it is not so nice. For classification, we will have an easier game with WAV files, because they can be directly read by the `scipy.io.wavfile` package. We would, therefore, have to convert our MP3 files in case we want to use them with our classifier.

 In case you don't have a conversion tool nearby, you might want to check out SoX at <http://sox.sourceforge.net>. It claims to be the Swiss Army Knife of sound processing, and we agree with this bold claim.

One advantage of having all our music files in the WAV format is that it is directly readable by the SciPy toolkit:

```
>>> sample_rate, X = scipy.io.wavfile.read(wave_filename)
```

`x` now contains the samples and `sample_rate` is the rate at which they were taken. Let us use that information to peek into some music files to get a first impression of what the data looks like.

Looking at music

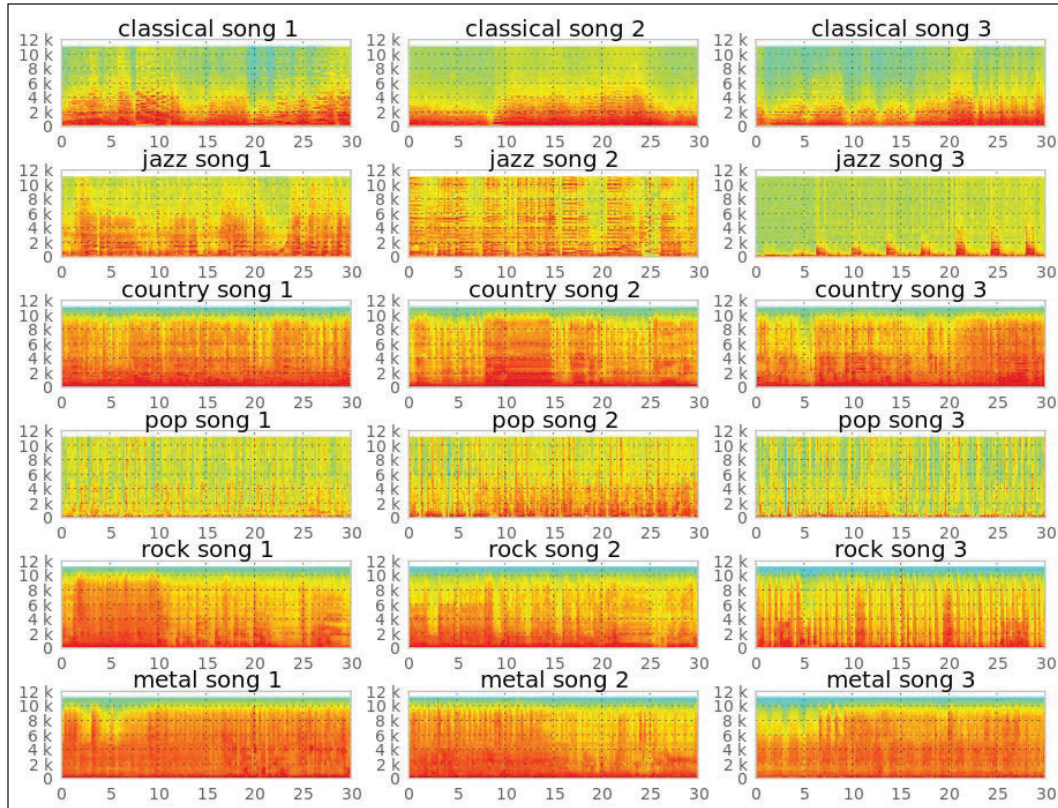
A very convenient way to get a quick impression of what the songs of the diverse genres "look" like is to draw a spectrogram for a set of songs of a genre. A spectrogram is a visual representation of the frequencies that occur in a song. It shows the intensity for the frequencies at the y axis in the specified time intervals at the x axis. That is, the darker the color, the stronger the frequency is in the particular time window of the song.

Matplotlib provides the convenient function `specgram()` that performs most of the under-the-hood calculation and plotting for us:

```
>>> import scipy
>>> from matplotlib.pyplot import specgram
>>> sample_rate, X = scipy.io.wavfile.read(wave_filename)
>>> print sample_rate, X.shape
22050, (661794,)
>>> specgram(X, Fs=sample_rate, xextent=(0,30))
```

The WAV file we just read in was sampled at a rate of 22,050 Hz and contains 661,794 samples.

If we now plot the spectrogram for these first 30 seconds for diverse WAV files, we can see that there are commonalities between songs of the same genre, as shown in the following image:



Just glancing at the image, we immediately see the difference in the spectrum between, for example, metal and classical songs. While metal songs have high intensity over most of the frequency spectrum all the time (they're energetic!), classical songs show a more diverse pattern over time.

It should be possible to train a classifier that discriminates at least between Metal and Classical songs with high enough accuracy. Other genre pairs like Country and Rock could pose a bigger challenge, though. This looks like a real challenge to us, since we need to discriminate not only between two classes, but between six. We need to be able to discriminate between all of them reasonably well.

Decomposing music into sine wave components

Our plan is to extract individual frequency intensities from the raw sample readings (stored in `x` earlier) and feed them into a classifier. These frequency intensities can be extracted by applying the so-called **fast Fourier transform (FFT)**. As the theory behind FFT is outside the scope of this chapter, let us just look at an example to get an intuition of what it accomplishes. Later on, we will treat it as a black box feature extractor.

For example, let us generate two WAV files, `sine_a.wav` and `sine_b.wav`, that contain the sound of 400 Hz and 3,000 Hz sine waves respectively. The aforementioned "Swiss Army Knife", SoX, is one way to achieve this:

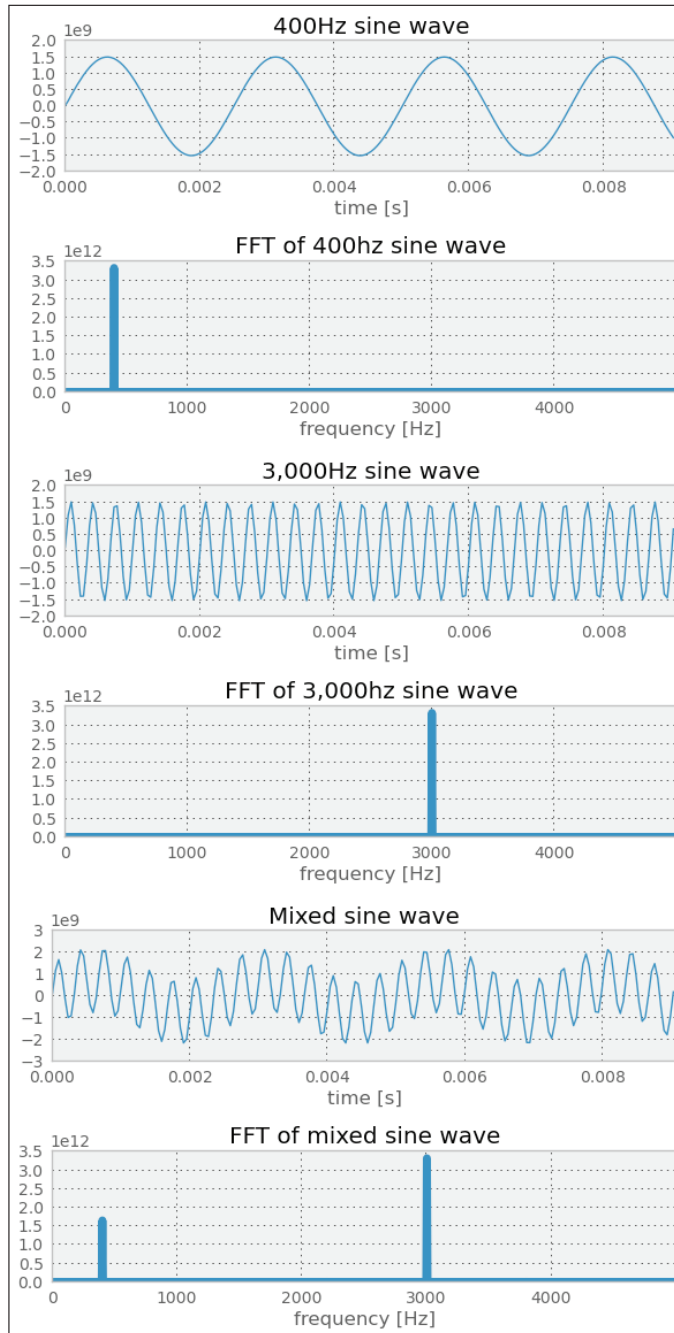
```
$ sox --null -r 22050 sine_a.wav synth 0.2 sine 400
$ sox --null -r 22050 sine_b.wav synth 0.2 sine 3000
```

In the following charts, we have plotted their first 0.008 seconds. Below we can see the FFT of the sine waves. Not surprisingly, we see a spike at 400 Hz and 3,000 Hz below the corresponding sine waves.

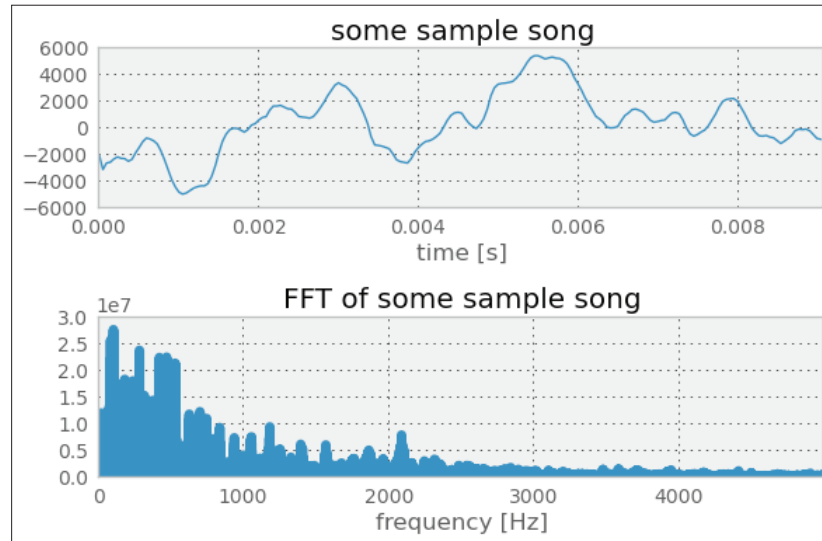
Now, let us mix them both, giving the 400 Hz sound half the volume of the 3,000 Hz one:

```
$ sox --combine mix --volume 1 sine_b.wav --volume 0.5 sine_a.wav
sine_mix.wav
```

We see two spikes in the FFT plot of the combined sound, of which the 3,000 Hz spike is almost double the size of the 400 Hz.



For real music, we quickly see that the FFT doesn't look as beautiful as in the preceding toy example:



Using FFT to build our first classifier

Nevertheless, we can now create some kind of musical fingerprint of a song using FFT. If we do that for a couple of songs and manually assign their corresponding genres as labels, we have the training data that we can feed into our first classifier.

Increasing experimentation agility

Before we dive into the classifier training, let us first spend some thoughts on experimentation agility. Although we have the word "fast" in FFT, it is much slower than the creation of the features in our text-based chapters. And because we are still in an experimentation phase, we might want to think about how we could speed up the whole feature creation process.

Of course, the creation of the FFT per file will be the same each time we are running the classifier. We could, therefore, cache it and read the cached FFT representation instead of the complete WAV file. We do this with the `create_fft()` function, which, in turn, uses `scipy.fft()` to create the FFT. For the sake of simplicity (and speed!), let us fix the number of FFT components to the first 1,000 in this example. With our current knowledge, we do not know whether these are the most important ones with regard to music genre classification—only that they show the highest intensities in the preceding FFT example. If we would later want to use more or fewer FFT components, we would of course have to recreate the FFT representations for each sound file.

```
import os
import scipy

def create_fft(fn):
    sample_rate, X = scipy.io.wavfile.read(fn)
    fft_features = abs(scipy.fft(X)[:1000])
    base_fn, ext = os.path.splitext(fn)
    data_fn = base_fn + ".fft"
    scipy.save(data_fn, fft_features)
```

We save the data using NumPy's `save()` function, which always appends `.npy` to the filename. We only have to do this once for every WAV file needed for training or predicting.

The corresponding FFT reading function is `read_fft()`:

```
import glob

def read_fft(genre_list, base_dir=GENRE_DIR):
    X = []
    y = []

    for label, genre in enumerate(genre_list):
        genre_dir = os.path.join(base_dir, genre, "*.fft.npy")
        file_list = glob.glob(genre_dir)

        for fn in file_list:
```



```

fft_features = scipy.load(fn)

X.append(fft_features[:1000])
y.append(label)

return np.array(X), np.array(y)

```

In our scrambled music directory, we expect the following music genres:

```
genre_list = ["classical", "jazz", "country", "pop", "rock", "metal"]
```

Training the classifier

Let us use the logistic regression classifier, which has already served us well in the *Chapter 6, Classification II - Sentiment Analysis*. The added difficulty is that we are now faced with a multiclass classification problem, whereas up to now we had to discriminate only between two classes.

Just to mention one aspect that is surprising is the evaluation of accuracy rates when first switching from binary to multiclass classification. In binary classification problems, we have learned that an accuracy of 50 percent is the worst case, as it could have been achieved by mere random guessing. In multiclass settings, 50 percent can already be very good. With our six genres, for instance, random guessing would result in only 16.7 percent (equal class sizes assumed).

Using a confusion matrix to measure accuracy in multiclass problems

With multiclass problems, we should not only be interested in how well we manage to correctly classify the genres. In addition, we should also look into which genres we actually confuse with each other. This can be done with the so-called confusion matrix, as shown in the following:

```

>>> from sklearn.metrics import confusion_matrix
>>> cm = confusion_matrix(y_test, y_pred)
>>> print(cm)
[[26  1  2  0  0  2]
 [ 4  7  5  0  5  3]
 [ 1  2 14  2  8  3]
 [ 5  4  7  3  7  5]
 [ 0  0 10  2 10 12]
 [ 1  0  4  0 13 12]]

```

This prints the distribution of labels that the classifier predicted for the test set for every genre. The diagonal represents the correct classifications. Since we have six genres, we have a six-by-six matrix. The first row in the matrix says that for 31 Classical songs (sum of first row), it predicted 26 to belong to the genre Classical, 1 to be a Jazz song, 2 to belong to the Country genre, and 2 to be Metal songs. The diagonal shows the correct classifications. In the first row, we see that out of $(26+1+2+2)=31$ songs, 26 have been correctly classified as classical and 5 were misclassifications. This is actually not that bad. The second row is more sobering: only 7 out of 24 Jazz songs have been correctly classified – that is, only 29 percent.

Of course, we follow the train/test split setup from the previous chapters, so that we actually have to record the confusion matrices per cross-validation fold. We have to average and normalize later on, so that we have a range between 0 (total failure) and 1 (everything classified correctly).

A graphical visualization is often much easier to read than NumPy arrays. The `matshow()` function of matplotlib is our friend:

```
from matplotlib import pylab

def plot_confusion_matrix(cm, genre_list, name, title):
    pylab.clf()
    pylab.matshow(cm, fignum=False, cmap='Blues',
                  vmin=0, vmax=1.0)

    ax = pylab.axes()    ax.set_xticks(range(len(genre_list)))
    ax.set_xticklabels(genre_list)
    ax.xaxis.set_ticks_position("bottom")
    ax.set_yticks(range(len(genre_list)))
    ax.set_yticklabels(genre_list)

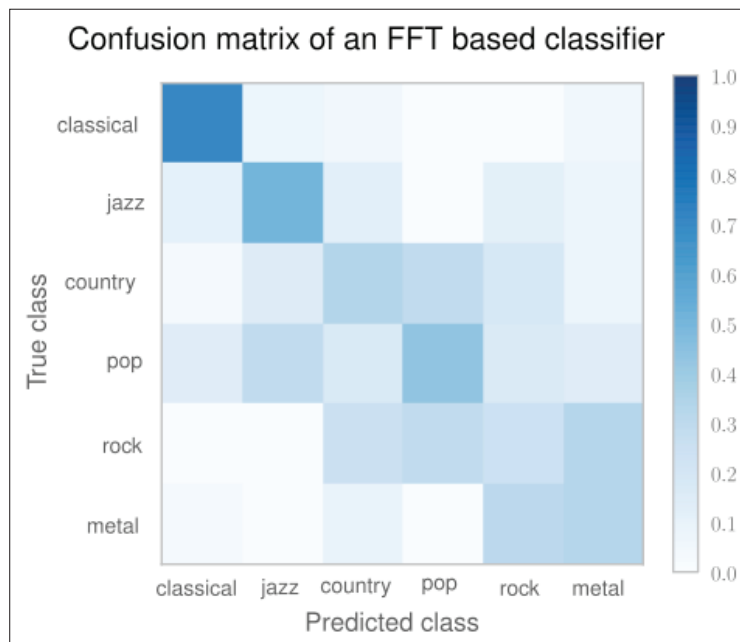
    pylab.title(title)
    pylab.colorbar()
    pylab.grid(False)
    pylab.xlabel('Predicted class')
    pylab.ylabel('True class')
    pylab.grid(False)

    pylab.show()
```



When you create a confusion matrix, be sure to choose a color map (the `cmap` parameter of `mat.show()`) with an appropriate color ordering so that it is immediately visible what a lighter or darker color means. Especially discouraged for these kinds of graphs are rainbow color maps, such as matplotlib's default `jet` or even the `Paired` color map.

The final graph looks like the following:



For a perfect classifier, we would have expected a diagonal of dark squares from the left-upper corner to the right lower one, and light colors for the remaining area. In the preceding graph, we immediately see that our FFT-based classifier is far away from being perfect. It only predicts Classical songs correctly (dark square). For Rock, for instance, it preferred the label Metal most of the time.

Obviously, using FFT points in the right direction (the Classical genre was not that bad), but is not enough to get a decent classifier. Surely, we can play with the number of FFT components (fixed to 1,000). But before we dive into parameter tuning, we should do our research. There we find that FFT is indeed not a bad feature for genre classification – it is just not refined enough. Shortly, we will see how we can boost our classification performance by using a processed version of it.

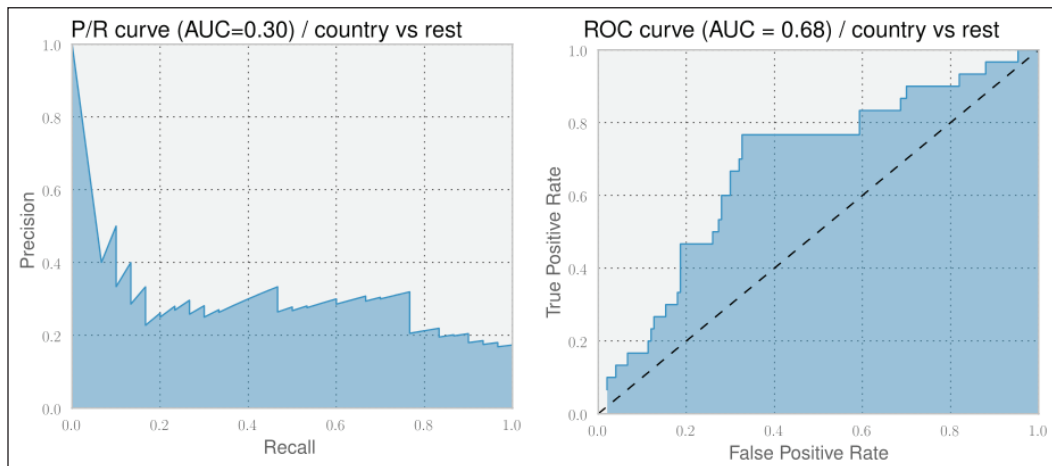
Before we do that, however, we will learn another method of measuring classification performance.

An alternative way to measure classifier performance using receiver-operator characteristics

We already learned that measuring accuracy is not enough to truly evaluate a classifier. Instead, we relied on **precision-recall (P/R)** curves to get a deeper understanding of how our classifiers perform.

There is a sister of P/R curves, called **receiver-operator-characteristics (ROC)**, which measures similar aspects of the classifier's performance, but provides another view of the classification performance. The key difference is that P/R curves are more suitable for tasks where the positive class is much more interesting than the negative one, or where the number of positive examples is much less than the number of negative ones. Information retrieval and fraud detection are typical application areas. On the other hand, ROC curves provide a better picture on how well the classifier behaves in general.

To better understand the differences, let us consider the performance of the previously trained classifier in classifying country songs correctly, as shown in the following graph:



On the left, we see the P/R curve. For an ideal classifier, we would have the curve going from the top left directly to the top right and then to the bottom right, resulting in an area under curve (AUC) of 1.0.

The right graph depicts the corresponding ROC curve. It plots the True Positive Rate over the False Positive Rate. There, an ideal classifier would have a curve going from the lower left to the top left, and then to the top right. A random classifier would be a straight line from the lower left to the upper right, as shown by the dashed line, having an AUC of 0.5. Therefore, we cannot compare an AUC of a P/R curve with that of an ROC curve.

Independent of the curve, when comparing two different classifiers on the same dataset, we are always safe to assume that a higher AUC of a P/R curve for one classifier also means a higher AUC of the corresponding ROC curve and vice versa. Thus, we never bother to generate both. More on this can be found in the very insightful paper *The Relationship Between Precision-Recall and ROC Curves* by Davis and Goadrich (ICML, 2006).

The following table summarizes the differences between P/R and ROC curves:

	x axis	y axis
P/R	$Recall = \frac{TP}{TP + FN}$	$Precision = \frac{TP}{TP + FP}$
ROC	$FPR = \frac{FP}{FP + TN}$	$TPR = \frac{TP}{TP + FN}$

Looking at the definitions of both curves' x and y axis, we see that the True Positive Rate in the ROC curve's y axis is the same as Recall of the P/R graph's x axis.

The False Positive Rate measures the fraction of true negative examples that were falsely identified as positive ones, giving a 0 in a perfect case (no false positives) and 1 otherwise. Contrast this to the precision, where we track exactly the opposite, namely the fraction of true positive examples that we correctly classified as such.

Going forward, let us use ROC curves to measure our classifiers' performance to get a better feeling for it. The only challenge for our multiclass problem is that both ROC and P/R curves assume a binary classification problem. For our purpose, let us, therefore, create one chart per genre that shows how the classifier performed a one versus rest classification:

```
from sklearn.metrics import roc_curve

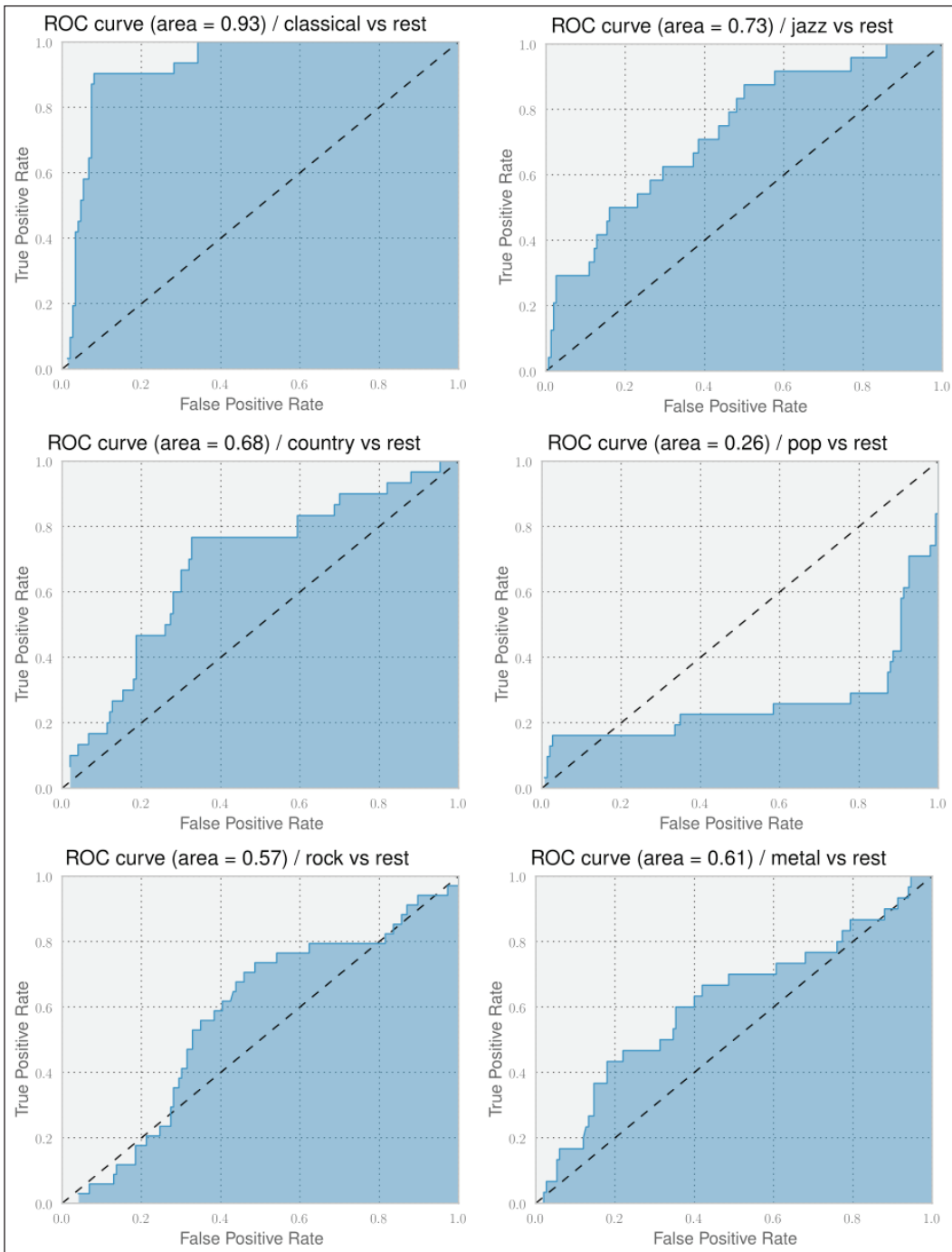
y_pred = clf.predict(X_test)

for label in labels:
    y_label_test = scipy.asarray(y_test==label, dtype=int)
    proba = clf.predict_proba(X_test)
    proba_label = proba[:,label]

    # calculate false and true positive rates as well as the
    # ROC thresholds
    fpr, tpr, roc_thres = roc_curve(y_label_test, proba_label)

    # plot tpr over fpr ...
```

The outcomes are the following six ROC plots. As we have already found out, our first version of a classifier only performs well on Classical songs. Looking at the individual ROC curves, however, tells us that we are really underperforming for most of the other genres. Only Jazz and Country provide some hope. The remaining genres are clearly not usable.



Improving classification performance with Mel Frequency Cepstral Coefficients

We already learned that FFT is pointing in the right direction, but in itself it will not be enough to finally arrive at a classifier that successfully manages to organize our scrambled directory of songs of diverse music genres into individual genre directories. We need a somewhat more advanced version of it.

At this point, it is always wise to acknowledge that we have to do more research. Other people might have had similar challenges in the past and already have found out new ways that might also help us. And, indeed, there is even a yearly conference dedicated to only music genre classification, organized by the **International Society for Music Information Retrieval (ISMIR)**. Apparently, **Automatic Music Genre Classification (AMGC)** is an established subfield of Music Information Retrieval. Glancing over some of the AMGC papers, we see that there is a bunch of work targeting automatic genre classification that might help us.

One technique that seems to be successfully applied in many of those works is called Mel Frequency Cepstral Coefficients. The **Mel Frequency Cepstrum (MFC)** encodes the power spectrum of a sound, which is the power of each frequency the sound contains. It is calculated as the Fourier transform of the logarithm of the signal's spectrum. If that sounds too complicated, simply remember that the name "cepstrum" originates from "spectrum" having the first four characters reversed. MFC has been successfully used in speech and speaker recognition. Let's see whether it also works in our case.

We are in a lucky situation in that someone else already needed exactly this and published an implementation of it as the Talkbox SciKit. We can install it from <https://pypi.python.org/pypi/scikits.talkbox>. Afterward, we can call the `mfcc()` function, which calculates the MFC coefficients, as follows:

```
>>> from scikits.talkbox.features import mfcc
>>> sample_rate, X = scipy.io.wavfile.read(fn)
>>> ceps, mspec, spec = mfcc(X)
>>> print(ceps.shape)
(4135, 13)
```


The data we would want to feed into our classifier is stored in `ceps`, which contains 13 coefficients (default value for the `nceps` parameter of `mfcc()`) for each of the 4,135 frames for the song with the filename `fn`. Taking all of the data would overwhelm our classifier. What we could do, instead, is to do an averaging per coefficient over all the frames. Assuming that the start and end of each song are possibly less genre specific than the middle part of it, we also ignore the first and last 10 percent:

```
x = np.mean(ceps[int(num_ceps*0.1):int(num_ceps*0.9)], axis=0)
```

Sure enough, the benchmark dataset we will be using contains only the first 30 seconds of each song, so that we would not need to cut off the last 10 percent. We do it, nevertheless, so that our code works on other datasets as well, which are most likely not truncated.

Similar to our work with FFT, we certainly would also want to cache the once generated MFCC features and read them instead of recreating them each time we train our classifier.

This leads to the following code:

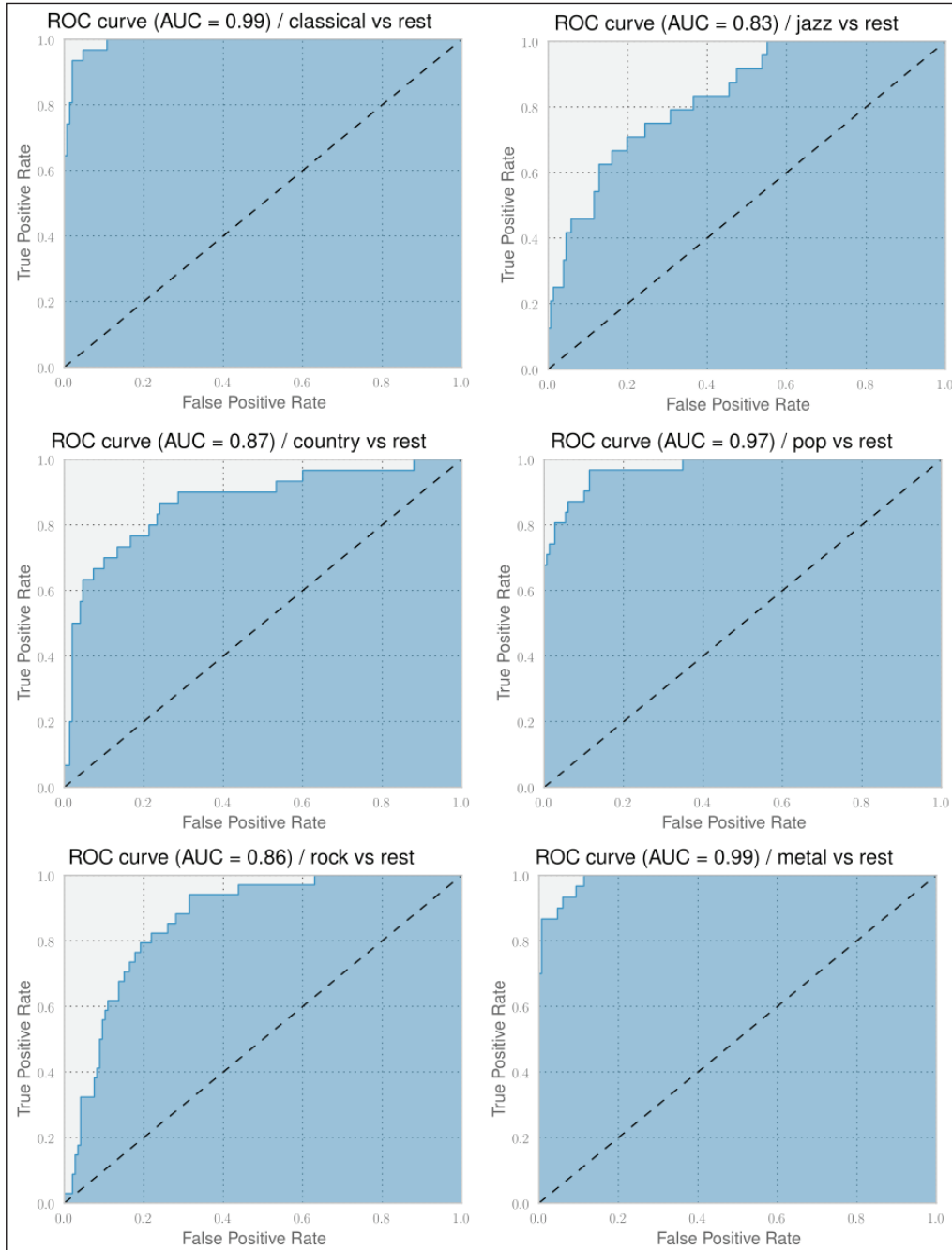
```
def write_ceps(ceps, fn):
    base_fn, ext = os.path.splitext(fn)
    data_fn = base_fn + ".ceps"
    np.save(data_fn, ceps)
    print("Written to %s" % data_fn)

def create_ceps(fn):
    sample_rate, X = scipy.io.wavfile.read(fn)
    ceps, mspec, spec = mfcc(X)
    write_ceps(ceps, fn)

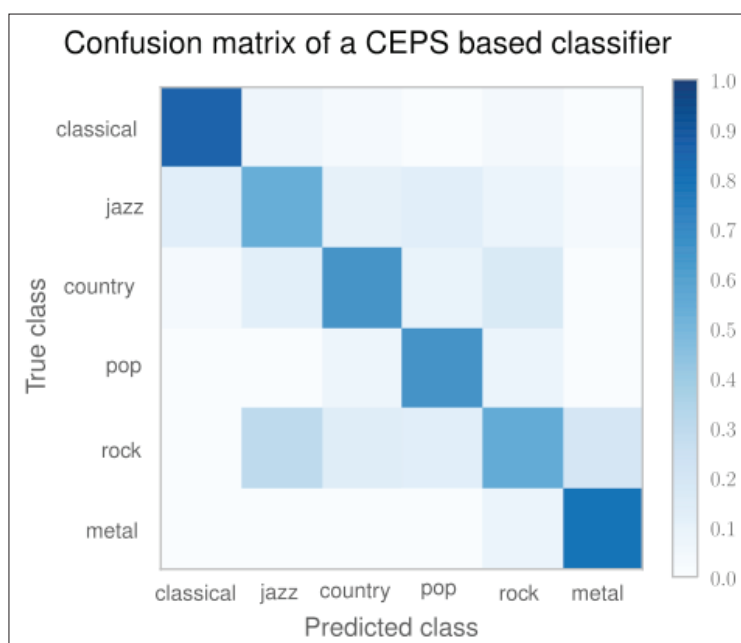
def read_ceps(genre_list, base_dir=GENRE_DIR):
    X, y = [], []
    for label, genre in enumerate(genre_list):
        for fn in glob.glob(os.path.join(
            base_dir, genre, "*.ceps.npy")):
            ceps = np.load(fn)
            num_ceps = len(ceps)
            X.append(np.mean(
                ceps[int(num_ceps*0.1):int(num_ceps*0.9)], axis=0))
            y.append(label)

    return np.array(X), np.array(y)
```

We get the following promising results with a classifier that uses only 13 features per song:



The classification performances for all genres have improved. Classical and Metal are even at almost 1.0 AUC. And indeed, also the confusion matrix in the following plot looks much better now. We can clearly see the diagonal showing that the classifier manages to classify the genres correctly in most of the cases. This classifier is actually quite usable to solve our initial task.



If we would want to improve on this, this confusion matrix tells us quickly what to focus on: the non-white spots on the non-diagonal places. For instance, we have a darker spot where we mislabel Rock songs as being Jazz with considerable probability. To fix this, we would probably need to dive deeper into the songs and extract things such as drum patterns and similar genre specific characteristics. And then – while glancing over the ISMIR papers – we also have read about the so-called **Auditory Filterbank Temporal Envelope (AFTE)** features, which seem to outperform MFCC features in certain situations. Maybe we should have a look at them as well?

The nice thing is that, only equipped with ROC curves and confusion matrices, we are free to pull in other experts' knowledge in terms of feature extractors without requiring ourselves to fully understand their inner workings. Our measurement tools will always tell us, when the direction is right and when to change it. Of course, being a machine learner who is eager to learn, we will always have the dim feeling that there is an exciting algorithm buried somewhere in a black box of our feature extractors, which is just waiting for us to be understood.

Summary

In this chapter, we totally stepped out of our comfort zone when we built a music genre classifier. Not having a deep understanding of music theory, at first we failed to train a classifier that predicts the music genre of songs with reasonable accuracy using FFT. But, then, we created a classifier that showed really usable performance using MFC features.

In both the cases, we used features that we understood only enough to know how and where to put them into our classifier setup. The one failed, the other succeeded. The difference between them is that in the second case we relied on features that were created by experts in the field.

And that is totally OK. If we are mainly interested in the result, we sometimes simply have to take shortcuts—we just have to make sure to take these shortcuts from experts in the specific domains. And because we had learned how to correctly measure the performance in this new multiclass classification problem, we took these shortcuts with confidence.

In the next chapter, we will look at how to apply techniques you have learned in the rest of this book to this specific type of data. We will learn how to use the mahotas computer vision package to preprocess images using traditional image processing functions.