lecture_4

October 1, 2019

[1]:
```
# %matplotlib notebook
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis as QDA
from sklearn.linear_model import LogisticRegression
```

1 Classification

In previous lectures we looked at the way we can use distributions to describe data, as well as for identifying inherent clustering/organisation within the properties of the data available. We can think of this approach as the unsupervised classification of data. If, however, we have labels for some of these data points (e.g., an object is tall, short, red, or blue) we can utilize this information to develop a relationship between the label and the properties of a source. We refer to this as supervised classification.

The motivation for supervised classification comes from the long history of classification in astronomy. Possibly the most well known of these classification schemes is that defined by Edwin Hubble for the morphological classification of galaxies based on their visual appearance. This simple classification scheme, subdividing the types of galaxies into seven categorical subclasses, was broadly adopted throughout extragalactic astronomy. Why such a simple classification became so predominant when subsequent works on the taxonomy of galaxy morphology (often with a better physical or mathematical grounding) did not, argues for the need to keep the models for classification simple. This agrees with the findings of George Miller who, in 1956, proposed that the number of items that people are capable of retaining within their short term memory was 7 ± 2!

1.1 Assigning Categories

Supervised classification takes a set of features and relates them to predefined sets of classes. We will not address how we define the labels or taxonomy for the classification other than noting that the time-honored system of having a graduate student label data does not scale to the size of today's data. We start by assuming that we have a set of predetermined labels that have been assigned to a subset of the data we are considering. Our goal is to characterize the relation between the

features in the data and their classes (machine learning) and apply these classifications to a larger set of unlabeled data (inference).

As we go we will illuminate the connections between classification, regression, and density estimation. Classification can be posed in terms of density estimation - this is called generative classification (so-called since we will have a full model of the density for each class, which is the same as saying we have a model which describes how data could be generated from each class). This will be our starting point, where we will visit a number of methods. Among the advantages of this approach is a high degree of interpretability.

Starting from the same principles we will go to classification methods that focus on finding the decision boundary that separates classes directly, avoiding the step of modeling each class's density, called discriminative classification, which can often be better in high-dimensional problems.

1.2 Classification Loss

Perhaps the most common loss (cost) function in classification is zero-one loss, where we assign a value of one for misclassification and zero for a correct classification. One particularly common case of classification in astronomy is that of "detection," where we wish to assign objects (i.e., regions of the sky or groups of pixels on a CCD) into one of two classes: a detection (usually with label 1) and a nondetection (usually with label 0). When thinking about this sort of problem, we may wish to distinguish between the two possible kinds of error: assigning a label 1 to an object whose true class is 0 (a "false positive"), and assigning the label 0 to an object whose true class is 1 (a "false negative").

Based on loss metrics such as the above, we can define two measures which will help us to determine how well our classifiers work. The first measure is completeness:

$$\text{completeness} = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}} \tag{1}$$

The second measure is contamination:

$$\text{contamination} = \frac{\text{false positives}}{\text{true positives} + \text{false negatives}} \tag{2}$$

The completeness measures the fraction of total detections identified by our classifier, while the contamination measures the fraction of detected objects which are misclassified. Depending on the nature of the problem and the goal of the classification, we may wish to optimize one or the other.

Alternative names for these measures abound: in some fields the completeness and contamination are respectively referred to as the "sensitivity" and the "Type I error". In astronomy, one minus the contamination is often referred to as the "efficiency". In machine learning communities, the efficiency and completeness are respectively referred to as the "precision" and "recall".

2 Naive Bayes Classification

Naive Bayes models are a group of extremely fast and simple classification algorithms that are often suitable for very high-dimensional datasets. Because they are so fast and have so few tunable parameters, they end up being very useful as a quick-and-dirty baseline for a classification problem.

Naive Bayes classifiers are built on Bayesian classification methods. These rely on Bayes's theorem, which is an equation describing the relationship of conditional probabilities of statistical quantities. In Bayesian classification, we're interested in finding the probability of a label given some observed features, which we can write as $P(L|\text{features})$. Bayes's theorem tells us how to express this in terms of quantities we can compute more directly:

$$P(L|\text{features}) = \frac{P(\text{features}|L)P(L)}{P(\text{features})} \tag{3}$$

If we are trying to decide between two labels - let's call them L1 and L2 - then one way to make this decision is to compute the ratio of the posterior probabilities for each label:

$$\frac{P(L_1|\text{features})}{P(L_2|\text{features})} = \frac{P(\text{features}|L_1)}{P(\text{features}|L_2)}\frac{P(L_1)}{P(L_2)} \tag{4}$$

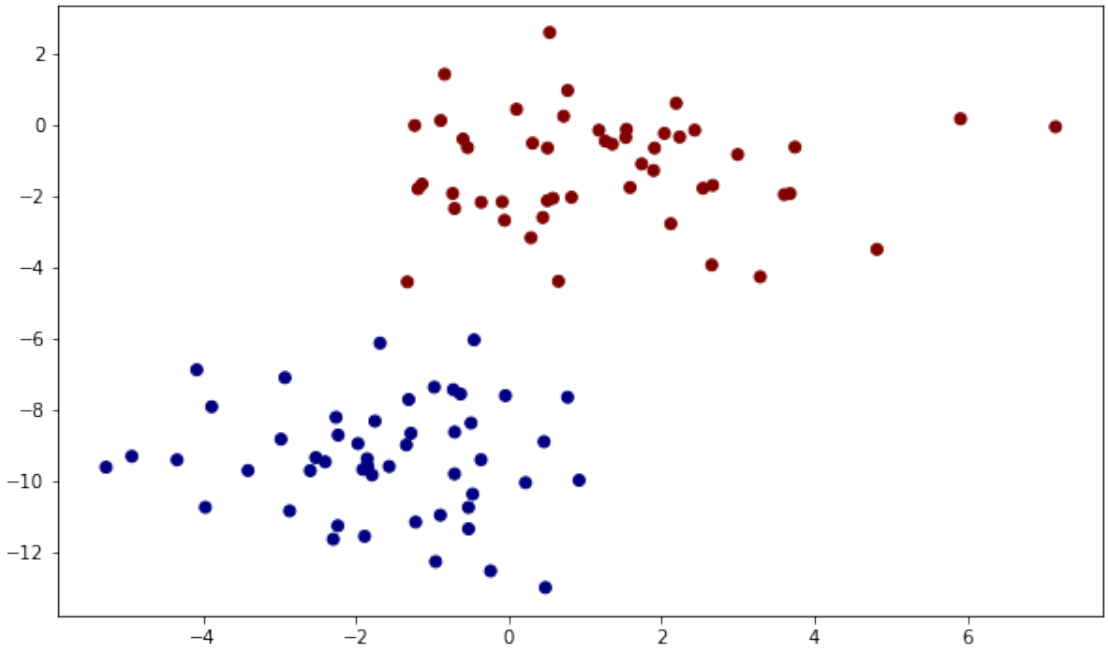
All we need now is some model by which we can compute $P(\text{features}|L_i)$ for each label. Such a model is called a generative model because it specifies the hypothetical random process that generates the data. Specifying this generative model for each label is the main piece of the training of such a Bayesian classifier. The general version of such a training step is a very difficult task, but we can make it simpler through the use of some simplifying assumptions about the form of this model.

This is where the "naive" in "naive Bayes" comes in: if we make very naive assumptions about the generative model for each label, we can find a rough approximation of the generative model for each class, and then proceed with the Bayesian classification. Different types of naive Bayes classifiers rest on different naive assumptions about the data, and we will examine a few of these in the following sections.

2.1 Gaussian Naive Bayes

Perhaps the easiest naive Bayes classifier to understand is Gaussian naive Bayes. In this classifier, the assumption is that data from each label is drawn from a simple Gaussian distribution. Imagine that you have the following data:

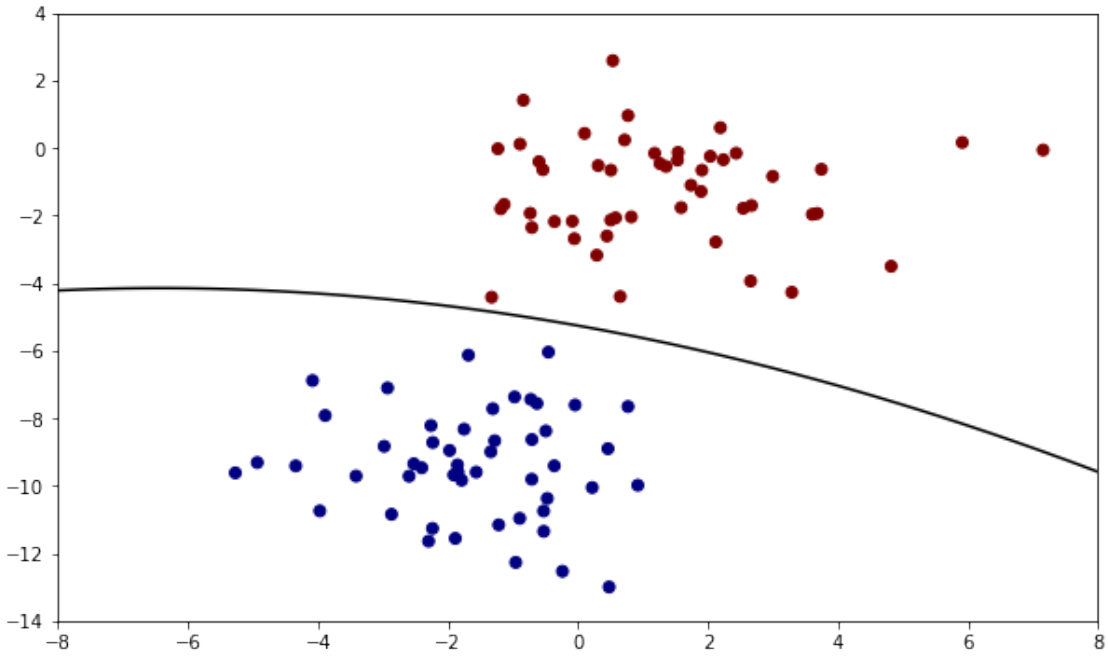
[2]:
```
from sklearn.datasets import make_blobs
X, y = make_blobs(100, 2, centers=2, random_state=2, cluster_std=1.5)
fig, ax0 = plt.subplots(figsize=(10, 6))
ax0.scatter(X[:, 0], X[:, 1], c=y, cmap=cm.jet)
plt.show()
```



One extremely fast way to create a simple model is to assume that the data is described by a Gaussian distribution with no covariance between dimensions. This model can be fit by simply finding the mean and standard deviation of the points within each label, which is all you need to define such a distribution. The result of this naive Gaussian assumption is shown in the following figure:

[3]:
```
# Fit the Naive Bayes classifier
gnb = GaussianNB()
gnb.fit(X, y)

# predict the classification probabilities on a grid
xlim = (-8, 8)
ylim = (-14, 4)
xx, yy = np.meshgrid(np.linspace(xlim[0], xlim[1], 100),
                     np.linspace(ylim[0], ylim[1], 100))

#convert to 1-D arrays
oned_xx = xx.ravel()
oned_yy = yy.ravel()

# get probabilities of each x,y coordinate from the NB classifier
# i.e. returns the probability of the samples for each class in the model.
Z = gnb.predict_proba(np.column_stack((oned_xx,oned_yy)))
Z = Z[:, 1].reshape(xx.shape)

# Plot the samples again
```

```
fig, ax = plt.subplots(figsize=(10, 6))
ax.scatter(X[:, 0], X[:, 1], c=y, cmap=cm.jet)

# Now plot the contour between the two classes at p=0.5
# i.e. for every coordinate in xx,yy, if z is at 0.5
# (equal probability) of both classes, then draw the contour at that coordinate
ax.contour(xx, yy, Z, [0.5], colors='k')

plt.show()
```



The assumption has generated a multivariate (2-D) Gaussian for each class. Towards the denser center of each class, the probability is higher, and falls the further we vary from the center. This normal distribution is done for both classes, and there is a point (the black line) where a sample could be generated from any of the two classes with equal probability.

When we utilized the 'gnb.predict_proba()' method of the Gaussian Naive Bayes classifier, the returned columns give the posterior probabilities of the first and second label, respectively. Since we measured a probability over an entire grid of possible samples, we have a map of probabilities for each class, at each point in the image. This information was used to draw the black contour delineating the point at which there is equal probabilitiy for each class to generate a sample along the contour.

2.1.1 Gaussian Naive Bayes for Star/LL-Rylae Classification

The Gaussian Naive Bayes classification we have considered so far was performed on a simple, well-separated data set. Examples like this one make classification straightforward, but data in the real world is rarely so clean. Instead, the distributions often overlap, and categories have hugely

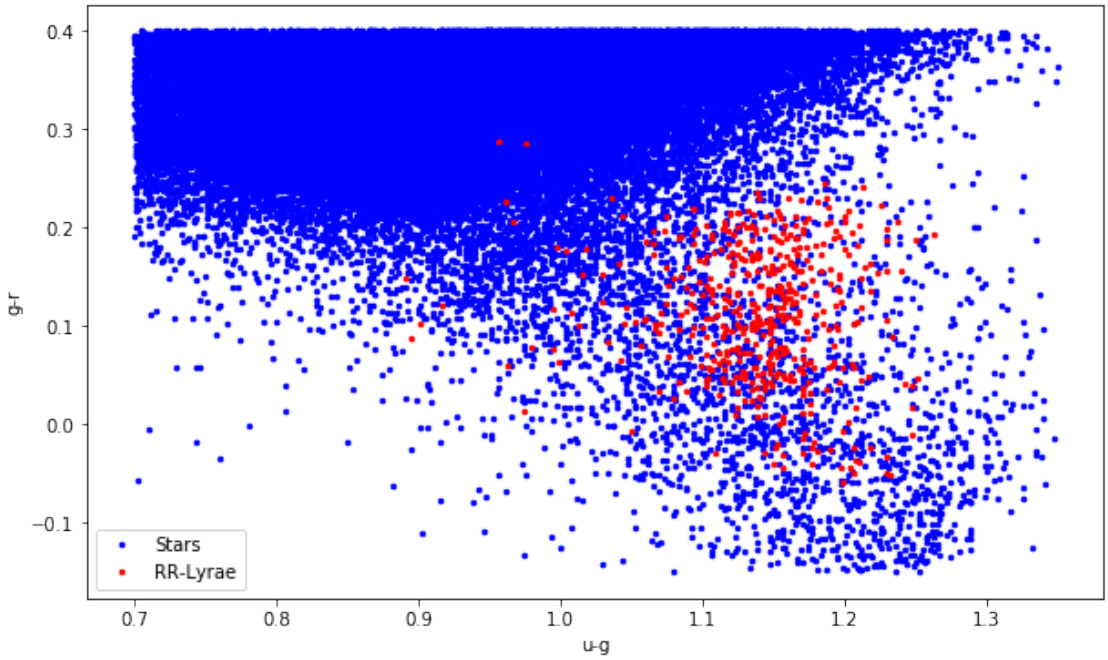
imbalanced numbers. To demonstrate this, we shall be applying Gaussian Naive Bayes classification to the RR Lyrae data set.

This data set is a set of photmetric observations of RR Lyrae stars in the Sloan Digital Sky Survey (SDSS). The particular dataset in question comes from SDSS Stripe 82, and combines the Stripe 82 standard stars, which represent observations of nonvariable stars; and the RR Lyrae variables, pulled from the same observations as the standard stars, and selected based on their variability using supplemental data. The sample is further constrained to a smaller region of the overall color–color space following ($0.7 < u-g < 1.35$, $-0.15 < g-r < 0.4$,$-0.15 < r-i < 0.22$, and $-0.21 < i-z < 0.25$).These selection criteria lead to a sample of 92,658 nonvariable stars, and 483 RR Lyraes. Two features of this combined data set make it a good candidate for testing classification algorithms:

1. The RR Lyrae stars and main sequence stars occupy a very similar region in u,g,r,i,z color space. The distributions overlap slightly, which makes the choice of decision boundaries subject to the completeness and contamination trade-off discussed earlier.
2. The extreme imbalance between the number of sources and the number of background objects is typical of real-world astronomical studies, where it is often desirable to select rare events out of a large background. Such unbalanced data aptly illustrates the strengths and weaknesses of various classification methods.

Let us have a look at the dataset in question:

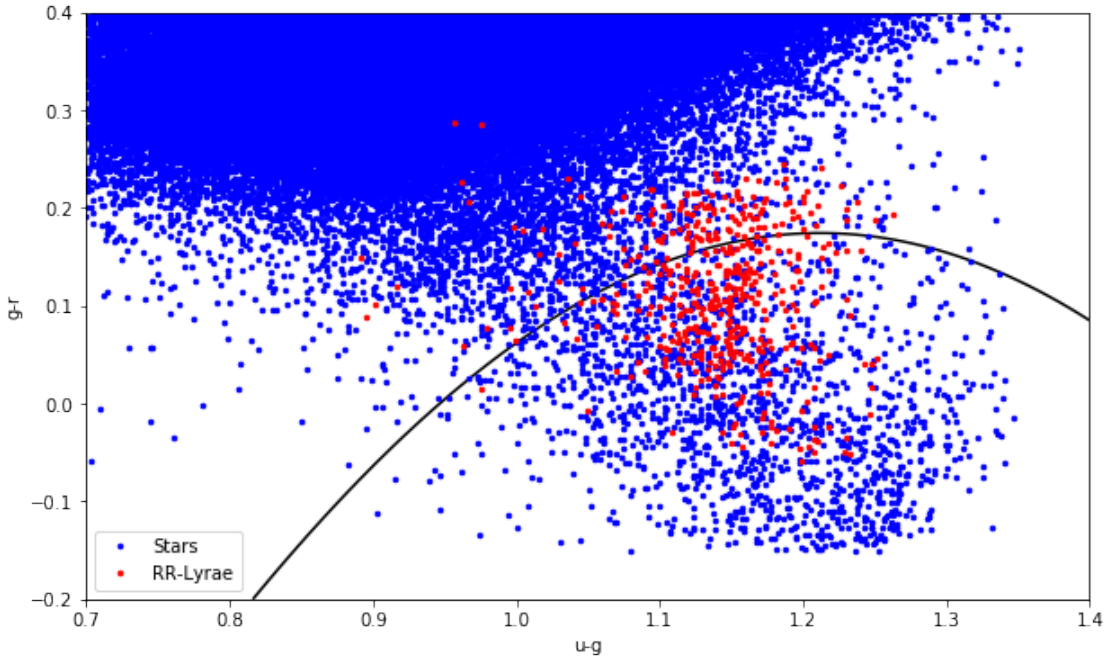
[4]:
```
# Load up RRLyrae Dataset
data = np.load('./data/rrlyrae.npz')
samples = data['data']
labels = data['labels']

# stars are indicated by 0 labels, and rrlyrae by 1 labels
stars = (labels == 0)
rrlyrae = (labels == 1)

# plot the results
fig, ax_data = plt.subplots(figsize=(10, 6))
ax_data = plt.axes()
ax_data.plot(samples[stars, 0], samples[stars, 1], '.', ms=5, c='b',␣
 ↪label='Stars')
ax_data.plot(samples[rrlyrae, 0], samples[rrlyrae, 1], '.', ms=5, c='r',␣
 ↪label='RR-Lyrae')

ax_data.legend(loc=3)
ax_data.set_xlabel('u-g')
ax_data.set_ylabel('g-r')

plt.show()
```



In the figure above, the blue samples show non-variable sources, while the red points show variable sources. We are only plotting two features for each sample, i.e. u-g vs. g-r. There are of course 4 features for the dataset. Let us try out Gaussian Naive Bayes classification to this highly overlapping dataset:

[5]:
```
# Fit the Naive Bayes classifier to the first 2 dimensions
samples_2d = samples[:,0:2]
gnb = GaussianNB()
gnb.fit(samples_2d, labels)

# predict the classification probabilities on a grid
xlim = (0.7, 1.4)
ylim = (-0.2, 0.4)
xx, yy = np.meshgrid(np.linspace(xlim[0], xlim[1], 100),
                     np.linspace(ylim[0], ylim[1], 100))

#convert to 1-D arrays
oned_xx = xx.ravel()
oned_yy = yy.ravel()

# get probabilities of each x,y coordinate from the NB classifier
# i.e. returns the probability of the samples for each class in the model.
Z = gnb.predict_proba(np.column_stack((oned_xx,oned_yy)))
Z = Z[:, 1].reshape(xx.shape)

# Plot the samples again
```

```
fig, ax_gnb = plt.subplots(figsize=(10, 6))
ax_gnb = plt.axes()
ax_gnb.plot(samples[stars, 0], samples[stars, 1], '.', ms=5, c='b',␣
 ↪label='Stars')
ax_gnb.plot(samples[rrlyrae, 0], samples[rrlyrae, 1], '.', ms=5, c='r',␣
 ↪label='RR-Lyrae')

ax_gnb.legend(loc=3)
ax_gnb.set_xlabel('u-g')
ax_gnb.set_ylabel('g-r')

# Now plot the contour between the two classes at p=0.5
# i.e. for every coordinate in xx,yy, if z is at 0.5
# (equal probability) of both classes, then draw the contour at that coordinate
ax_gnb.contour(xx, yy, Z, [0.5], colors='k')

plt.show()
```



The classification boundary is unable to separate regions of high overlap between the two categories, but it gives the best possible separation based on the generative model employed - a multivariate Gaussian distribution.

We can now look at the predictions (classifications) of the data that was used to train the model, and analyze it via the criteria of completeness and contamination. For this purpose, we will retrain the model to employ multivariate Gaussians covering all dimensions in the dataset:

[6]:
```
# Fit the Naive Bayes classifier to all original dimensions
gnb = GaussianNB()
gnb.fit(samples, labels)

# now predict
labels_pred = gnb.predict(samples)

#get completeness score (equivalent to recall)
completeness_score = recall_score(labels,labels_pred)
#get contamination score (equivalent to 1-precision)
contamination_score = (1-precision_score(labels,labels_pred))

print('Completeness: %f'%completeness_score)
print('Contamination: %f'%contamination_score)
```

```
Completeness: 0.857143
Contamination: 0.823001
```

2.1.2 Gaussian Naive Bayes Recap

This section introduced the concept of Naive Bayes classification, based on Gaussian models. Of course, any model that provides predictions for a sample can be used in this framework - it need not be Gaussian. We have also introduced and demonstrated a number of classification concepts which will be useful for other classification methods.

As a rule of thumb, when should you be inclined to use Naive Bayes classification? Keep in mind that you are making stringent assumptions about the data based on the particular model of choice. That being said, Naive Bayes classifiers have a number of advantages: * They are extremely fast for both training and prediction * They provide straightforward probabilistic prediction * They are often very easy to interpret * They have very few (if any) parameters that need tuning

These advantages mean a naive Bayesian classifier is often a good choice as an initial baseline classification. If it performs suitably, then congratulations: you have a very fast, very interpretable classifier for your problem. If it does not perform well, then you can begin exploring more sophisticated models, with some baseline knowledge of how well they should perform.

Naive Bayes classifiers tend to perform especially well in one of the following situations: * When the naive assumptions actually match the data (very rare in practice) * For very well-separated categories, when model complexity is less important * For very high-dimensional data, when model complexity is less important

The last two points seem distinct, but they actually are related: as the dimension of a dataset grows, it is much less likely for any two points to be found close together (after all, they must be close in every single dimension to be close overall). This means that clusters in high dimensions tend to be more separated, on average, than clusters in low dimensions, assuming the new dimensions actually add information. For this reason, simplistic classifiers like naive Bayes tend to work as well or better than more complicated classifiers as the dimensionality grows: once you have enough data, even a simple model can be very powerful.

2.2 Linear and Quadratic Discriminant Analysis

Linear Discriminant Analysis (LDA), like Gaussian Naive Bayes, relies on some simplifying assumptions about the class distribution. In particular, it assumes that these distributions have identical covariances for all the K classes. This makes all classes a set of shifted Gaussians.

The optimal classifier can then be derived from the log of the class posteriors to be:

$$g_k(x) = x^T \Sigma^{-1} \mu_k - \frac{1}{2} \mu_k^T \Sigma^{-1} + log \pi_k \tag{5}$$

With μ_k as the mean of class k, and Σ the covariance of the Gaussians (shared across all classes). The class dependent covariancs that would normally give rise to a quadratic dependence on x, cancel out if they are assumed to be constant. The Bayes cassifier is, therefore, linear with respect to x.

The discriminant boundary between the classes, is going to be linear. There is a discriminant boundary between each pair of classes in the dataset. The minimized overlap between the two Gaussians that define the pair of classes is given as:

$$g_k(x) - g_l(x) = x^T \Sigma^{-1} (\mu_k - \mu_l) - \frac{1}{2} (\mu_k - \mu_l)^T \Sigma^{-1} (\mu_k - \mu_l) + log \left(\frac{\pi_k}{\pi_l} \right) = 0 \tag{6}$$

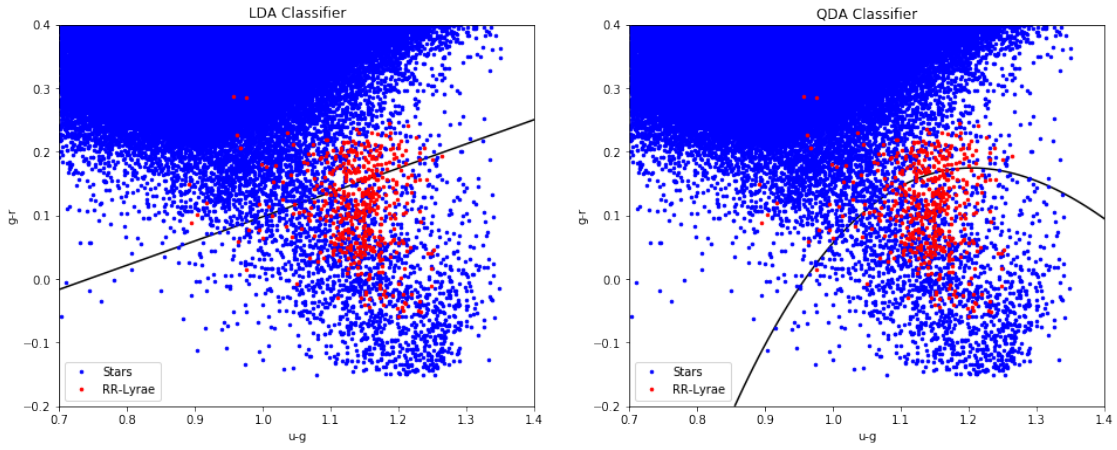
If we were to relax the requirement that the covariances of the Gaussians are constant, the discriminant function for the classes becomes quadratic in x:

$$g_k(x) = -\frac{1}{2} log |\Sigma_k| - \frac{1}{2} (x - \mu_k)^T C^{-1} (x - \mu_k) + log \pi_k \tag{7}$$

This is sometimes known as Quadratic Discriminant Analysis (QDA), and the boundary between classes is described by a quadratic function of the features x. In Python, we can utilize LDA/QDA classifiers for the same star vs RR-Lyrae problem we had earlier like so:

[7]:
```
# Load up RRLyrae Dataset
data = np.load('./data/rrlyrae.npz')
samples = data['data']
labels = data['labels']

# Fit LDA/QDA to the first 2 dimensions
samples_2d = samples[:,0:2]

# fit an LDA classifier and get predictions
lda = LDA()
lda.fit(samples_2d, labels)
lda_labels_pred = lda.predict(samples_2d)

# fit a QDA classifier and get predictions
qda = QDA()
qda.fit(samples_2d, labels)
qda_labels_pred = qda.predict(samples_2d)
```

```
# Create 2 subplots to view LDA/QDA together
f, (ax_lda, ax_qda) = plt.subplots(1, 2, figsize=(16, 6))

# predict the LDA classification probabilities on a grid (we've seen this␣
 ↪before)
xlim = (0.7, 1.4)
ylim = (-0.2, 0.4)
xx, yy = np.meshgrid(np.linspace(xlim[0], xlim[1], 100),
                     np.linspace(ylim[0], ylim[1], 100))
oned_xx = xx.ravel()
oned_yy = yy.ravel()
Z = lda.predict_proba(np.column_stack((oned_xx,oned_yy)))
Z = Z[:, 1].reshape(xx.shape)
ax_lda.plot(samples[stars, 0], samples[stars, 1], '.', ms=5, c='b',␣
 ↪label='Stars')
ax_lda.plot(samples[rrlyrae, 0], samples[rrlyrae, 1], '.', ms=5, c='r',␣
 ↪label='RR-Lyrae')
ax_lda.legend(loc=3)
ax_lda.set_xlabel('u-g')
ax_lda.set_ylabel('g-r')
ax_lda.set_title('LDA Classifier')
ax_lda.contour(xx, yy, Z, [0.5], colors='k')

# predict the QDA classification probabilities on a grid (we've seen this␣
 ↪before)
xlim = (0.7, 1.4)
ylim = (-0.2, 0.4)
xx, yy = np.meshgrid(np.linspace(xlim[0], xlim[1], 100),
                     np.linspace(ylim[0], ylim[1], 100))
oned_xx = xx.ravel()
oned_yy = yy.ravel()
Z = qda.predict_proba(np.column_stack((oned_xx,oned_yy)))
Z = Z[:, 1].reshape(xx.shape)
ax_qda.plot(samples[stars, 0], samples[stars, 1], '.', ms=5, c='b',␣
 ↪label='Stars')
ax_qda.plot(samples[rrlyrae, 0], samples[rrlyrae, 1], '.', ms=5, c='r',␣
 ↪label='RR-Lyrae')
ax_qda.legend(loc=3)
ax_qda.set_xlabel('u-g')
ax_qda.set_ylabel('g-r')
ax_qda.set_title('QDA Classifier')
ax_qda.contour(xx, yy, Z, [0.5], colors='k')

plt.show()
```



The results of LDA and QDA show, true to their names, a linear boundary between the two classes for LDA, while QDA results in a quadratic boundary. As may be expected with a more sophisticated model, QDA yields improved completeness and contamination in comparison to LDA. Also, and recall what assumptions were made to generate a QDA classifier. We relaxed the original LDA requirement that the covariances of the Gaussians are constant. What happens is that the two classes are defined by two different Gaussians. Is that not also the way the Gaussian Naive Bayes classifier was built? Indeed, QDA and Gaussian Naive Bayes would be equivalent if we further constrain Σ to be a diagonal covariance matrix - something we do not force for QDA, but generally do for Gaussian Naive Bayes.

I am sure you are curious on how well these classifiers perform compared to Gaussian Naive Bayes. Let us check them out:

[8]:
```
# Fit the LDA classifier to all original dimensions
lda = LDA()
lda.fit(samples, labels)
lda_labels_pred = lda.predict(samples)

# get LDA scores
completeness_score = recall_score(labels,lda_labels_pred)
contamination_score = (1-precision_score(labels,lda_labels_pred))
print('LDA')
print('Completeness: %f'%completeness_score)
print('Contamination: %f'%contamination_score)


# Fit the QDA classifier to all original dimensions
qda = QDA()
qda.fit(samples, labels)
qda_labels_pred = qda.predict(samples)

#get QDA scores
```

```
completeness_score = recall_score(labels,qda_labels_pred)
contamination_score = (1-precision_score(labels,qda_labels_pred))
print('QDA')
print('Completeness: %f'%completeness_score)
print('Contamination: %f'%contamination_score)
```

```
LDA
Completeness: 0.670807
Contamination: 0.835533
QDA
Completeness: 0.774327
Contamination: 0.786530
```

3 Discriminative Classification

With the classifiers we looked at so far, the idea was focused on analyzing the data for each particular class, and to find some distribution that characterises the data for the class well, whilst providing low probabilities to samples from other classes. We termed this form of classification as generative classification, as the end result is a distribution that could generate samples similar to the actual data used to model it in the first place. This inadvertently results in a cross over in the data space where the distributions for both classes give the same probability for a particular data point. We delineated this contour in the examples so far.

Another form of classification is discriminative classification where the class density estimates are skipped in favour of a classification decision boundary. We therefore do not tie the boundary by assumptions about the distribution of the data.

3.1 Logistic Regression

Logistic regression is a predictive modelling algorithm that is used when the class variable (Y) is binary categorical. That is, it can take only two values like 1 or 0. The goal is to determine a mathematical equation that can be used to predict the probability of event 1. Once the equation is established, it can be used to predict the class when only the sample points (X's) are known.

Earlier we saw what is linear regression and how to use it to predict continuous Y variables for an input X. In linear regression the Y variable is always a continuous variable. If suppose, the Y variable was categorical, you cannot use linear regression to model it. So what would you do when the Y is a categorical variable with 2 classes - besides the methods we discussed for generative classification?

Logistic regression is a discriminative classifier that can be used to model and solve such problems, also called as binary classification problems. A key point to note here is that Y can have 2 classes only and not more than that. If Y has more than 2 classes, it would become a multi-class classification and you can no longer use the vanilla logistic regression for that. You would have to build a "one-versus-all" type binary classifier for every class in the dataset, reducing the problem to multiple binary classifiers.

Even so, logistic regression is a classic predictive modelling technique and still remains a popular choice for modelling binary categorical variables. Another advantage of logistic regression is that it

computes a prediction probability score of an event. More on that as we start building the models.

The example we have considered so far (Star vs RR-Lyrae classification), is a binary classification task for which logistic regression is suitable. So let us get down to visualzing the main difference between the concept of linear regression and logistic regression.

When the response variable Y has only 2 possible values, it is desirable to have a model that predicts the value either as 0 or 1 or as a probability score that ranges between 0 and 1. Linear regression does not have this capability. Because, if you use linear regression to model a binary response variable, the resulting model may not restrict the predicted Y values within 0 and 1.

This is where logistic regression comes into play. In logistic regression, you get a probability score that reflects the probability of the occurence of the event. An event in this case is each sample of the training dataset. It could be something like classifying if a given value in a color-color space belongs to a star or not.

Logistic regression achieves this by taking the log odds of the event $ln(P/1-P)$, where, P is the probability of an event/sample. So P always lies between 0 and 1. The logistic regression formulation is:

$$p(y = 1|x) = \frac{\exp\left[\sum_j \theta_j x^j\right]}{1 + \exp\left[\sum_j \theta_j x^j\right]} \tag{8}$$

where the logistic function (logit) is defined as:

$$\text{logit}(p_i) = \log\left(\frac{p_i}{1 - p_i}\right) = \sum_j \theta_j x_i^j \tag{9}$$

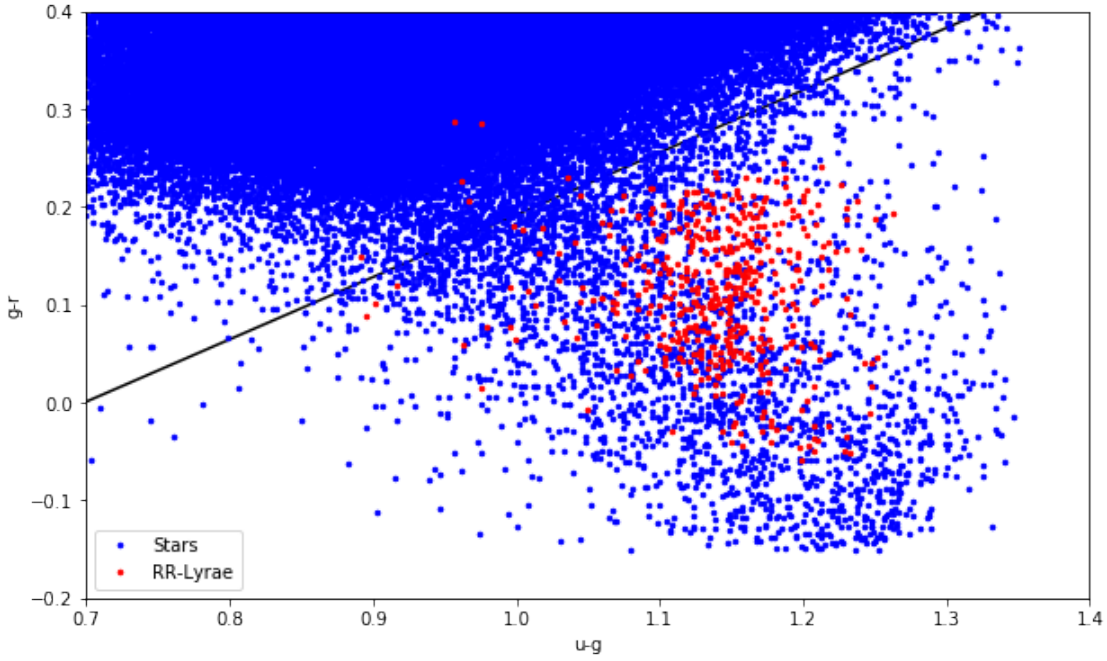
The name logistic regression comes from the fact that the function $e^x/(1 + e^x)$ is called the logistic (or sigmoid) function. Its name is due to its roots in regression, even though it is a method for classification.

Let us look at how logistic regression performs on our sample dataset:

[9]:
```
# Load up RRLyrae Dataset
data = np.load('./data/rrlyrae.npz')
samples = data['data']
labels = data['labels']

# Fit LOGIT to the first 2 dimensions
samples_2d = samples[:,0:2]

# fit a LOGIT classifier and get predictions
logit = LogisticRegression(class_weight='balanced')
logit.fit(samples_2d, labels)
logit_labels_pred = logit.predict(samples_2d)

# predict the classification probabilities on a grid
xlim = (0.7, 1.4)
```

```
ylim = (-0.2, 0.4)
xx, yy = np.meshgrid(np.linspace(xlim[0], xlim[1], 100),
                     np.linspace(ylim[0], ylim[1], 100))
oned_xx = xx.ravel()
oned_yy = yy.ravel()
Z = logit.predict_proba(np.column_stack((oned_xx,oned_yy)))
Z = Z[:, 1].reshape(xx.shape)
fig, ax_logit = plt.subplots(figsize=(10, 6))
ax_logit = plt.axes()
ax_logit.plot(samples[stars, 0], samples[stars, 1], '.', ms=5, c='b',␣
 ↪label='Stars')
ax_logit.plot(samples[rrlyrae, 0], samples[rrlyrae, 1], '.', ms=5, c='r',␣
 ↪label='RR-Lyrae')
ax_logit.legend(loc=3)
ax_logit.set_xlabel('u-g')
ax_logit.set_ylabel('g-r')

# Now plot the contour between the two classes at p=0.5
# i.e. for every coordinate in xx,yy, if z is at 0.5
# (equal probability) of both classes, then draw the contour at that coordinate
ax_logit.contour(xx, yy, Z, [0.5], colors='k')

plt.show()
```

```
/Users/andrea/Work/ISSA/Lecturing/PHY3287/venv/lib/python3.7/site-
packages/sklearn/linear_model/logistic.py:432: FutureWarning: Default solver
will be changed to 'lbfgs' in 0.22. Specify a solver to silence this warning.
  FutureWarning)
```

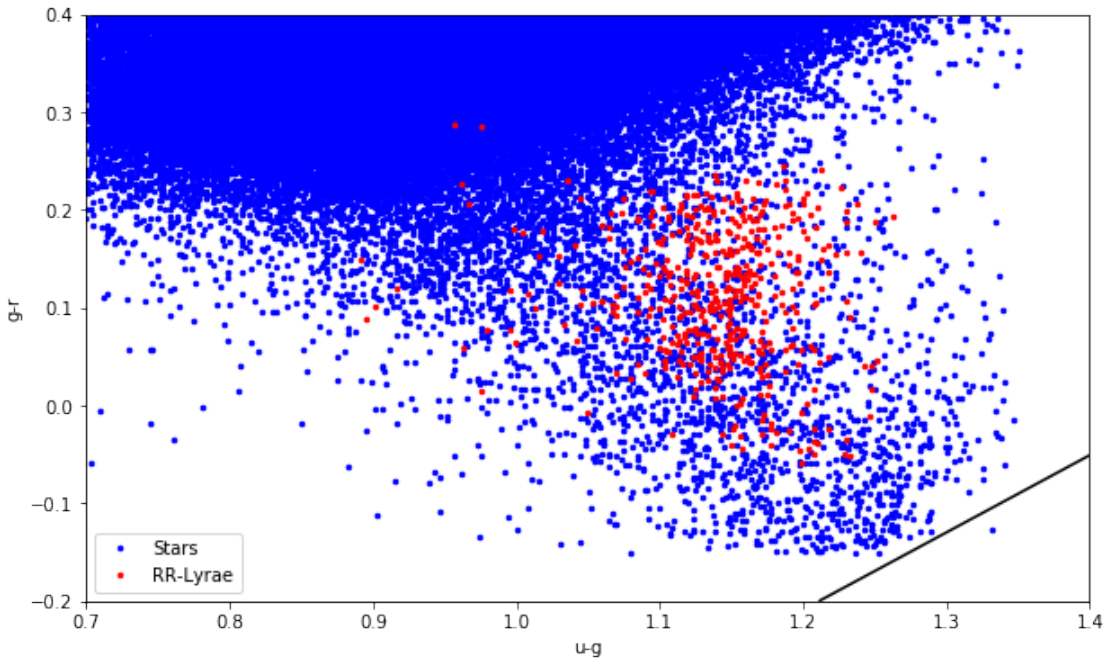


In general, there are many tweakable parameters that can be tried for logistic regression in Python. In the example above, we have used one particular parameter called 'class_weight', and set this to 'balanced'. This is quite imporant for our dataset. Each class is given a weight. By default, all classes are supposed to have a weight of one. What this means is that there are the same amount of samples for each class. In the case of our dataset, this is not true, and would result in a very bad classification boundary. To fix this problem, we indicate that we would like to balance the class weights, inversely proportional to the class sample frequencies in the input data. If we had ignored the data imbalance present in the dataset, the resulting classifiction boundary would have been heavily biased towards classifying almost all samples as stars:

[10]:
```
# Load up RRLyrae Dataset
data = np.load('./data/rrlyrae.npz')
samples = data['data']
labels = data['labels']

# Fit LOGIT to the first 2 dimensions
samples_2d = samples[:,0:2]

# fit a LOGIT classifier and get predictions
logit = LogisticRegression()
logit.fit(samples_2d, labels)
logit_labels_pred = logit.predict(samples_2d)

# predict the classification probabilities on a grid
xlim = (0.7, 1.4)
ylim = (-0.2, 0.4)
xx, yy = np.meshgrid(np.linspace(xlim[0], xlim[1], 100),
                     np.linspace(ylim[0], ylim[1], 100))
oned_xx = xx.ravel()
oned_yy = yy.ravel()
Z = logit.predict_proba(np.column_stack((oned_xx,oned_yy)))
Z = Z[:, 1].reshape(xx.shape)
fig, ax_logit2 = plt.subplots(figsize=(10, 6))
ax_logit2 = plt.axes()
ax_logit2.plot(samples[stars, 0], samples[stars, 1], '.', ms=5, c='b',␣
 ↪label='Stars')
ax_logit2.plot(samples[rrlyrae, 0], samples[rrlyrae, 1], '.', ms=5, c='r',␣
 ↪label='RR-Lyrae')
ax_logit2.legend(loc=3)
ax_logit2.set_xlabel('u-g')
ax_logit2.set_ylabel('g-r')

# Now plot the contour between the two classes at p=0.5
# i.e. for every coordinate in xx,yy, if z is at 0.5
# (equal probability) of both classes, then draw the contour at that coordinate
```

```
ax_logit2.contour(xx, yy, Z, [0.5], colors='k')

plt.show()
```

```
/Users/andrea/Work/ISSA/Lecturing/PHY3287/venv/lib/python3.7/site-
packages/sklearn/linear_model/logistic.py:432: FutureWarning: Default solver
will be changed to 'lbfgs' in 0.22. Specify a solver to silence this warning.
  FutureWarning)
```



Now that we have built a logistic regression classifier, the next thing to do is to run our standard classifier scoring procedure for completeness and contamination:

[11]:
```
# fit a LOGIT classifier and get predictions
logit = LogisticRegression(class_weight='balanced')
logit.fit(samples, labels)
logit_labels_pred = logit.predict(samples)

# get LOGIT scores
completeness_score = recall_score(labels,logit_labels_pred)
contamination_score = (1-precision_score(labels,logit_labels_pred))
print('LOGIT')
print('Completeness: %f'%completeness_score)
print('Contamination: %f'%contamination_score)
```

```
LOGIT
Completeness: 0.989648
```

```
Contamination: 0.857484
```

```
/Users/andrea/Work/ISSA/Lecturing/PHY3287/venv/lib/python3.7/site-
packages/sklearn/linear_model/logistic.py:432: FutureWarning: Default solver
will be changed to 'lbfgs' in 0.22. Specify a solver to silence this warning.
  FutureWarning)
```

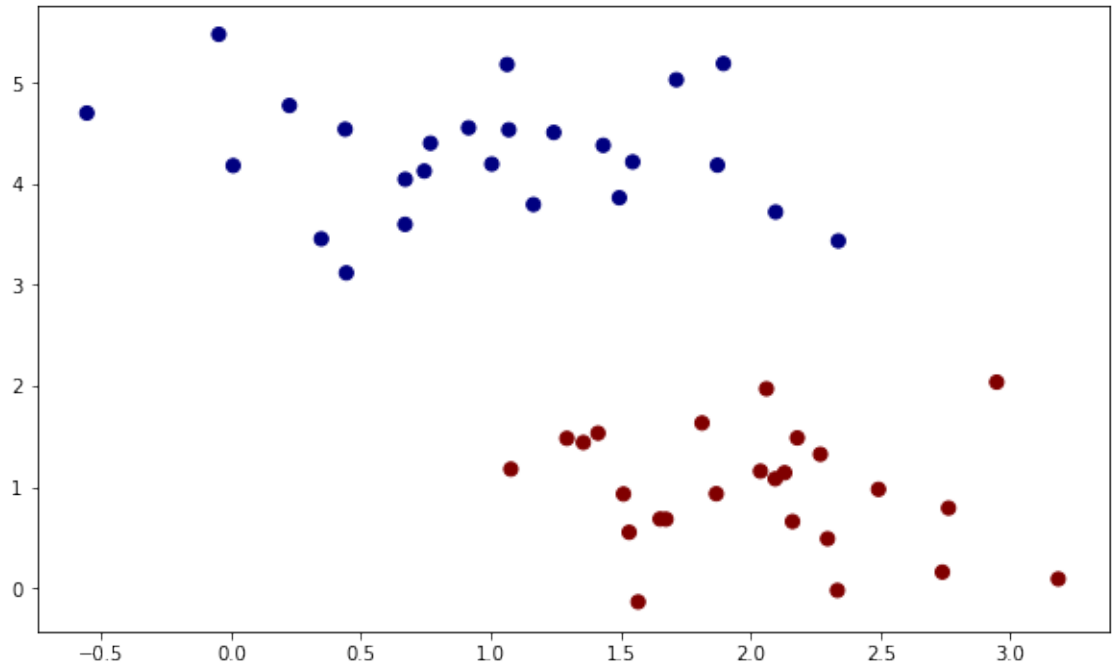
The results so far seem to be the best on this dataset. Discriminative classifiers in general are very powerful, and can account for unseen training data more than generative classifiers on average. However, the results on one particular dataset do not guarantee the same kind of performance on any problem out there. That is why it is important to try out different methods for your particular problem to find the best classifier.

3.2 Support Vector Machines

Now let us look at yet another way of choosing a linear decision boundary, which leads off in an entirely different direction, that of support vector machines (SVMs). Support vector machines (SVMs) are a particularly powerful and flexible class of supervised algorithms for both classification and regression. In this section, we will develop the intuition behind support vector machines and their use in classification problems.

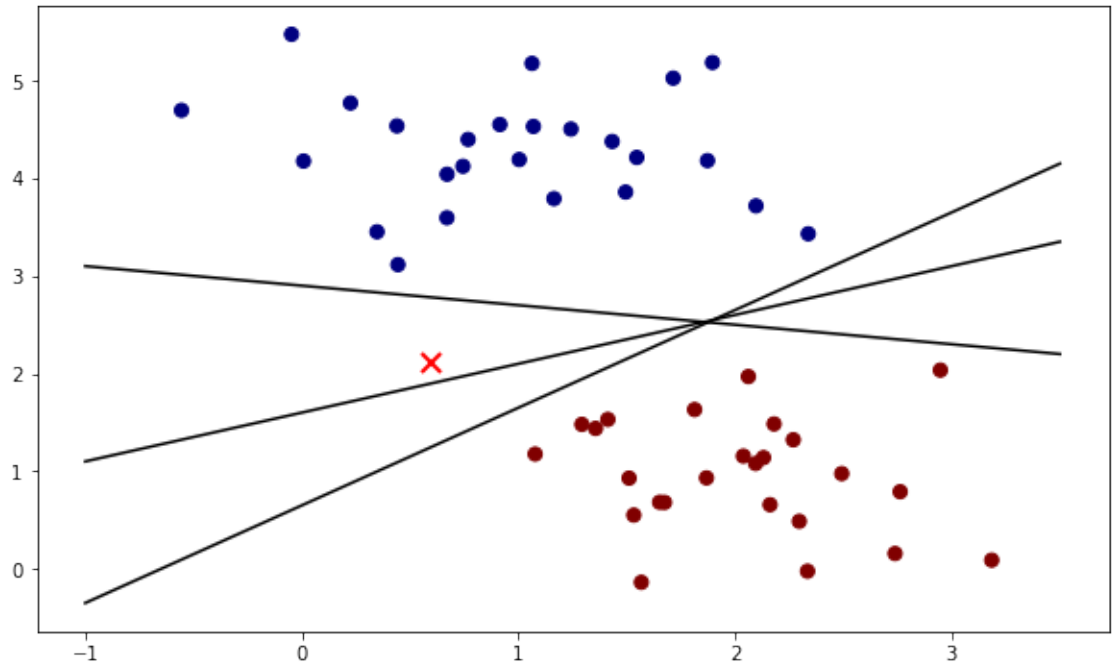
Consider finding a hyperplane that maximizes the distance of the closest point from either class in the figure below. We call this distance the margin. Points on the margin are called support vectors. Let us begin by assuming the classes are linearly separable. We want to find a line or curve (in two dimensions) or manifold (in multiple dimensions) that divides the classes from each other.More importantly, we need to find the optimally discriminating line/curve/manifold.

[12]:
```
from sklearn.datasets.samples_generator import make_blobs
X, y = make_blobs(n_samples=50, centers=2,
                  random_state=0, cluster_std=0.60)

fig, svm1 = plt.subplots(figsize=(10, 6))
svm1.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='jet');
```



A linear discriminative classifier would attempt to draw a straight line separating the two sets of data, and thereby create a model for classification. For two dimensional data like that shown here, this is a task we could do by hand. But immediately we see a problem: there is more than one possible dividing line that can perfectly discriminate between the two classes. We can draw them as follows:

[13]:
```
xfit = np.linspace(-1, 3.5)
fig, svm2 = plt.subplots(figsize=(10, 6))
svm2.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='jet')
svm2.plot([0.6], [2.1], 'x', color='red', markeredgewidth=2, markersize=10)

for m, b in [(1, 0.65), (0.5, 1.6), (-0.2, 2.9)]:
    svm2.plot(xfit, m * xfit + b, '-k')

plt.show()
```

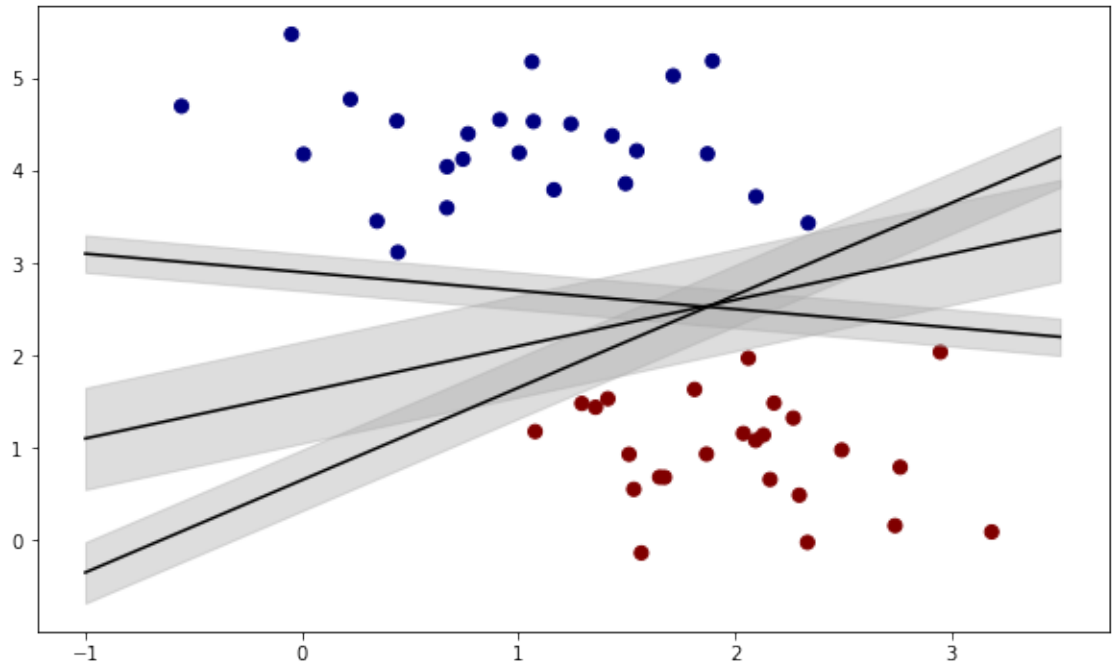


These are three very different separators which, nevertheless, perfectly discriminate between these samples. Depending on which you choose, a new data point (e.g., the one marked by the "X" in this plot) will be assigned a different label! Evidently our simple intuition of "drawing a line between classes" is not enough, and we need to think a bit deeper.

3.2.1 Maximizing the Margin

Support vector machines offer one way to improve on this. The intuition is this: rather than simply drawing a zero-width line between the classes, we can draw around each line a margin of some width, up to the nearest point. Here is an example of how this might look:

[14]:
```
xfit = np.linspace(-1, 3.5)
fig, svm3 = plt.subplots(figsize=(10, 6))
svm3.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='jet')

for m, b, d in [(1, 0.65, 0.33), (0.5, 1.6, 0.55), (-0.2, 2.9, 0.2)]:
    yfit = m * xfit + b
    svm3.plot(xfit, yfit, '-k')
    svm3.fill_between(xfit, yfit - d, yfit + d, edgecolor='none',
                      color='#AAAAAA', alpha=0.4)
```



In support vector machines, the line that maximizes this margin is the one we will choose as the optimal model. Support vector machines are an example of such a maximum margin estimator.

3.2.2 Fitting a SVM

Let's see the result of an actual fit to this data: we will use Scikit-Learn's support vector classifier to train an SVM model on this data. For the time being, we will use a linear kernel and set the C parameter to a very large number (we will discuss the meaning of these in more depth momentarily).

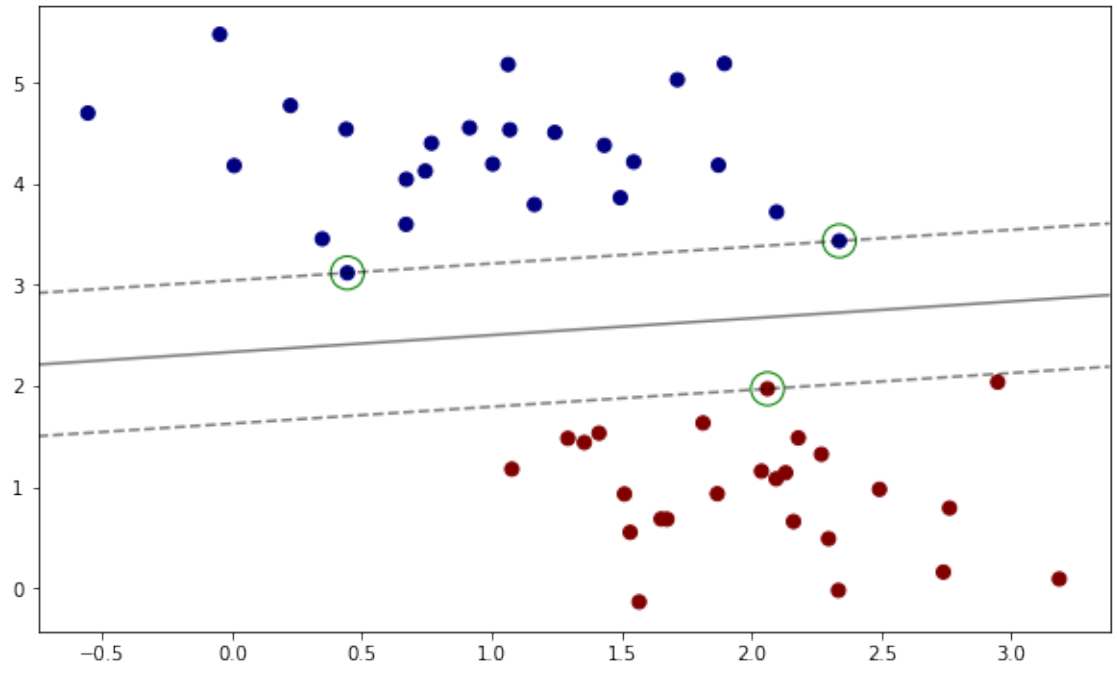
[15]:
```
from sklearn.svm import SVC # "Support vector classifier"
model = SVC(kernel='linear', C=1E10)
model.fit(X, y)
```

[15]:
```
SVC(C=10000000000.0, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='auto_deprecated',
    kernel='linear', max_iter=-1, probability=False, random_state=None,
    shrinking=True, tol=0.001, verbose=False)
```

To better visualize what's happening here, let's create a quick convenience function that will plot SVM decision boundaries for us:

[16]:
```
def plot_svc_decision_function(model, ax=None, plot_support=True):
    """Plot the decision function for a 2D SVC"""
    if ax is None:
        ax = plt.gca()
    xlim = ax.get_xlim()
```

```
    ylim = ax.get_ylim()

    # create grid to evaluate model
    x = np.linspace(xlim[0], xlim[1], 30)
    y = np.linspace(ylim[0], ylim[1], 30)
    Y, X = np.meshgrid(y, x)
    xy = np.vstack([X.ravel(), Y.ravel()]).T
    P = model.decision_function(xy).reshape(X.shape)

    # plot decision boundary and margins
    ax.contour(X, Y, P, colors='k',
               levels=[-1, 0, 1], alpha=0.5,
               linestyles=['--', '-', '--'])

    # plot support vectors
    if plot_support:
        ax.scatter(model.support_vectors_[:, 0],
                   model.support_vectors_[:, 1],
                   s=300, linewidth=1, facecolors='none', edgecolors='g');
    ax.set_xlim(xlim)
    ax.set_ylim(ylim)

xfit = np.linspace(-1, 3.5)
fig, svm3 = plt.subplots(figsize=(10, 6))
svm3.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='jet')
plot_svc_decision_function(model,ax=svm3);
```



This is the dividing line that maximizes the margin between the two sets of points. Notice that a few of the training points just touch the margin: they are indicated by the green circles in this figure. These points are the pivotal elements of this fit, and are known as the support vectors, and give the algorithm its name. In Scikit-Learn, the identity of these points are stored in the 'support_vectors_' attribute of the classifier:

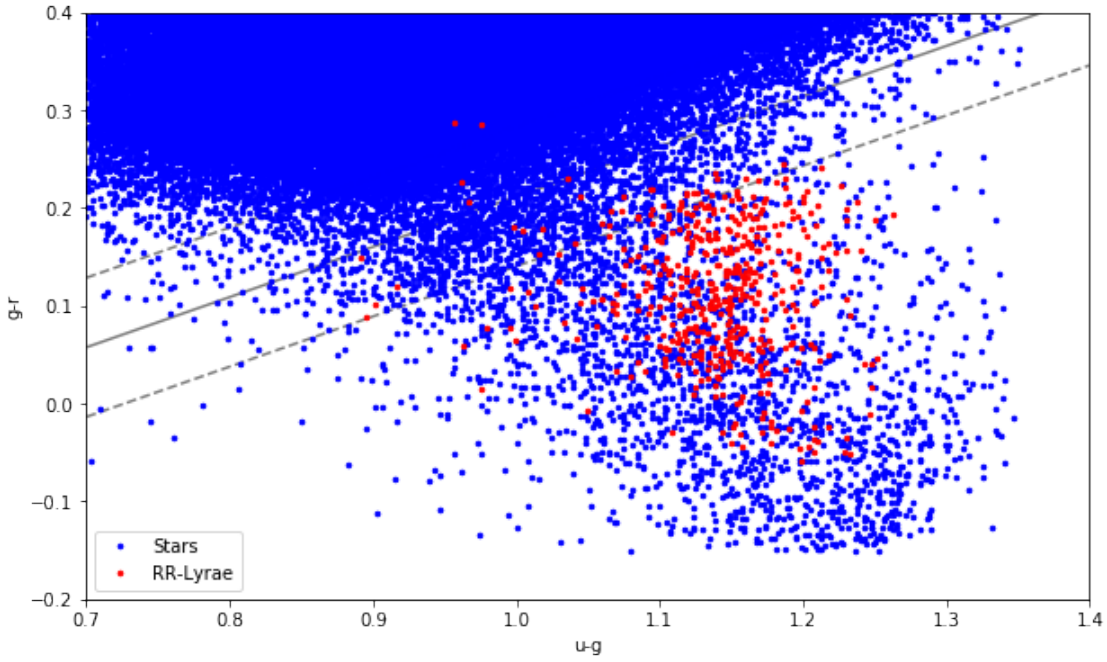
[17]:
```
model.support_vectors_
```

[17]:
```
array([[0.44359863, 3.11530945],
       [2.33812285, 3.43116792],
       [2.06156753, 1.96918596]])
```

A key to this classifier's success is that for the fit, only the position of the support vectors matter; any points further from the margin which are on the correct side do not modify the fit! Technically, this is because these points do not contribute to the loss function used to fit the model, so their position and number do not matter so long as they do not cross the margin.

Let us now see how this linear SVM classifier performs on our Star vs RR-Lyrae classification problem.

[18]:
```
# Load up RRLyrae Dataset
data = np.load('./data/rrlyrae.npz')
samples = data['data']
labels = data['labels']

# Fit SVC to the first 2 dimensions
samples_2d = samples[:,0:2]

# fit an SVM classifier, and get predictions too
from sklearn.svm import SVC # "Support vector classifier"
svc_model = SVC(kernel='linear', class_weight='balanced', C=100)
svc_model.fit(samples_2d, labels)
svc_pred = svc_model.predict(samples_2d)

fig, svm4 = plt.subplots(figsize=(10, 6))
xlim = (0.7, 1.4)
ylim = (-0.2, 0.4)
svm4.plot(samples[stars, 0], samples[stars, 1], '.', ms=5, c='b', label='Stars')
svm4.plot(samples[rrlyrae, 0], samples[rrlyrae, 1], '.', ms=5, c='r',␣
 ↪label='RR-Lyrae')
svm4.legend(loc=3)
svm4.set_xlabel('u-g')
svm4.set_ylabel('g-r')
svm4.set_xlim(xlim)
svm4.set_ylim(ylim)
plot_svc_decision_function(svc_model,ax=svm4,plot_support=False);
```



The classification is not perfect as when we had clear separation between data points. It is best to evaluate this classifier numerically again, with completeness and contamination factors. We will also include all 4 dimensions of our dataset for a full result.

[19]:
```
# Load up RRLyrae Dataset
data = np.load('./data/rrlyrae.npz')
samples = data['data']
labels = data['labels']

# fit an SVM classifier, and get predictions too
from sklearn.svm import SVC # "Support vector classifier"
svc_model = SVC(kernel='linear', class_weight='balanced', C=100)
svc_model.fit(samples, labels)
svc_pred = svc_model.predict(samples)

completeness_score = recall_score(labels,svc_pred)
contamination_score = (1-precision_score(labels,svc_pred))
print('SVC')
print('Completeness: %f'%completeness_score)
print('Contamination: %f'%contamination_score)
```

```
SVC
Completeness: 0.991718
Contamination: 0.870365
```

The results are the best so far on this dataset. We get a very high completeness compared to the other methods discussed above: it is not swayed by the fact that the background sources outnumber

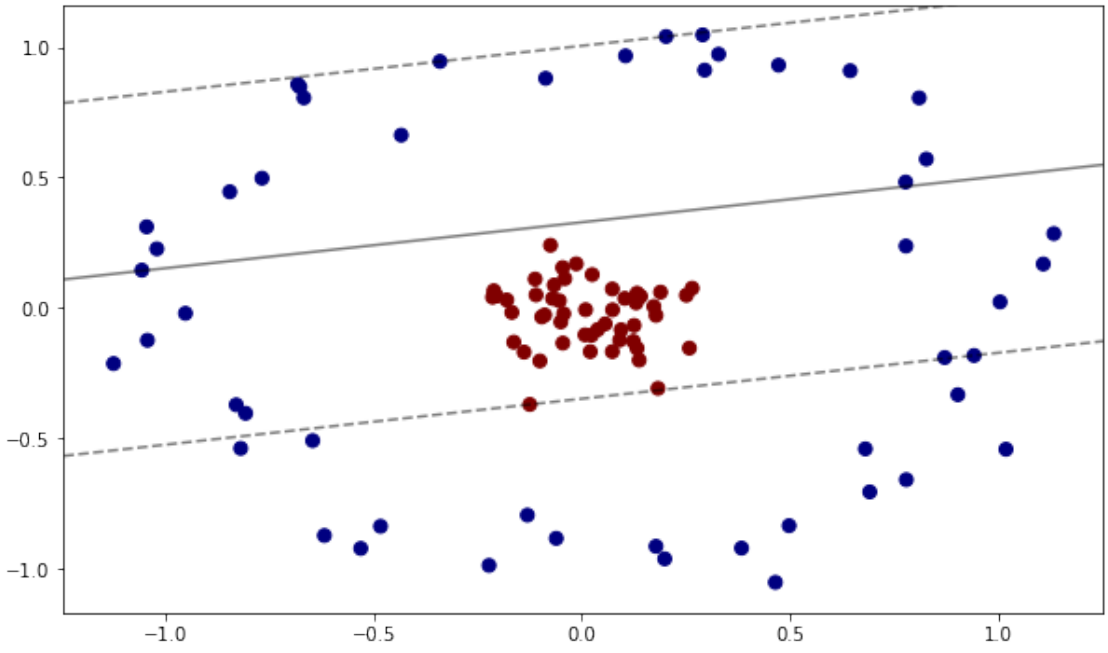
the RR Lyrae stars by a factor of 200 to 1: it simply determines the best boundary between the small RR Lyrae clump and the large background clump. This completeness, however, comes at the cost of a relatively large contamination level.

SVMs however, can be a lot more powerful than the linear SVM we have considered so far, if we go beyond the restriction of finding a linear boundary between classes.

3.2.3 Non-Linear (Kernel) SVMs

Where SVM becomes extremely powerful is when it is combined with kernels. The idea of a kernel is a function that projects our data into a higher-dimensional space defined by polynomials and Gaussian basis functions, such that within that higher-dimensional space, there will be a linear separation between the classes - thereby able to fit for nonlinear relationships with a linear classifier. To understand this concept let us look at some data that is not linearly separable:

[20]:
```
from sklearn.datasets.samples_generator import make_circles
X, y = make_circles(100, factor=.1, noise=.1)

clf = SVC(kernel='linear').fit(X, y)

fig, svm5 = plt.subplots(figsize=(10, 6))
svm5.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='jet')
plot_svc_decision_function(clf,ax=svm5,plot_support=False);
```



It is clear that no linear discrimination will ever be able to separate this data. How might we project the data into a higher dimension such that a linear separator would be sufficient? For example, one simple projection we could use would be to compute a radial basis function centered

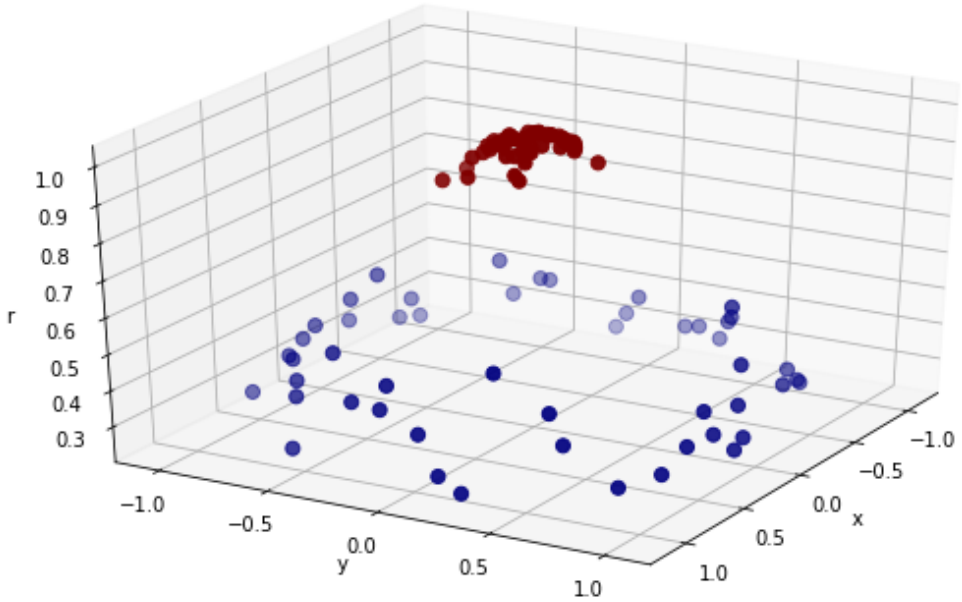
on the middle clump.

A radial basis function (RBF) is a real-valued function whose value depends only on the distance from the origin.

[21]:
```
r = np.exp(-(X ** 2).sum(1))
```

We can visualize this extra data dimension using a three-dimensional plot:

[22]:
```
from mpl_toolkits import mplot3d

def plot_3D(X=X, y=y, ax=None):
    if ax is None:
        ax = plt.gca()

    ax = plt.subplot(projection='3d')
    ax.scatter3D(X[:, 0], X[:, 1], r, c=y, s=50, cmap='jet')
    ax.view_init(elev=30, azim=30)
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_zlabel('r')

fig, svm6 = plt.subplots(figsize=(10, 6))
plot_3D(X=X, y=y, ax=svm6);
```



We can see that with this additional dimension, the data becomes trivially linearly separable, by drawing a separating plane at, say, r=0.7. Here we had to choose and carefully tune our projection:

if we had not centered our radial basis function in the right location, we would not have seen such clean, linearly separable results. In general, the need to make such a choice is a problem: we would like to somehow automatically find the best basis functions to use.

One strategy to this end is to compute a basis function centered at every point in the dataset, and let the SVM algorithm sift through the results. This type of basis function transformation is known as a kernel transformation, as it is based on a similarity relationship (or kernel) between each pair of points.

A potential problem with this strategy—projecting N points into N dimensions is that it might become very computationally intensive as N grows large. However, because of a neat little procedure known as the kernel trick, a fit on kernel-transformed data can be done implicitly — that is, without ever building the full N-dimensional representation of the kernel projection! This kernel trick is built into the SVM, and is one of the reasons the method is so powerful. We can work in the implicit feature space without ever computing the coordinates of the data in that space, but rather by simply computing the inner products between all pairs of samples in the feature space. This operation is often computationally cheaper than the explicit computation of the coordinates. Kernel functions have been introduced for sequence data, graphs, text, images, as well as vectors.

In this simple formulation, one simply replaces each occurrence of (x_i, x_i) with a kernel function $K(x_i, x_i)$ with certain properties which allow one to think of the SVM as operating in a higher-dimensional space. One such kernel is the Gaussian kernel:

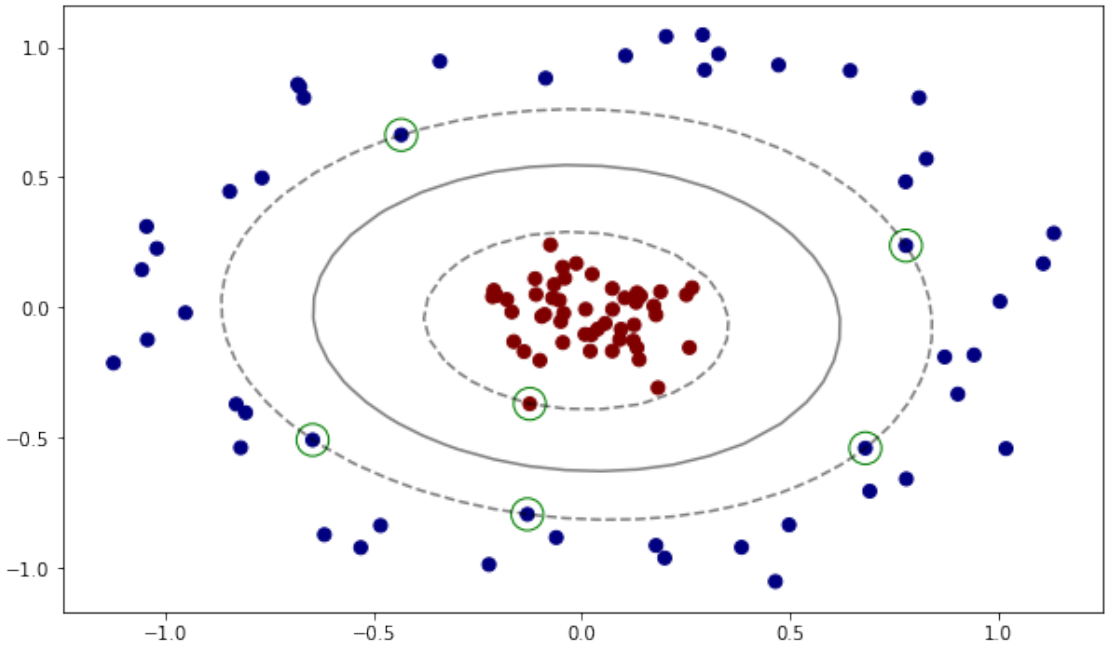
$$K(x_i, x_i) = e^{-\gamma||x_i - x_{i'}||^2} \tag{10}$$

where γ is a parameter to be learned via validation testing. In Scikit-Learn, we can apply kernelized SVM simply by changing our linear kernel to an RBF (radial basis function) kernel, using the kernel model hyperparameter:

[23]:
```
clf = SVC(kernel='rbf', C=1E6, gamma='auto')
clf.fit(X, y)
```

[23]:
```
SVC(C=1000000.0, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='auto', kernel='rbf',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
```

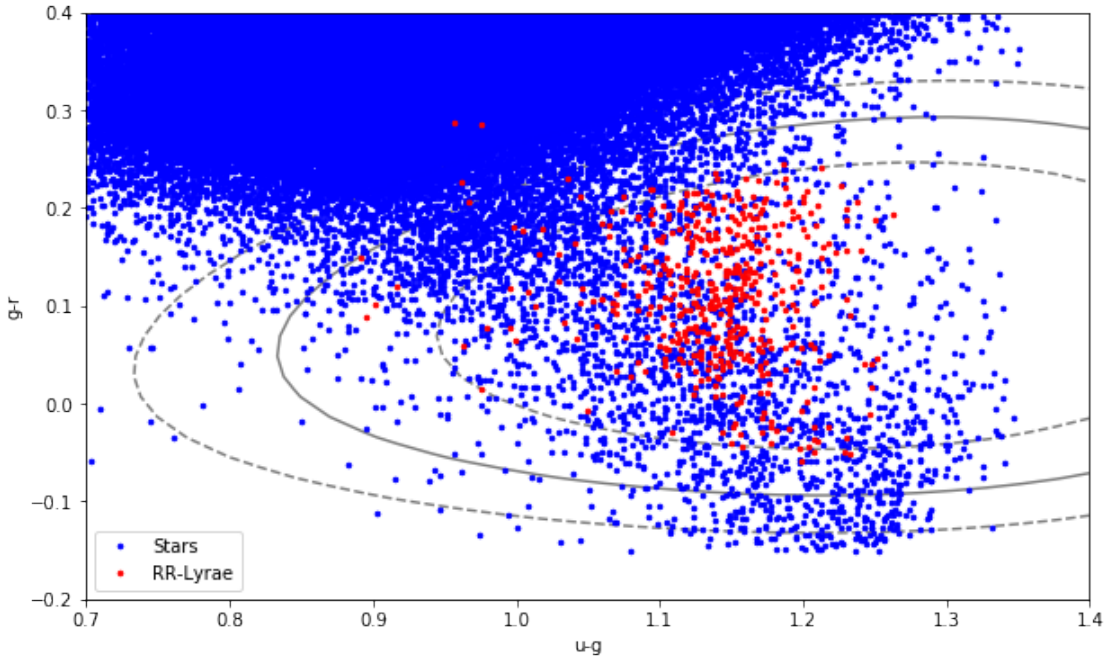
[24]:
```
fig, svm7 = plt.subplots(figsize=(10, 6))
svm7.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='jet')
plot_svc_decision_function(clf,ax=svm7)
```



Using this kernelized support vector machine, we learn a suitable nonlinear decision boundary. This kernel transformation strategy is used often in machine learning to turn fast linear methods into fast nonlinear methods, especially for models in which the kernel trick can be used. The next step, as usual, is to try out this method onto our exampe dataset:

[25]:
```
# Load up RRLyrae Dataset
data = np.load('./data/rrlyrae.npz')
samples = data['data']
labels = data['labels']

# Fit SVC to the first 2 dimensions
samples_2d = samples[:,0:2]

# fit an SVM classifier, and get predictions too
from sklearn.svm import SVC # "Support vector classifier"
svc_model = SVC(kernel='rbf', class_weight='balanced', C=100, gamma='auto')
svc_model.fit(samples_2d, labels)
svc_pred = svc_model.predict(samples_2d)

fig, svm8 = plt.subplots(figsize=(10, 6))
xlim = (0.7, 1.4)
ylim = (-0.2, 0.4)
svm8.plot(samples[stars, 0], samples[stars, 1], '.', ms=5, c='b', label='Stars')
svm8.plot(samples[rrlyrae, 0], samples[rrlyrae, 1], '.', ms=5, c='r',␣
 ↪label='RR-Lyrae')
svm8.legend(loc=3)
```

```
svm8.set_xlabel('u-g')
svm8.set_ylabel('g-r')
svm8.set_xlim(xlim)
svm8.set_ylim(ylim)
plot_svc_decision_function(svc_model,ax=svm8,plot_support=False);
```



And in order to assess the performance of the classifier, we extract completness and contamination scores:

[26]:
```
# Load up RRLyrae Dataset
data = np.load('./data/rrlyrae.npz')
samples = data['data']
labels = data['labels']

# fit an SVM classifier, and get predictions too
from sklearn.svm import SVC # "Support vector classifier"
svc_model = SVC(kernel='rbf', class_weight='balanced', C=100, gamma='auto')
svc_model.fit(samples, labels)
svc_pred = svc_model.predict(samples)

completeness_score = recall_score(labels,svc_pred)
contamination_score = (1-precision_score(labels,svc_pred))
print('SVC - RBF')
print('Completeness: %f'%completeness_score)
print('Contamination: %f'%contamination_score)
```

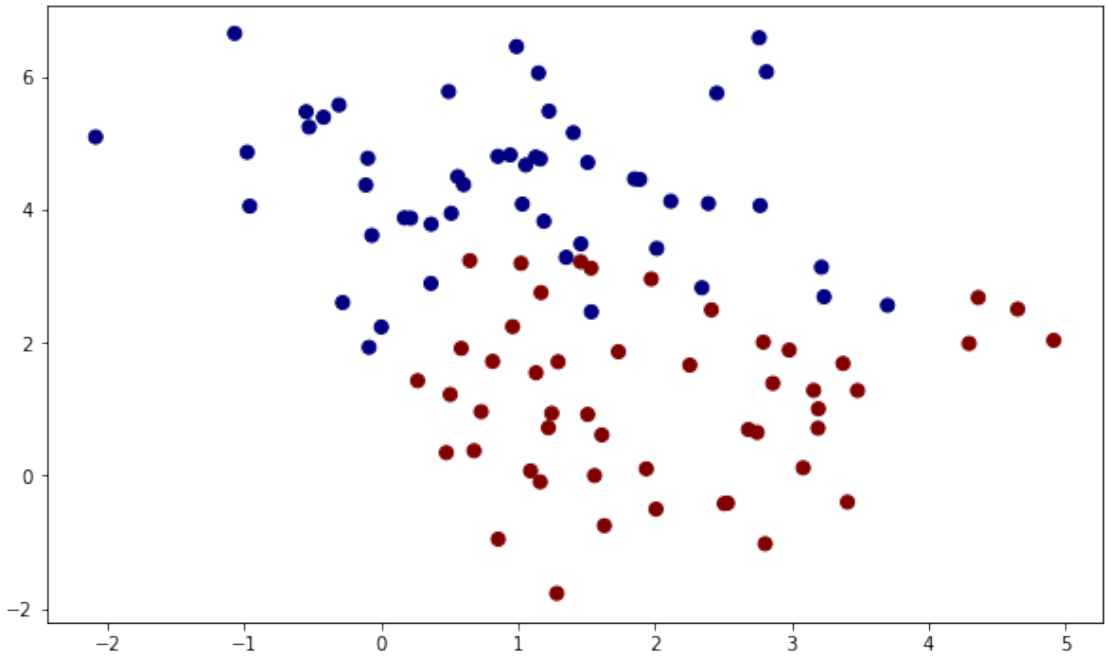
```
SVC - RBF
Completeness: 0.991718
Contamination: 0.850825
```

Again we get a high completeness compared to the other methods discussed, with a slight decrease in contamination compared to a linear SVM. This nonlinear classification improves over the linear version only slightly. For this particular data set, the contamination is not driven by nonlinear effects.

3.2.4 Tuning the SVM - Softening Margins

Our discussion thus far has centered around very clean datasets for learning how an SVM works (and trying the variants out on our sample star dataset), in which a perfect decision boundary exists. But what if your data has some amount of overlap, as is in our start dataset? Let us first look at a simpler overlapping dataset than our stars example:

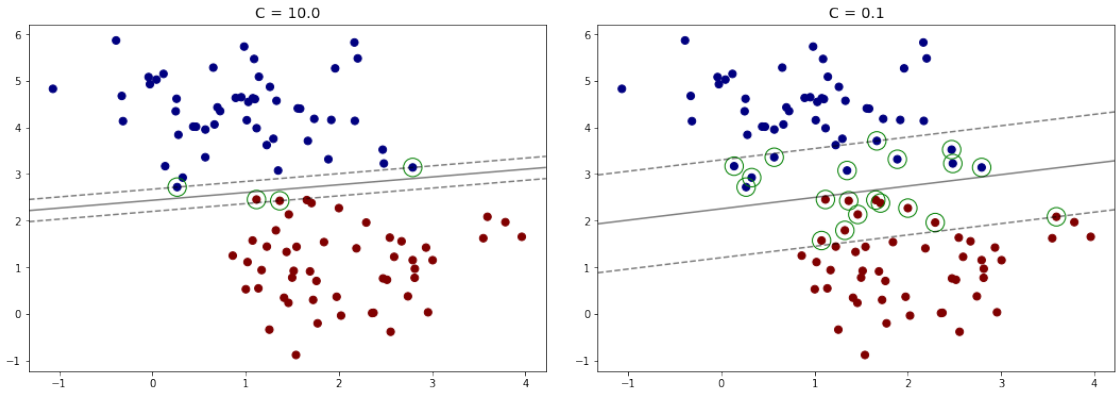
[27]:
```
fig, svm9 = plt.subplots(figsize=(10, 6))
X, y = make_blobs(n_samples=100, centers=2,
                  random_state=0, cluster_std=1.2)
svm9.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='jet');
```



To handle this case, the SVM implementation has a bit of a fudge-factor which "softens" the margin: that is, it allows some of the points to creep into the margin if that allows a better fit. The hardness of the margin is controlled by a tuning parameter, most often known as C. For very large C, the margin is hard, and points cannot lie in it. For smaller C, the margin is softer, and can grow to encompass some points.

The plot shown below gives a visual picture of how a changing C parameter affects the final fit, via the softening of the margin:

[28]:
```
X, y = make_blobs(n_samples=100, centers=2,
                  random_state=0, cluster_std=0.8)

fig, ax = plt.subplots(1, 2, figsize=(16, 6))
fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)

for axi, C in zip(ax, [10.0, 0.1]):
    model = SVC(kernel='linear', C=C).fit(X, y)
    axi.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='jet')
    plot_svc_decision_function(model, axi)
    axi.scatter(model.support_vectors_[:, 0],
                model.support_vectors_[:, 1],
                s=300, lw=1, facecolors='none');
    axi.set_title('C = {0:.1f}'.format(C), size=14)
```



The optimal value of the C parameter will depend on your dataset, and should be tuned using cross-validation or a similar procedure. Let us try and assess the classifier performance on our dataset based on different values of C:

[29]:
```
from sklearn.svm import SVC # "Support vector classifier"

# Load up RRLyrae Dataset
data = np.load('./data/rrlyrae.npz')
samples = data['data']
labels = data['labels']

for c_value in [1,10,100]:
    # fit an SVM classifier, and get predictions too
    svc_model = SVC(kernel='rbf', class_weight='balanced', C=c_value,␣
 ↪gamma='auto')
    svc_model.fit(samples, labels)
```

```
    svc_pred = svc_model.predict(samples)

    completeness_score = recall_score(labels,svc_pred)
    contamination_score = (1-precision_score(labels,svc_pred))
    print('SVC - RBF: C={0:.1f}'.format(c_value))
    print('Completeness: %f'%completeness_score)
    print('Contamination: %f'%contamination_score)
```

```
SVC - RBF: C=1.0
Completeness: 0.991718
Contamination: 0.871959
SVC - RBF: C=10.0
Completeness: 0.991718
Contamination: 0.862711
SVC - RBF: C=100.0
Completeness: 0.991718
Contamination: 0.850825
```

3.2.5 Overfitting

You can observe that as C is getting higher, the contamination is decreased. This is a good thing - but be careful not to be too strict, as it may have the adverse effect of learning an SVM classifier that is too intrinsically tied to the data used for training - whilst not generalizing well for future data. This phenomenon is called overfitting. One way to test for overfitting for SVM classifiers, and indeed for all the classifiers we have seen so far (and all other classifiers you will work on), is to split your data into two chunks. The first chunk will be called your training set whilst the second chunk will be called your validation set. The validation set will therefore contain data that the classifier never 'saw' during training, and will therefore be very similar to what the classifier might encounter when being utilised for inference. If you notices that your training completeness/contamination scores are very good for your training set but get visibly wors during verification, then you can expect the same bad results to occur during inference. This is usually a sign of ovefitting.

3.2.6 SVM Summary

We have seen here a brief intuitive introduction to the principals behind support vector machines. These methods are a powerful classification method for a number of reasons:

- Their dependence on relatively few support vectors means that they are very compact models, and take up very little memory.
- Once the model is trained, the prediction phase is very fast.
- Because they are affected only by points near the margin, they work well with high-dimensional data - even data with more dimensions than samples, which is a challenging regime for other algorithms.
- Their integration with kernel methods makes them very versatile, able to adapt to many types of data.

However, SVMs have several disadvantages as well:

- The scaling with the number of samples N is $\mathcal{O}(N^3)$ at worst, or $\mathcal{O}(N^2)$ for efficient implementations. For large numbers of training samples, this computational cost can be prohibitive.
- The results are strongly dependent on a suitable choice for the softening parameter C. This must be carefully chosen via cross-validation, which can be expensive as datasets grow in size.
- The results do not have a direct probabilistic interpretation. This can be estimated via an internal cross-validation (see the probability parameter of SVC), but this extra estimation is costly.

With those traits in mind, you should generally only turn to SVMs once other simpler, faster, and less tuning-intensive methods have been shown to be insufficient for your needs. Nevertheless, if you have the CPU cycles to commit to training and cross-validating an SVM on your data, the method can lead to excellent results.

3.3 Decision Trees and Random Forests

Previously we have looked in depth at a simple generative classifier (naive Bayes) and a powerful discriminative classifier (support vector machines). Here we'll take a look at motivating another powerful algorithm — a non-parametric algorithm called random forests. Random forests are an example of an ensemble method, meaning that it relies on aggregating the results of an ensemble of simpler estimators. The somewhat surprising result with such ensemble methods is that the sum can be greater than the parts: that is, a majority vote among a number of estimators can end up being better than any of the individual estimators doing the voting! We will see examples of this in the following sections.

3.3.1 Motivation for Random Forests: Decision Trees

Random forests are an example of an ensemble learner built on decision trees. For this reason we shall start by discussing decision trees themselves.

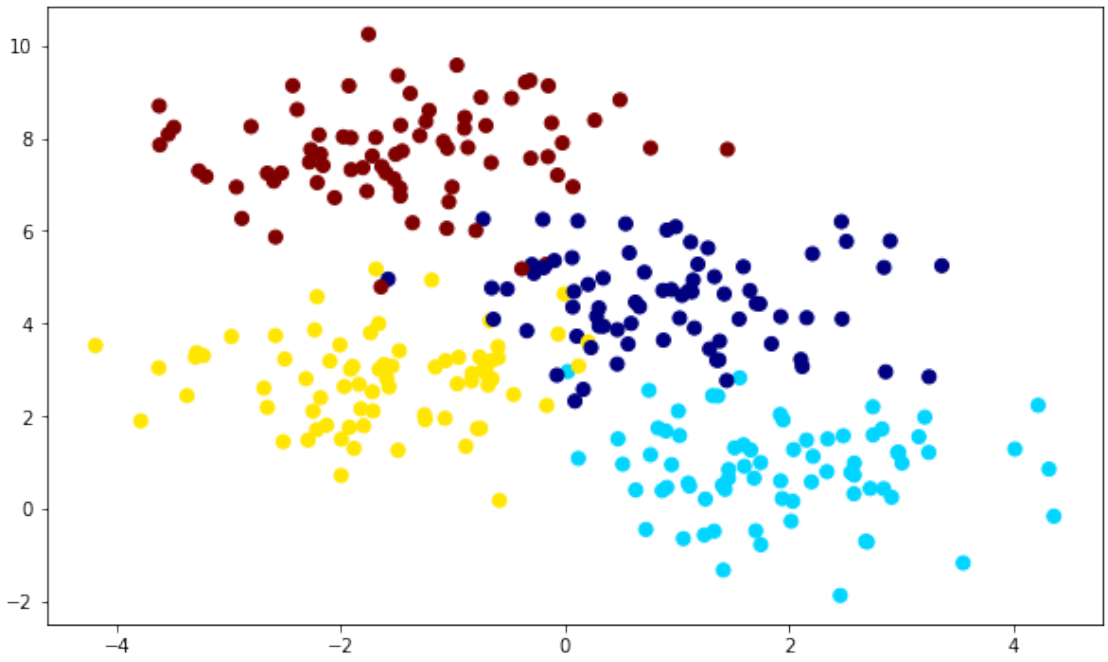
Decision trees are extremely intuitive ways to classify or label objects: you simply ask a series of questions designed to zero-in on the classification. For example, if you wanted to build a decision tree to classify an animal you come across while on a hike, you might construct the one shown here:

The binary splitting makes this extremely efficient: in a well-constructed tree, each question will cut the number of options by approximately half, very quickly narrowing the options even among a large number of classes. The trick, of course, comes in deciding which questions to ask at each step. In machine learning implementations of decision trees, the questions generally take the form of axis-aligned splits in the data: that is, each node in the tree splits the data into two groups using a cutoff value within one of the features. Let's now look at an example of this.

3.3.2 Your First Decision Tree

Consider the following two-dimensional data, which has one of four class labels:

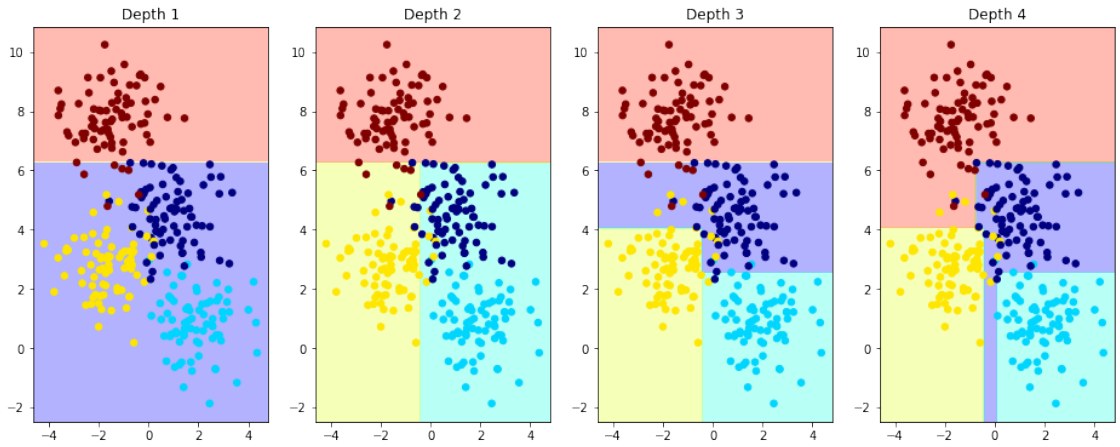
[30]:
```
from sklearn.datasets import make_blobs

X, y = make_blobs(n_samples=300, centers=4,
                  random_state=0, cluster_std=1.0)
fig, dt1 = plt.subplots(figsize=(10, 6))
dt1.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='jet');
```



A simple decision tree built on this data will iteratively split the data along one or the other axis according to some quantitative criterion, and at each level assign the label of the new region according to a majority vote of points within it. Let us look into how a decision tree may split this data at various iteration steps:

[31]:
```
from sklearn.tree import DecisionTreeClassifier

# A nice utility function that will help us visualize a Decision Tree classifier
def visualize_classifier(model, X, y, ax=None, cmap='jet'):
    ax = ax or plt.gca()

    # Plot the training points
    ax.scatter(X[:, 0], X[:, 1], c=y, s=30, cmap=cmap,
               clim=(y.min(), y.max()), zorder=3)
    ax.axis('tight')
    # ax.axis('off')
    xlim = ax.get_xlim()
    ylim = ax.get_ylim()

    # fit the estimator
    model.fit(X, y)
    xx, yy = np.meshgrid(np.linspace(*xlim, num=200),
                         np.linspace(*ylim, num=200))
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)

    # Create a color plot with the results
```

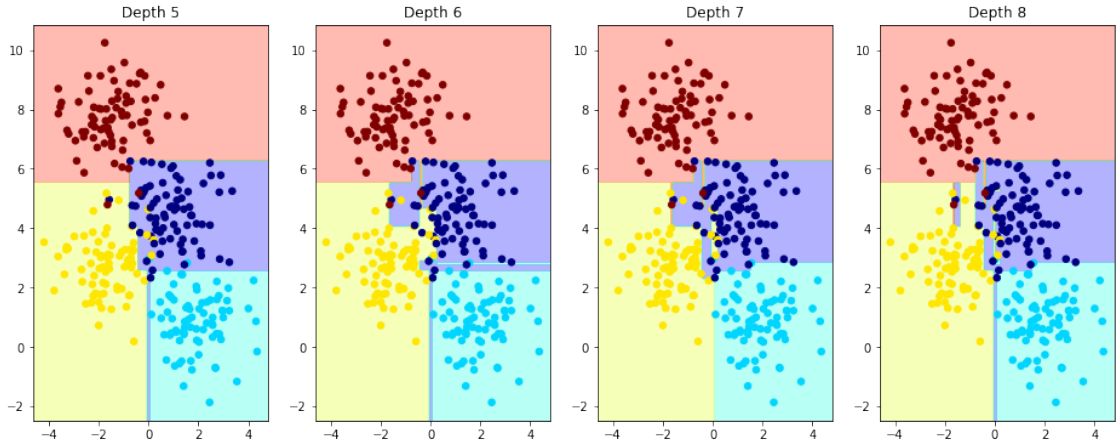
```
    n_classes = len(np.unique(y))
    contours = ax.contourf(xx, yy, Z, alpha=0.3,
                           levels=np.arange(n_classes + 1) - 0.5,
                           cmap=cmap,
#                            cmap=cmap, clim=(y.min(), y.max()),
                           zorder=1)

    ax.set(xlim=xlim, ylim=ylim)


f, axes_depths = plt.subplots(1, 4, figsize=(16, 6))
for depth in [1,2,3,4]:
    tree = DecisionTreeClassifier(max_depth=depth)
    visualize_classifier(model=tree,X=X,y=y,ax=axes_depths[depth-1])
    axes_depths[depth-1].set_title('Depth {0:d}'.format(depth))

plt.show()
```



Notice that after the first split, every point in the upper branch remains unchanged, so there is no need to further subdivide this branch. Except for nodes that contain all of one color, at each level every region is again split along one of the two features.

Now what if we keep splitting further (increasing the tree depth)? Let's have a look:

[32]:
```
f, axes_depths2 = plt.subplots(1, 4, figsize=(16, 6))
for depth in [1,2,3,4]:
    tree = DecisionTreeClassifier(max_depth=depth+4)
    visualize_classifier(model=tree,X=X,y=y,ax=axes_depths2[depth-1])
    axes_depths2[depth-1].set_title('Depth {0:d}'.format(depth+4))

plt.show()
```

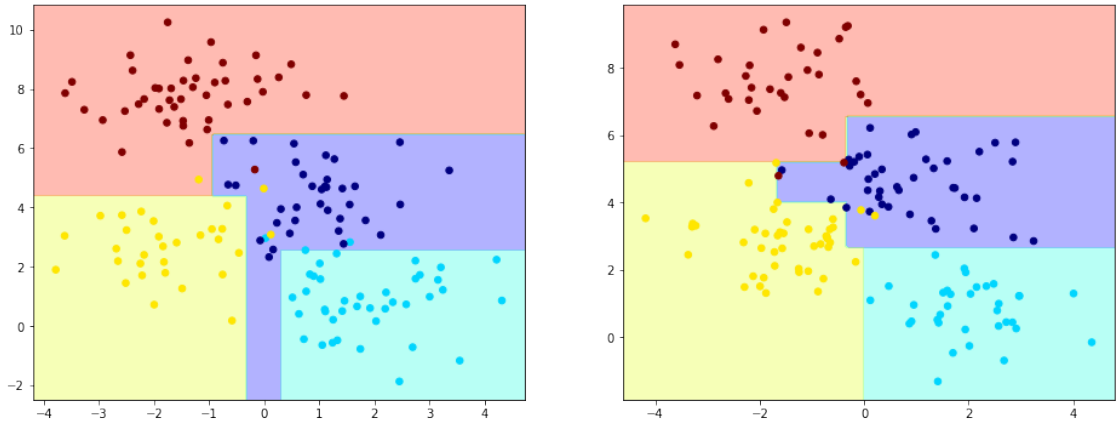


Notice that as the depth increases, we tend to get very strangely shaped classification regions; for example, at a depth of five, there is a tall and skinny purple region between the yellow and blue regions. It is clear that this is less a result of the true, intrinsic data distribution, and more a result of the particular sampling or noise properties of the data. That is, this decision tree, even at only five levels deep, is clearly over-fitting our data.

3.3.3 Decision Trees and Overfitting

Such over-fitting turns out to be a general property of decision trees: it is very easy to go too deep in the tree, and thus to fit details of the particular data rather than the overall properties of the distributions they are drawn from. Another way to see this over-fitting is to look at models trained on different subsets of the data—for example, in this figure we train two different trees, each on half of the original data:

[33]:
```
f, dt2 = plt.subplots(1, 2, figsize=(16, 6))
tree = DecisionTreeClassifier(max_depth=4)
visualize_classifier(model=tree, X=X[::2], y=y[::2], ax=dt2[0])
visualize_classifier(model=tree, X=X[1::2], y=y[1::2], ax=dt2[1])
plt.show()
```



It is clear that in some places, the two trees produce consistent results (e.g., in the four corners), while in other places, the two trees give very different classifications (e.g., in the regions between any two clusters). The key observation is that the inconsistencies tend to happen where the classification is less certain, and thus by using information from both of these trees, we might come up with a better result.

Now that we have an intuition of decision trees, let us try out a decision tree classifier on our stars dataset:

[34]:
```
# Load up RRLyrae Dataset
data = np.load('./data/rrlyrae.npz')
samples = data['data']
labels = data['labels']

for depth in [1,2,3,4,5]:
    tree = DecisionTreeClassifier(max_depth=depth,class_weight='balanced').
 ↪fit(samples,labels)
    tree_pred = tree.predict(samples)

    completeness_score = recall_score(labels,tree_pred)
    contamination_score = (1-precision_score(labels,tree_pred))
    print('Decision Tree: Depth {0:d}'.format(depth))
    print('Completeness: %f'%completeness_score)
    print('Contamination: %f'%contamination_score)
```

```
Decision Tree: Depth 1
Completeness: 0.989648
Contamination: 0.906458
Decision Tree: Depth 2
Completeness: 0.981366
Contamination: 0.851597
Decision Tree: Depth 3
Completeness: 0.995859
Contamination: 0.855120
Decision Tree: Depth 4
Completeness: 0.997930
Contamination: 0.841656
Decision Tree: Depth 5
Completeness: 1.000000
Contamination: 0.830467
```

The completeness score increases with tree depth, and the contamination is decreasing. This is a very good result on our dataset. However keep in mind that like in all other demonstrations before, we are predicting the class of data points which were seen during training. In reality this will never be the case (we only proceeded this way for brevity). But for the sake of showing how the performance of a decision tree varies when this is not so, let us re-run this experiment, this

time splitting the data into two sets: training and validation (the same technique can be used in all other experiments):

[35]:
```
from sklearn.model_selection import train_test_split

# Load up RRLyrae Dataset
data = np.load('./data/rrlyrae.npz')
samples = data['data']
labels = data['labels']

# keep 33% of the data for testing/validation
X_train, X_test, y_train, y_test = train_test_split(samples, labels,␣
 ↪test_size=0.33)

for depth in [1,2,3,4,5]:
    tree = DecisionTreeClassifier(max_depth=depth,class_weight='balanced').
 ↪fit(X_train,y_train)
    tree_pred = tree.predict(X_test)

    completeness_score = recall_score(y_test,tree_pred)
    contamination_score = (1-precision_score(y_test,tree_pred))
    print('Decision Tree: Depth {0:d}'.format(depth))
    print('Completeness: %f'%completeness_score)
    print('Contamination: %f'%contamination_score)
```

```
Decision Tree: Depth 1
Completeness: 0.985714
Contamination: 0.915181
Decision Tree: Depth 2
Completeness: 0.978571
Contamination: 0.865025
Decision Tree: Depth 3
Completeness: 0.992857
Contamination: 0.867619
Decision Tree: Depth 4
Completeness: 0.985714
Contamination: 0.848352
Decision Tree: Depth 5
Completeness: 0.978571
Contamination: 0.843070
```

The scores change slightly - primarily completeness hits a bit of a wall and does not increase with tree depth. On the other hand, contamination does decrease with tree depth.

In principle, the recursive splitting of the tree could continue until there is a single point per node. This is, however, inefficient as it results in $\mathcal{O}(N)$ computational cost for both the construction and traversal of the tree. A common criterion for stopping the recursion is, therefore, to cease splitting the nodes when either a node contains only one class of object, when a split does not improve the information gain or reduce the misclassifications, or when the number of points per node reaches a

predefined value.

As with all model fitting, as we increase the complexity of the model we run into the issue of overfitting the data. For decision trees the complexity is defined by the number of levels or depth of the tree. As the depth of the tree increases, the error on the training set will decrease. At some point, however, the tree will cease to represent the correlations within the data and will reflect the noise within the training set. We can, therefore, use some cross-validation techniques (multiple train/validation data splits) and observe the overall misclassification error to optimize the depth of the tree.

3.3.4 Ensembles of Estimators

The notion that overfitting estimators can be combined to reduce the effect of this overfitting is what underlies an ensemble method called bagging. Bagging (from bootstrap aggregation) averages the predictive results of a series of bootstrap samples from a training set of data. Often applied to decision trees, bagging is applicable to regression and many nonlinear model fitting or classification techniques. For a sample of N points in a training set, bagging generates K equally sized bootstrap samples from which to estimate the function $f_i(x)$. The final estimator, defined by bagging, is then:

$$f(x) = \frac{1}{K} \sum_{i}^{K} f_i(x) \tag{11}$$

Bagging makes use of an ensemble (a grab bag, perhaps) of parallel estimators, each of which over-fits the data, and averages the results to find a better classification.

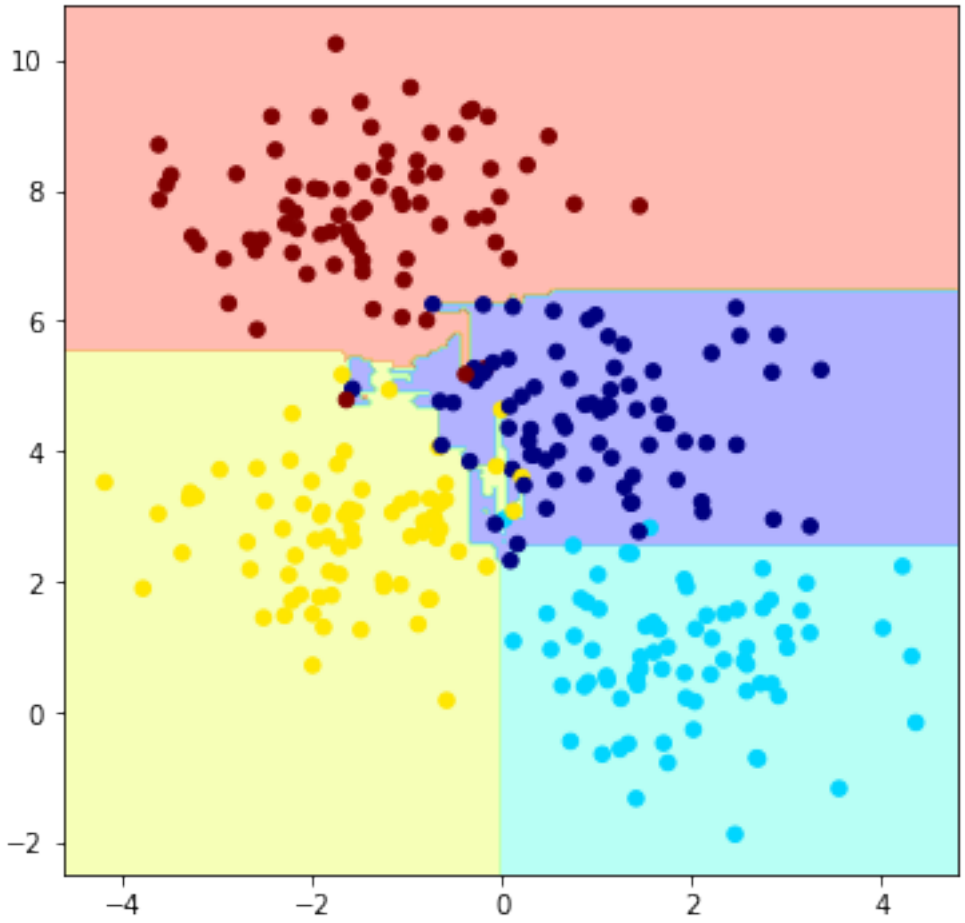
Random forests expand upon the bootstrap aspects of bagging by generating a set of decision trees from these bootstrap samples. The features on which to generate the tree are selected at random from the full set of features in the data. The final classification from the random forest is based on the averaging of the classifications of each of the individual decision trees. In so doing, random forests address two limitations of decision trees: the overfitting of the data if the trees are inherently deep, and the fact that axis-aligned partitioning of the data does not accurately reflect the potentially correlated and/or nonlinear decision boundaries that exist within data sets.

In generating a random forest we define n, the number of trees that we will generate, and m, the number of attributes that we will consider splitting on at each level of the tree. For each decision tree a subsample (bootstrap sample) of data is selected from the full data set. At each node of the tree, a set of m variables are randomly selected and the split criteria is evaluated for each of these attributes; a different set of m attributes are used for each node. The classification is derived from the mean or mode of the results from all of the trees. Keeping m small compared to the number of features controls the complexity of the model and reduces the concerns of overfitting.

We can create this type of bagging classification using Scikit-Learn's BaggingClassifier estimator, as shown here:

[36]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import BaggingClassifier
from sklearn.datasets import make_blobs

X, y = make_blobs(n_samples=300, centers=4,
```

```
                    random_state=0, cluster_std=1.0)

tree = DecisionTreeClassifier()
bag = BaggingClassifier(tree, n_estimators=100, max_samples=0.8,
                        random_state=1)

bag.fit(X, y)

f, dt3 = plt.subplots(figsize=(6, 6))
visualize_classifier(bag, X, y)
plt.show()
```

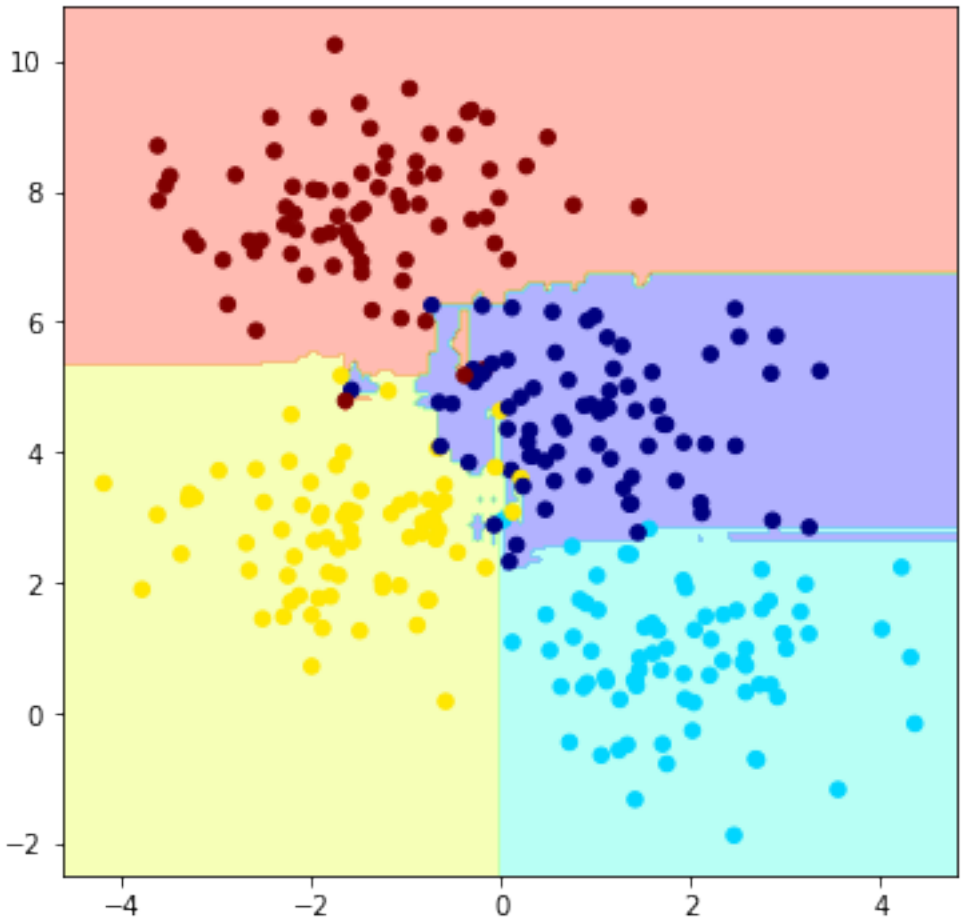


In this example, we have randomized the data by fitting each estimator with a random subset of 80% of the training points. In practice, decision trees are more effectively randomized by injecting some stochasticity in how the splits are chosen: this way all the data contributes to the fit each time, but the results of the fit still have the desired randomness. For example, when determining which feature to split on, the randomized tree might select from among the top several features. You can read more technical details about these randomization strategies in the Scikit-Learn documentation

and references within.

In Scikit-Learn, such an optimized ensemble of randomized decision trees is implemented in the RandomForestClassifier estimator, which takes care of all the randomization automatically. All you need to do is select a number of estimators, and it will very quickly (in parallel, if desired) fit the ensemble of trees:

[37]:
```
from sklearn.ensemble import RandomForestClassifier

model = RandomForestClassifier(n_estimators=100, random_state=0)

f, dt4 = plt.subplots(figsize=(6, 6))
visualize_classifier(model, X, y)
plt.show()
```



We see that by averaging over 100 random models, we end up with an overall model that is much closer to our intuition about how the parameter space should be split.

Similar to the cross-validation technique used to arrive at the optimal depth of the tree, cross-validation can also be used to determine the number of trees, n, and the number of random features

m, simply by optimizing over all free parameters. With random forests, n is typically increased until the cross-validation error plateaus, and $\sqrt{m}$ is often chosen to be K , where K is the number of attributes in the sample.

4 Classifier Evaluation - ROC Curves

Comparing the performance of classifiers is an important part of choosing the best classifier for a given task. "Best" in this case can be highly subjective: for some problems, one might wish for high completeness at the expense of contamination; at other times, one might wish to minimize contamination at the expense of completeness. One way to visualize this is to plot receiver operating characteristic (ROC) curves. An ROC curve usually shows the true-positive rate as a function of the false-positive rate as the discriminant function is varied. How the function is varied depends on the model: in the example of Gaussian naive Bayes, the curve is drawn by classifying data using relative probabilities between 0 and 1.

To conclude the classification topic and introduce ROC Curves at the same time, we shall be considering a new dataset for the classification problem of stars vs. quasars from four-colour photometry. This second dataset for photometric classification, two catalogs of quasars and stars from the SDSS Spectroscopic Catalog are used. The quasars are derived from the DR7 Quasar Catalog, while the stars are derived from the SEGUE Stellar Parameters Catalog. The combined data has approximately 100,000 quasars and 300,000 stars. We shall be looking into the u−g, g−r, r−i, and i−z colors to demonstrate photometric classification of these objects. We stress that because of the different selection functions involved in creating the two catalogs, the combined sample does not reflect a real-world sample of the objects: we use it for purposes of illustration only.

The stars outnumber the quasars by a factor of 3, meaning that a false-positive rate of 0.3 corresponds to a contamination of 50%. In the previous dataset, because there are fewer than five sources for every 1000 background objects, a false-positive rate of even 0.05 means that false positives outnumber true positives ten to one!

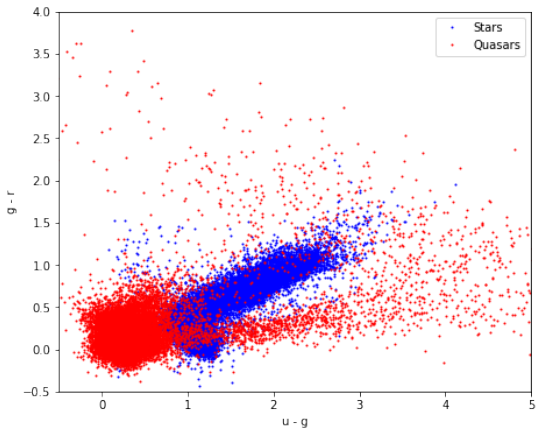
Let us look at how to generate and interpret ROC curves. Scikit-learn has some built-in tools for computing ROC curves and completeness–efficiency curves (known as precision-recall curves in the machine learning community). We shall be using most of the stuff we learnt throughout this topic in this final example - classifiers, data handling, data splits, plotting tricks, etc.:

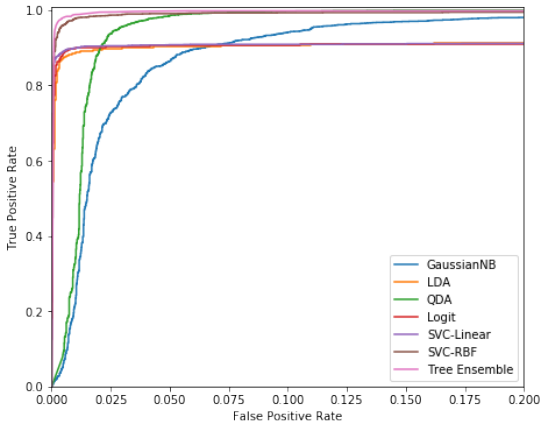
[38]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve
from sklearn.naive_bayes import GaussianNB
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis as QDA
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import BaggingClassifier
from sklearn.model_selection import train_test_split

# Load up Star vs Quasar Dataset
```

```
data = np.load('./data/dr7_quasar.npz')
quasars = data['quasars']
stars = data['stars']

# Truncate data for speed
quasars = quasars[::5]
stars = stars[::5]

# # stars are indicated by 0 labels, and quasars by 1 labels
# stars = (labels == 0)
# quasars = (labels == 1)

# stack colors into matrix X
Nqso = len(quasars)
Nstars = len(stars)
X = np.empty((Nqso + Nstars, 4), dtype=float)

X[:Nqso, 0] = quasars['mag_u'] - quasars['mag_g']
X[:Nqso, 1] = quasars['mag_g'] - quasars['mag_r']
X[:Nqso, 2] = quasars['mag_r'] - quasars['mag_i']
X[:Nqso, 3] = quasars['mag_i'] - quasars['mag_z']

X[Nqso:, 0] = stars['upsf'] - stars['gpsf']
X[Nqso:, 1] = stars['gpsf'] - stars['rpsf']
X[Nqso:, 2] = stars['rpsf'] - stars['ipsf']
X[Nqso:, 3] = stars['ipsf'] - stars['zpsf']

y = np.zeros(Nqso + Nstars, dtype=int)
y[:Nqso] = 1

# split into training and test sets
# keep 10% of the data for testing/validation
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1)


# Show sample data on left axis
f, (roc1, roc2) = plt.subplots(1, 2, figsize=(16, 6))
star_index = np.where(y_train==0)[0]
quasar_index = np.where(y_train==1)[0]
roc1.plot(X_train[star_index, 0], X_train[star_index, 1], '.', ms=2, c='b',␣
 ↪label='Stars')
roc1.plot(X_train[quasar_index, 0], X_train[quasar_index, 1], '.', ms=2, c='r',␣
 ↪label='Quasars')
roc1.set_xlim(-0.5, 5.0)
roc1.set_ylim(-0.5, 4)
roc1.set_xlabel('u - g')
roc1.set_ylabel('g - r')
```

```
roc1.legend(loc="upper right")


## Run fits for all classifiers
names = []
probs = []

# Fit the Naive Bayes classifier and store test results
gnb = GaussianNB()
gnb.fit(X_train, y_train)
y_prob = gnb.predict_proba(X_test)
names.append('GaussianNB')
probs.append(y_prob[:, 1])

# Fit the LDA classifier and store test results
lda = LDA()
lda.fit(X_train, y_train)
y_prob = lda.predict_proba(X_test)
names.append('LDA')
probs.append(y_prob[:, 1])

# Fit the QDA classifier and store test results
qda = QDA()
qda.fit(X_train, y_train)
y_prob = qda.predict_proba(X_test)
names.append('QDA')
probs.append(y_prob[:, 1])

# Fit the Logistic Regression classifier and store test results
logit = LogisticRegression(class_weight='balanced',solver='lbfgs')
logit.fit(X_train, y_train)
y_prob = logit.predict_proba(X_test)
names.append('Logit')
probs.append(y_prob[:, 1])

# SVM Training will take a while, uncomment to add to results
# Fit a linear SVM classifier and store test results
svc_model = SVC(kernel='linear', class_weight='balanced', C=10,␣
 ↪probability=True)
svc_model.fit(X_train, y_train)
y_prob = svc_model.predict_proba(X_test)
names.append('SVC-Linear')
probs.append(y_prob[:, 1])

# Fit an RBF SVM classifier and store test results
svc_model = SVC(kernel='rbf', class_weight='balanced', C=10, gamma='auto',␣
 ↪probability=True)
```

```
svc_model.fit(X_train, y_train)
y_prob = svc_model.predict_proba(X_test)
names.append('SVC-RBF')
probs.append(y_prob[:, 1])

# Fit Decision Tree ensemble classifier and store test results
tree = DecisionTreeClassifier()
bag = BaggingClassifier(tree, n_estimators=30, max_samples=0.8, random_state=1)
bag.fit(X_train, y_train)
y_prob = bag.predict_proba(X_test)
names.append('Tree Ensemble')
probs.append(y_prob[:, 1])

for i in range(0,len(names)):
    classifier = names[i]
    y_prob = probs[i]
    fpr, tpr, thresholds = roc_curve(y_test, y_prob)
    fpr = np.concatenate([[0], fpr])
    tpr = np.concatenate([[0], tpr])
    roc2.plot(fpr, tpr, label=classifier)

roc2.legend(loc=4)
roc2.set_xlabel('False Positive Rate')
roc2.set_ylabel('True Positive Rate')
roc2.set_xlim(0, 0.2)
roc2.set_ylim(0, 1.01)

plt.show()
```





Interpreting an ROC curve is easy. We are aiming to maximize the true positive rate, and to minimize the false positive rate. That means that the curves closest to the upper left of the plot are the best classifiers. In our exampe therefore, the best classifiers would be the Tree Ensemble

and RBF Kernel SVM. Classifiers like SVC-Linear, LDA, Logit all seem to plataeu at a true positive rate of around 0.85. These simple classifiers, whilst useful in some situations, do not adequately explain these photometric data. GaussianNB, also a simpler classifier, hits a slightly higher true positive rate at the expense of a very high false positive rate. QDA fares slightly better, with a reduced false positive rate, but still far off from the capabilities of a Tree Ensemble or RBF Kernel SVM.