Classification

PHY3287 - Computational Astronomy
Dr Andrea DeMarco
andrea.demarco@um.edu.mt

Classification

- Unsupervised Data Analysis
    - Data distributions
    - Inherent structure in data
- Supervised Classification
    - What if we had labels for our data?
    - We can build models that learn the relationship between data and its label
- We shall be looking for **classification boundaries** between distributions of data with different labels

Assigning Categories

- Relate a set of features to a predefine set of classes
- Where do we get labels from? Perhaps students have enough free time on their hands…
- We want to use ML to learn models of specific classes, and then apply these model to assign categories to future data - **inference**
- We shall look at two approaches - generative models and discriminative models

Classification Loss

- We need a way to quantify how good our classification models are
- A common cost/loss ruction is zero-one loss - assign a value of 1 for a misclassification and 0 for a correct classification e.g. HomePod identifying its owner (class label 1) vs. the rest of the world (class label 0)
- Furthermore we can distinguish between two types of error
- Assigning a label 1 to an object whose true class is 0 (false positive)
- Assigning a label 0 to an object whose true class is 1 (false negative)

Classification Loss

- We can formalise two measures to help us determine classifier performance.

- **Completeness**

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

- **Contamination**

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

Classification Loss

- Completeness measures the fraction of total detections identified
- Contamination measures the fraction of detected objects which are misclassified.
- We may want to optimise one over the other
- In the literature, completeness/contamination are sometimes referred to as sensitivity/Type 1 Error, or precision/recall

Naive Bayes Classification

- A group of extremely fast/simple classification algorithms
- Suitable for high-dimensional datasets
- Few parameters needing tuning
- A good baseline when starting off
- Relies on Bayes' theorem:

$$P(L \mid \text{features}) = \frac{P(\text{features} \mid L)P(L)}{P(\text{features})}$$

Naive Bayes Classification

- If we are trying to classify between 2 labels, L1 and L2, one way is to compute the ratio of posterior probabilities:

$$\frac{P(L_1 \mid \text{features})}{P(L_2 \mid \text{features})} = \frac{P(\text{features} \mid L_1)}{P(\text{features} \mid L_2)} \frac{P(L_1)}{P(L_2)}$$

- Just build a model to compute $P(\text{features} \mid L_i)$

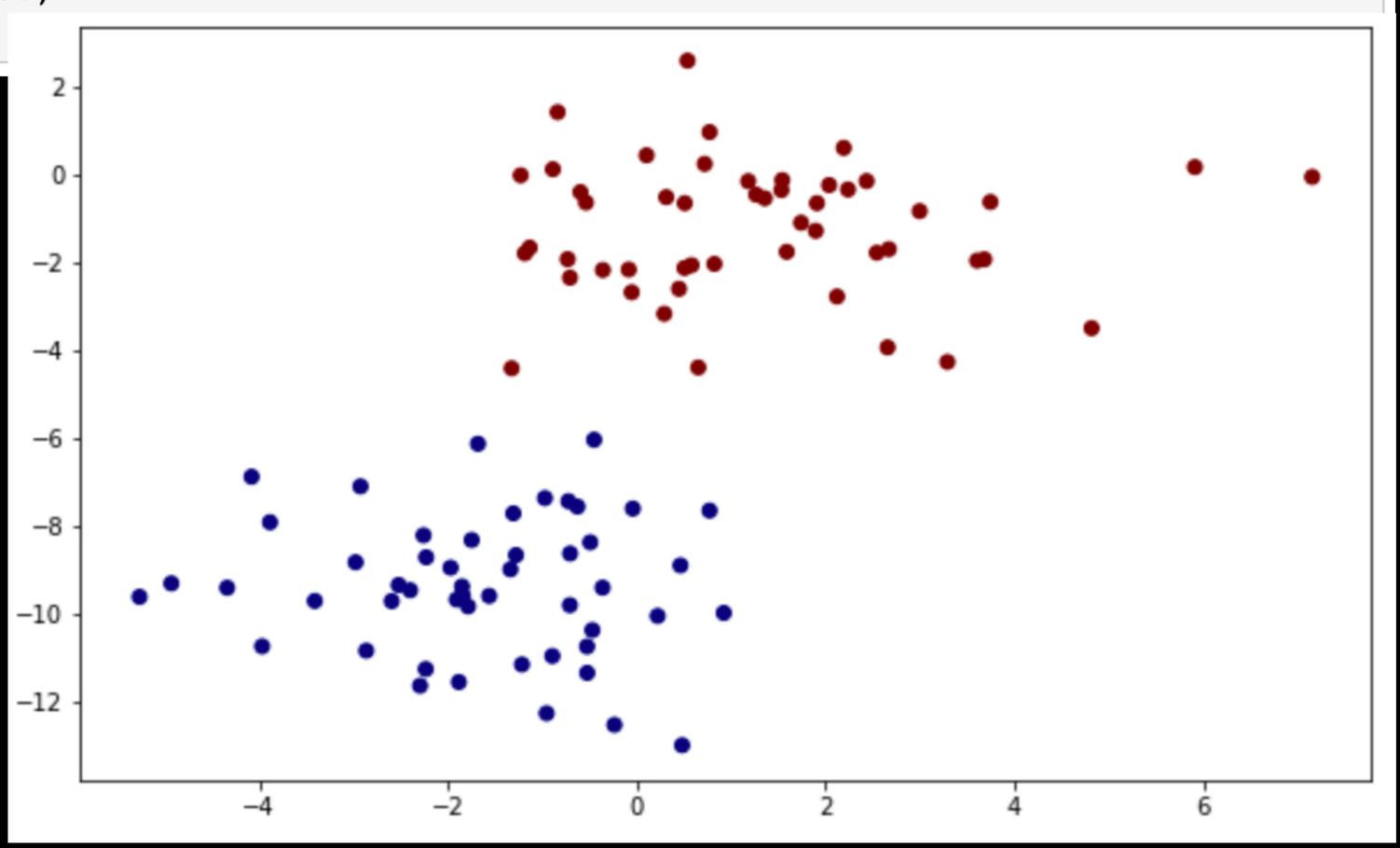
- These models are called **generative models** - they specify a hypothetical random process that could have generated the data in the first place

- We need to assume a model - this is where the ‘naive’ part comes in - we often assume a very simple model

Gaussian Naive Bayes

- The easiest naive Bayes classifier - Gaussian naive Bayes
- Assume that data/features are drawn from a Gaussian

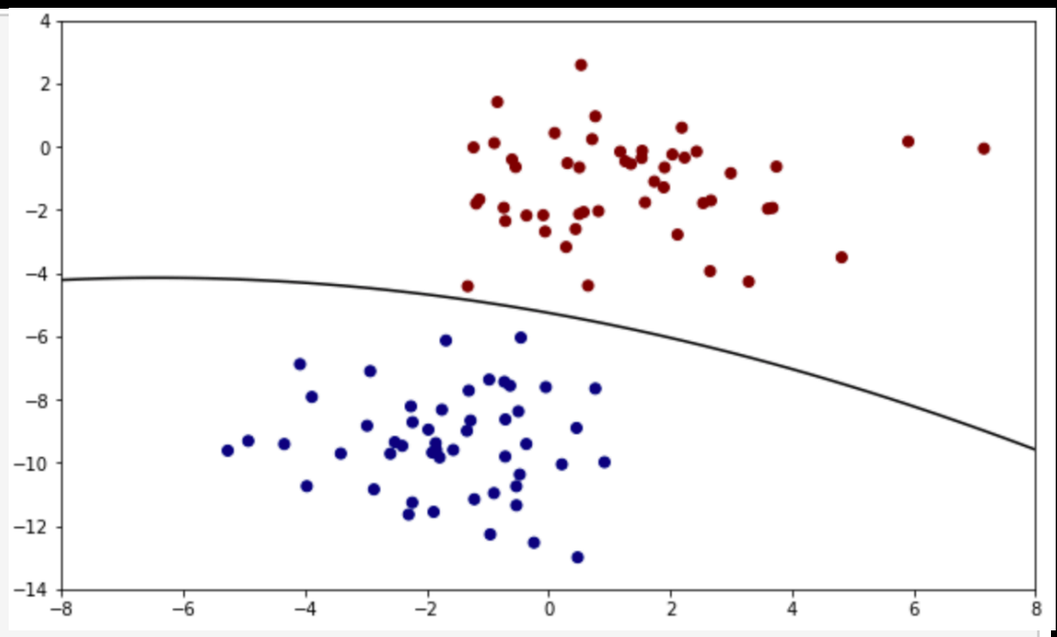
```
from sklearn.datasets import make_blobs
X, y = make_blobs(100, 2, centers=2, random_state=2, cluster_std=1.5)
fig, ax0 = plt.subplots(figsize=(10, 6))
ax0.scatter(X[:, 0], X[:, 1], c=y, cmap=cm.jet)
plt.show()
```



Gaussian Naive Bayes

- Assume data is generated by a Gaussian, just find mean/ standard deviation for each class/label

```
# 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()
```



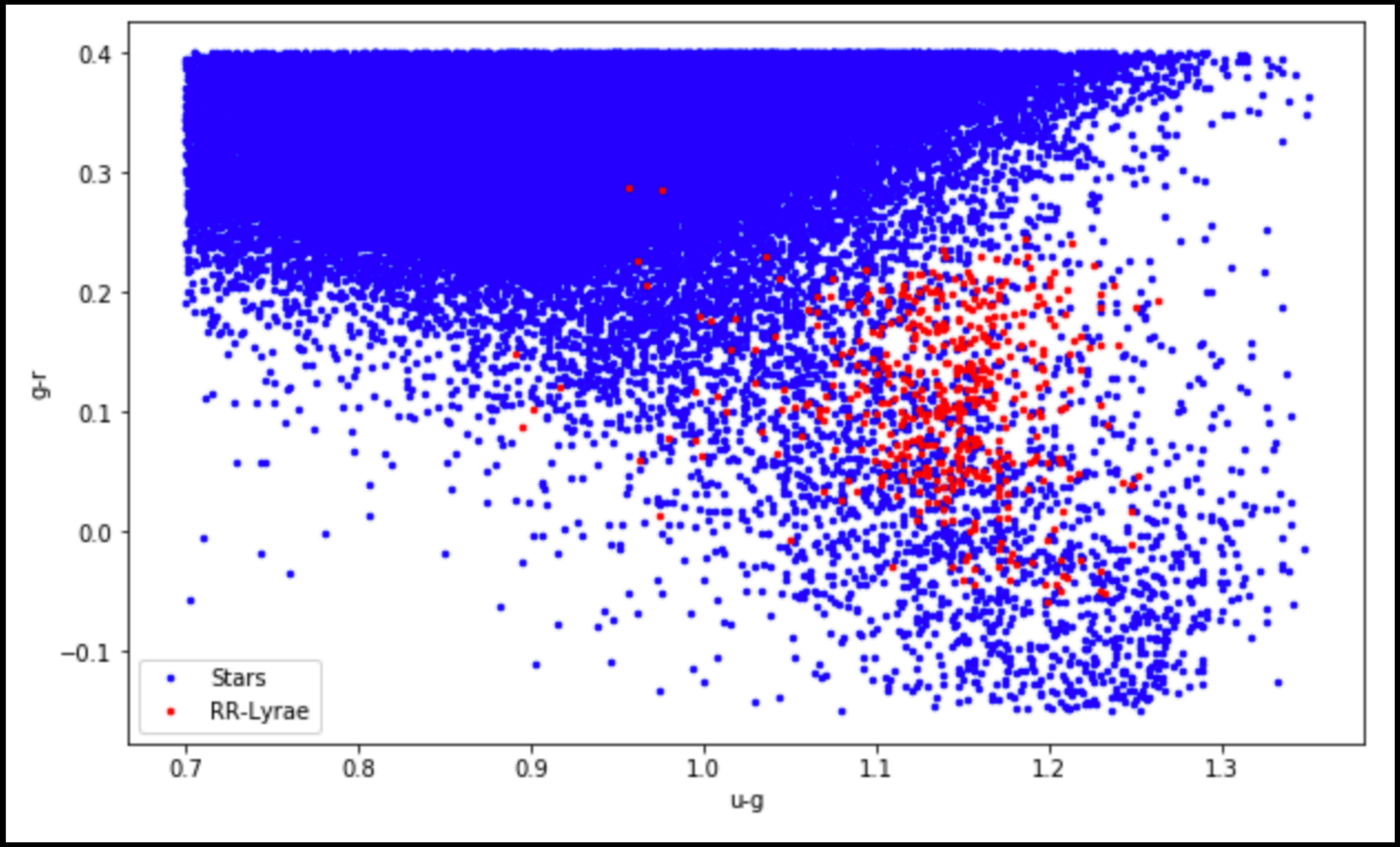
Gaussian Naive Bayes

- We have generated a multivariate (2D) Gaussian for each class
- Towards the denser centre, probability is higher and falls the further away we vary from the centre
- This Gaussian fit is done for both classes, and both models score the test data
- The model giving the highest probability for a sample ‘wins’ the classification battle

GNB Case Study: Star/LL-Rylae Classification

```
# 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()
```

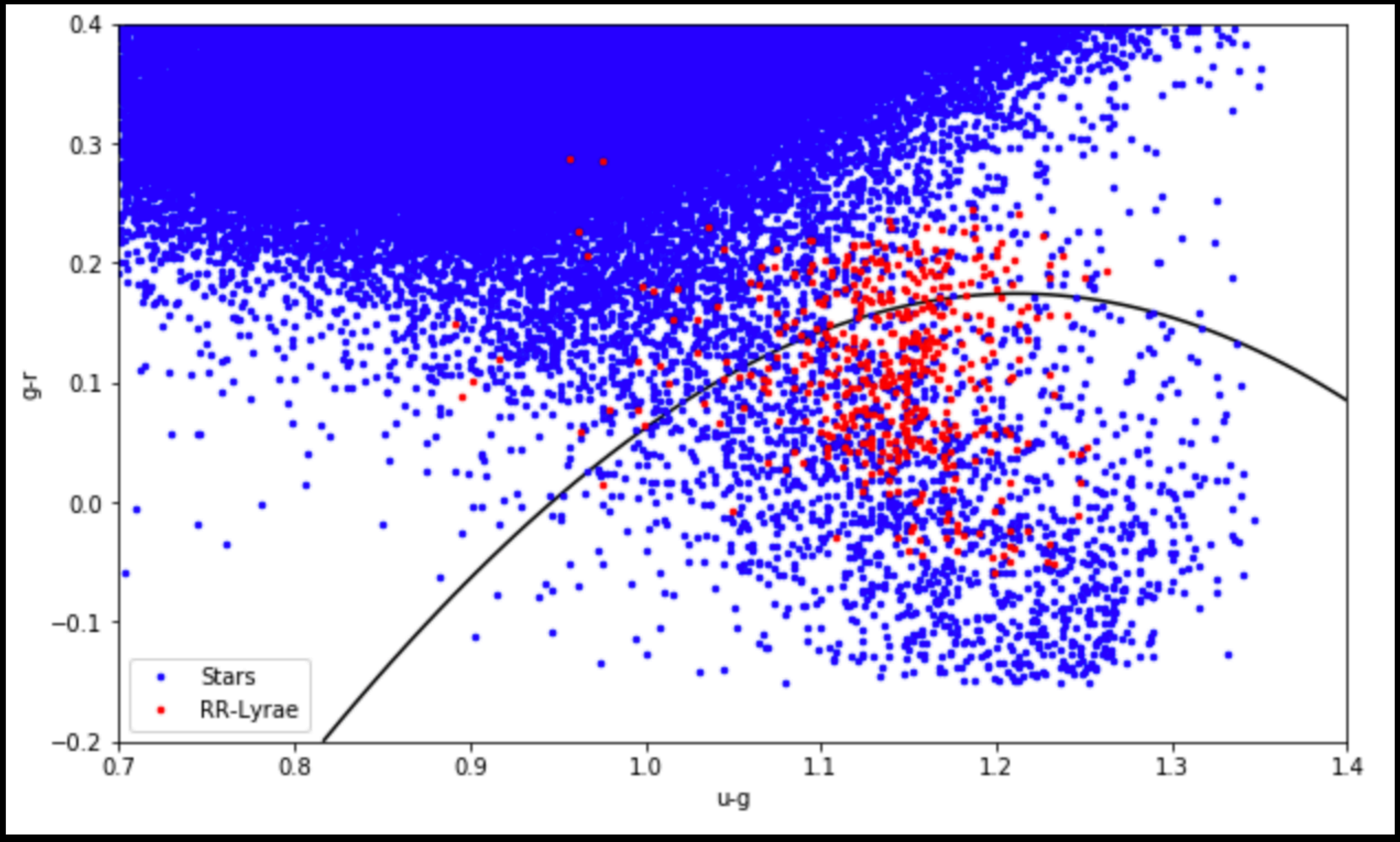
GNB Case Study: Star/LL-Rylae Classification



GNB Case Study: Star/LL-Rylae Classification

```
# 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()
```

GNB Case Study: Star/LL-Rylae Classification



GNB Case Study: Star/LL-Rylae Classification

- We now calculate completeness and contamination to evaluate our model

```
# 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
```

Gaussian Naive Bayes

- We can alternatively use other models (not just Gaussians)
- We are making stringent choices every time, but Naive Bayes classifiers are useful because:
  - They are extremely fast for both training/prediction
  - They provide a straightforward probabilistic prediction
  - Very easy to interpret
  - Very few parameters (if any) to tune

Gaussian Naive Bayes

- You can expect Naive Bayes Classifiers to perform well if:
  - The naive assumptions actually match the data (rare in practice)
  - Categories are well-separated, and model complexity becomes less important
  - Data is high-dimensional (and again, model complexity is less important)

Linear and Quadratic Discriminant Analysis (LDA/QDA)

- LDA also relies on simplifying assumptions about the class distribution

- Assumes the distributions have identical covariances for all K classes - i.e. each class is a shifted Gaussian from another

- Classifier derived from class posteriors in this model:

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

- Forms a linear boundary between all pairs of classes

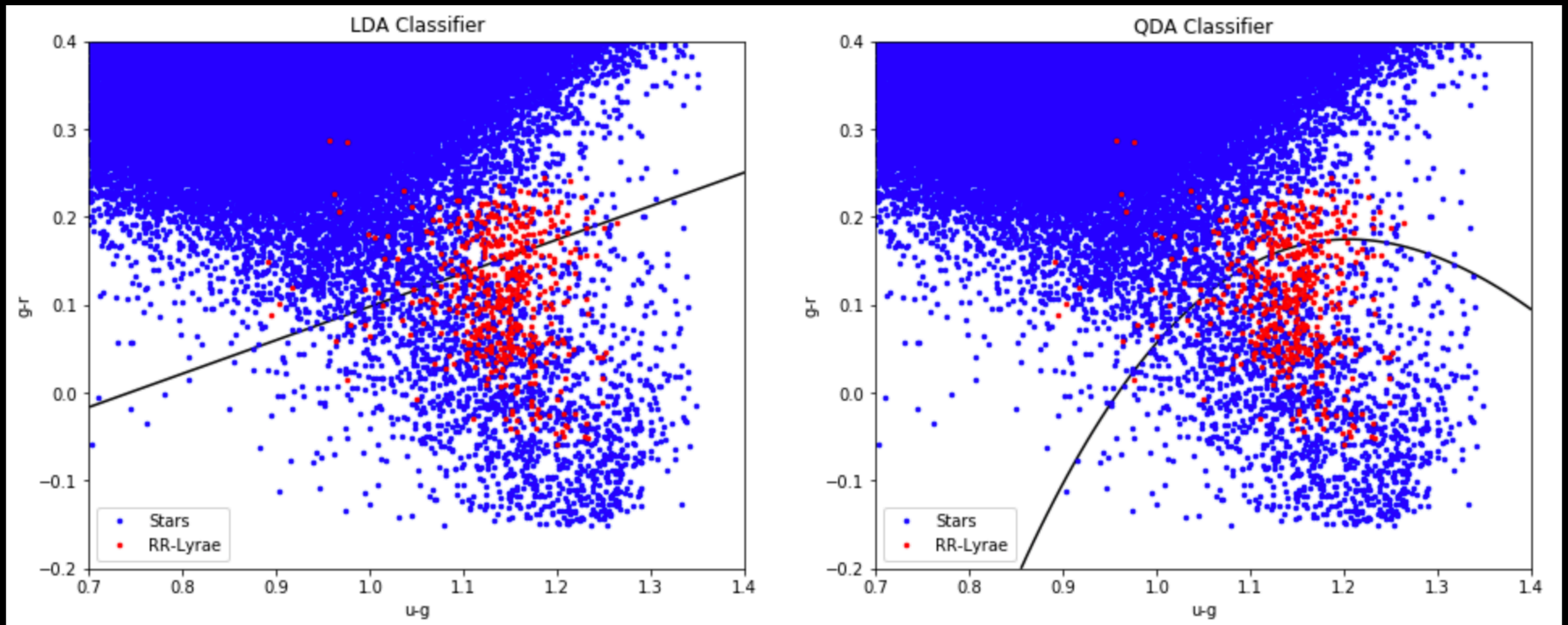
Linear and Quadratic Discriminant Analysis (LDA/QDA)

- In QDA, we relax the constraint of a shared covariance across all classes

- The discriminating boundary is now quadratic between each pair of classes

```
22 # fit an LDA classifier and get predictions
23 lda = LDA()
24 lda.fit(samples_2d, labels)
25 lda_labels_pred = lda.predict(samples_2d)
26
27 # fit a QDA classifier and get predictions
28 qda = QDA()
29 qda.fit(samples_2d, labels)
30 qda_labels_pred = qda.predict(samples_2d)
```

Linear and Quadratic Discriminant Analysis (LDA/QDA)



- As can be expected, QDA yields better completeness/ contamination

Linear and Quadratic Discriminant Analysis (LDA/QDA)

- And now for scoring the classifiers….

```
# 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
```

Discriminative Classification

- The classifiers we have look at so far focus on analysing the data for each particular class, finding distribution characteristics for a good fit on the data

- Optimise fit on data of one class, minimising fit on data from another class

- We termed this form as **generative** classification.

- We shall now discuss **discriminative** classification - the class density estimates are skipped in favour of a classification decision boundary (no assumptions about the data distribution)

Logistic Regression

- A modelling algorithm for binary categorical class variable Y i.e. either 1 or 0

- The goal is to determine a mathematical equation that can be used to predict the probability of event 1

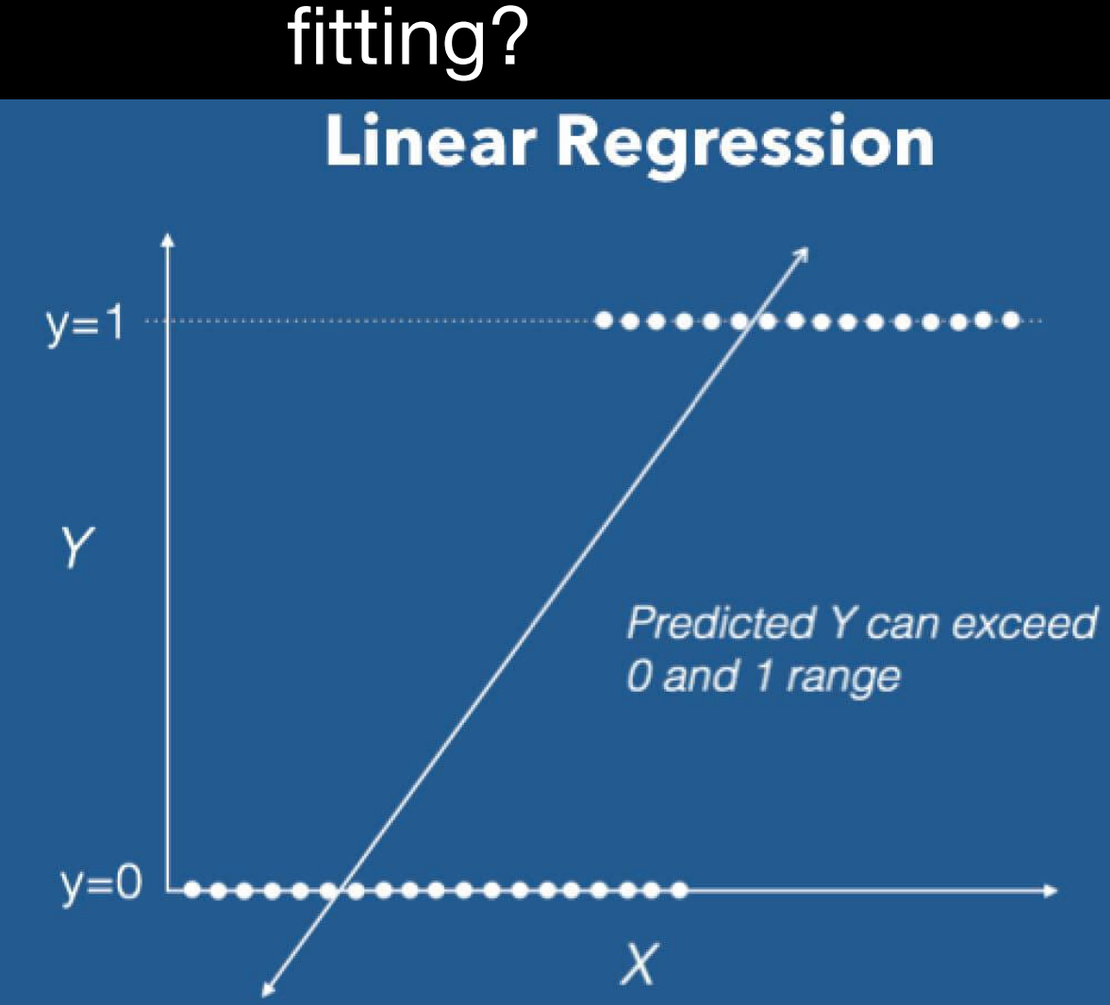
- We previously discussed linear regression to predict a continuous value of Y for an input X

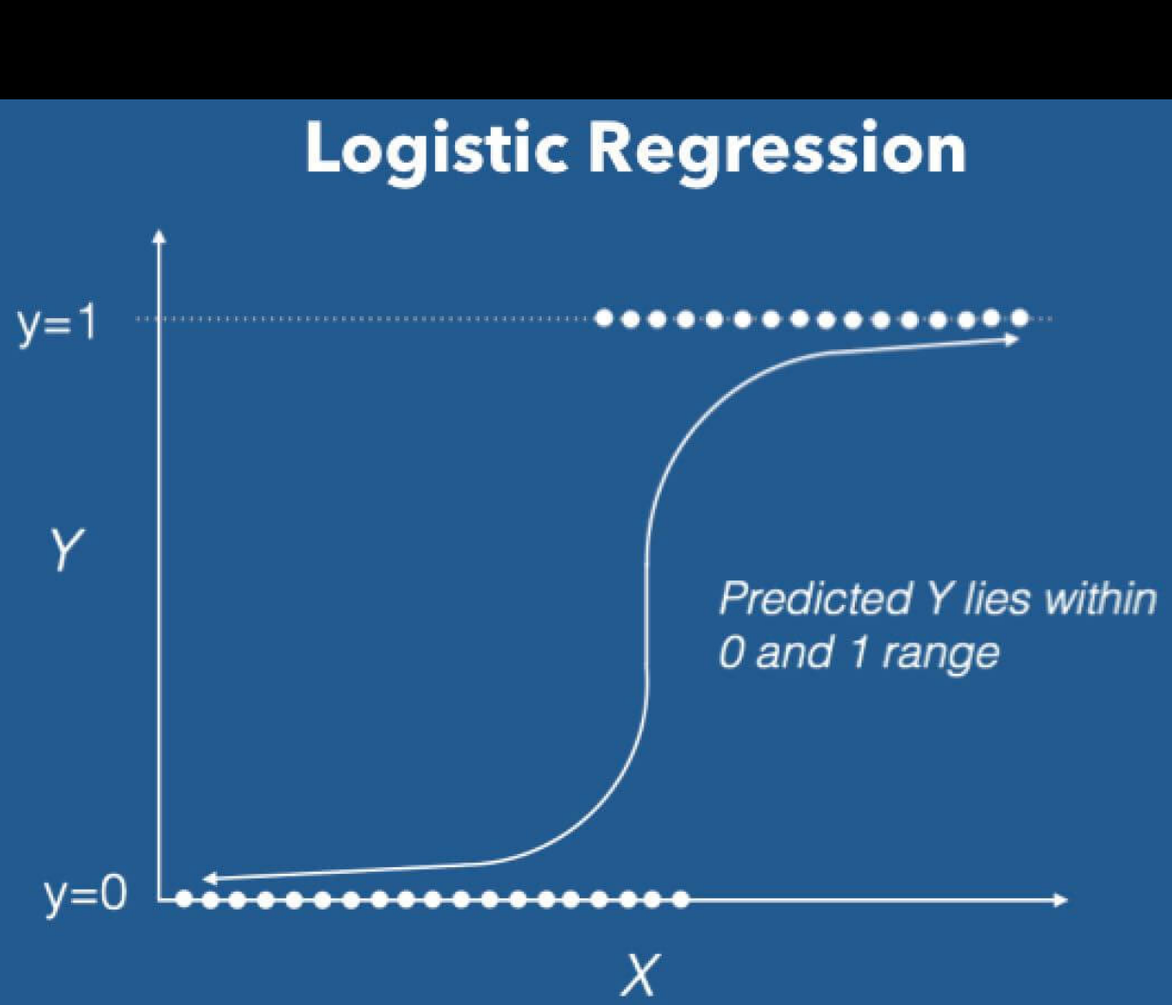
- Here we need Y to be categorical (discrete)

- Vanilla logistic regression only works for binary classification, for multiple classes, we need a few tweaks like multiple one-vs-all classifiers

Logistic Regression

- Our example of Star vs LL-Rylae classification is a suitable problem
- So why can’t we use logistic regression for category fitting?





Logistic Regression

- With logistic regression we predict the probability score that reflects the probability of occurrence of an event.

- Each sample in the training/test set is an ‘event’ - e.g. a sample could be a given value in a color-color space

- Take the log odds of the event $ln(P/1-P)$, where P is the probability of an event, always lying between 0 and 1

Logistic Regression

- The logistic regression formulation is:

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

- Where the logistic function (logic) is defined as:

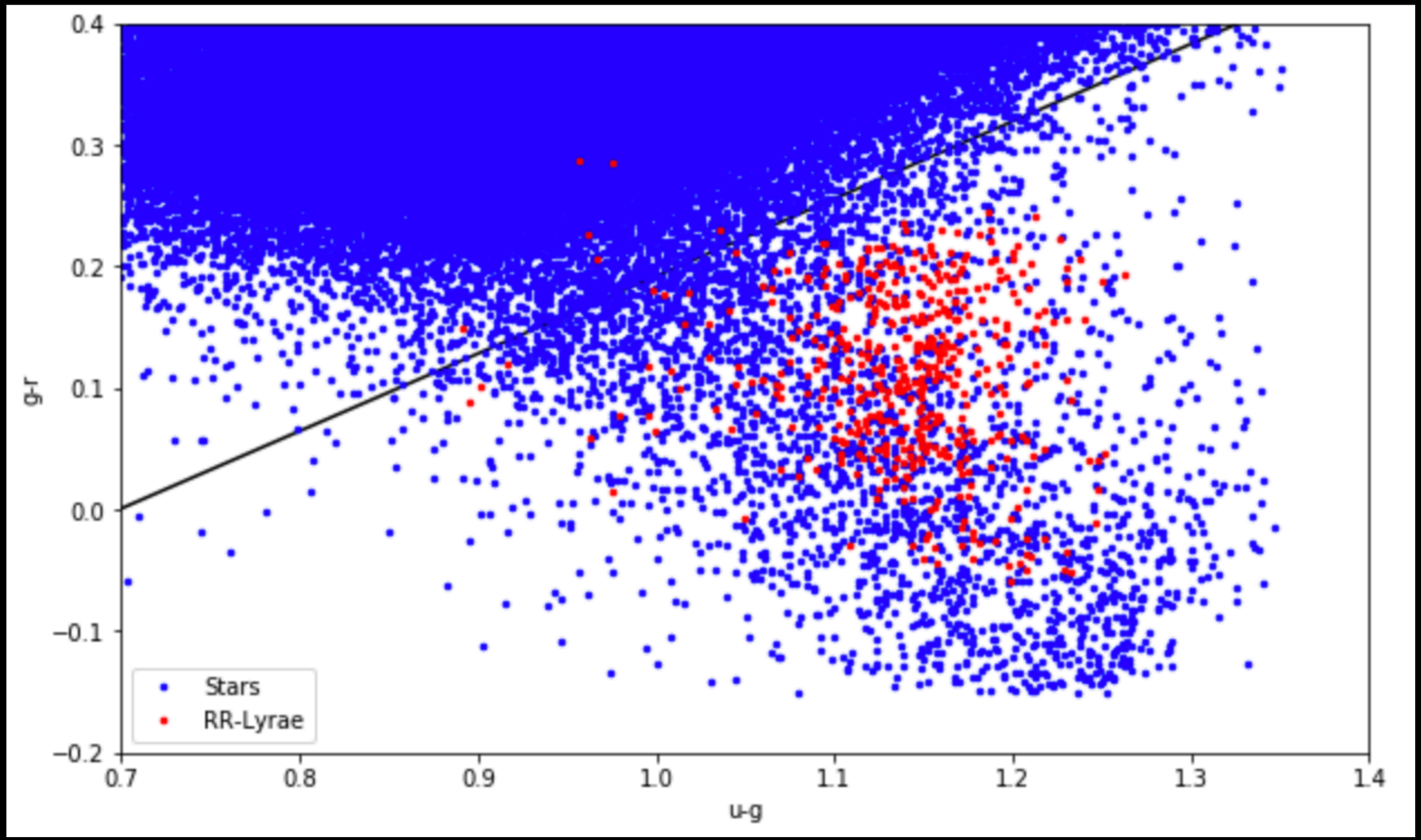
$$\text{logit}(p_i) = \log\left(\frac{p_i}{1 - p_i}\right) = \sum_j \theta_j x_i^j$$

- The name logistic regression comes from the fact that the function $e^x/(1 + e^x)$ is called the logistic/sigmoid function

Logistic Regression

```
# 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()
```

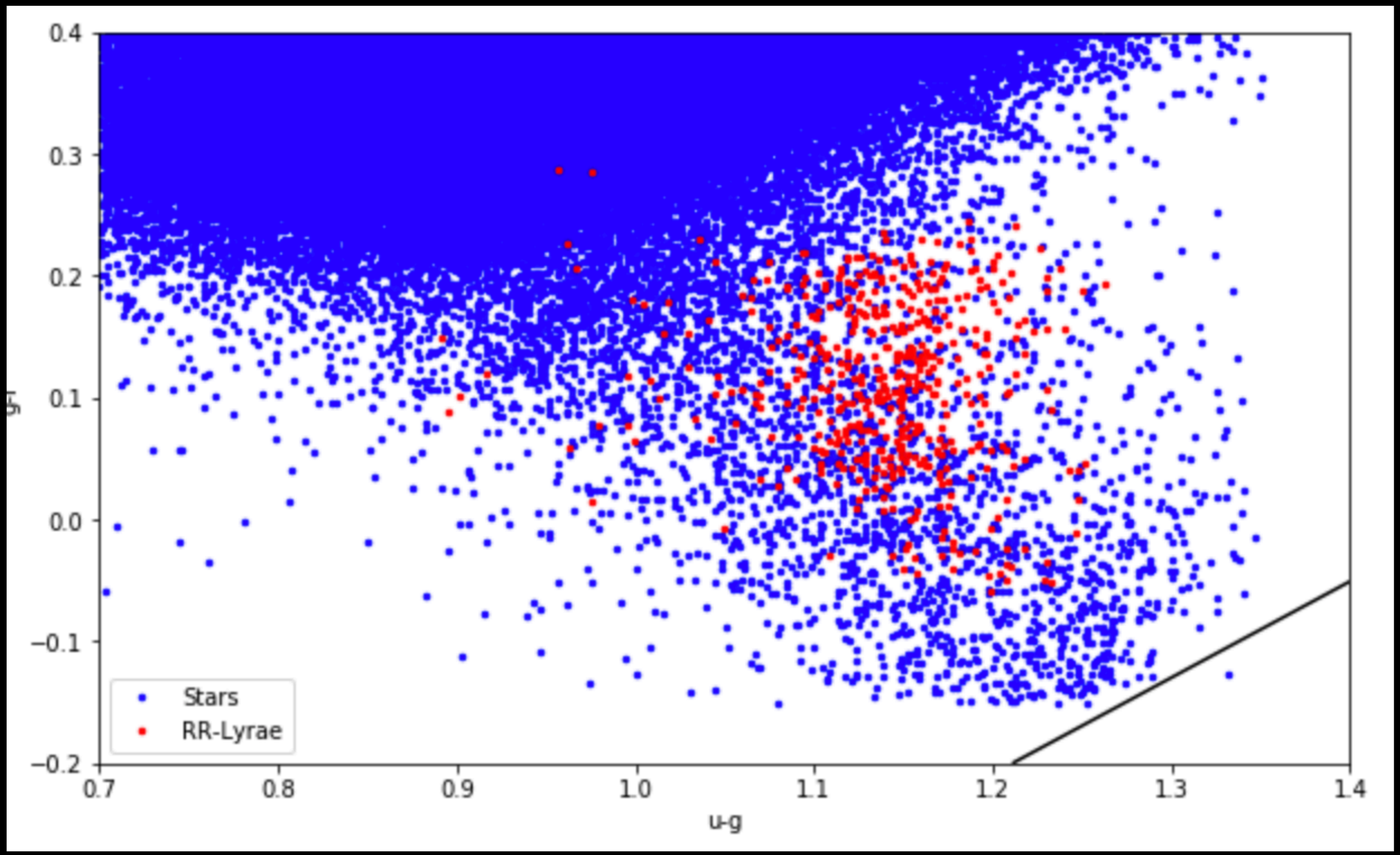
Logistic Regression



Logistic Regression

- In general there are many parameters we could tweak for Logit
- We have used one ‘class_weight’ and set this to ‘balanced’
- This is very important - by default all classes have a weight of one - it is assumed there are roughly an equal amount of samples per class
- When this is not the case (almost all the time), we have to balance out weights inversely proportional to class sample frequencies
- If we ignore this imbalance for our data…

Logistic Regression



Logistic Regression

- And now we evaluate the classifier:

```
# 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
```

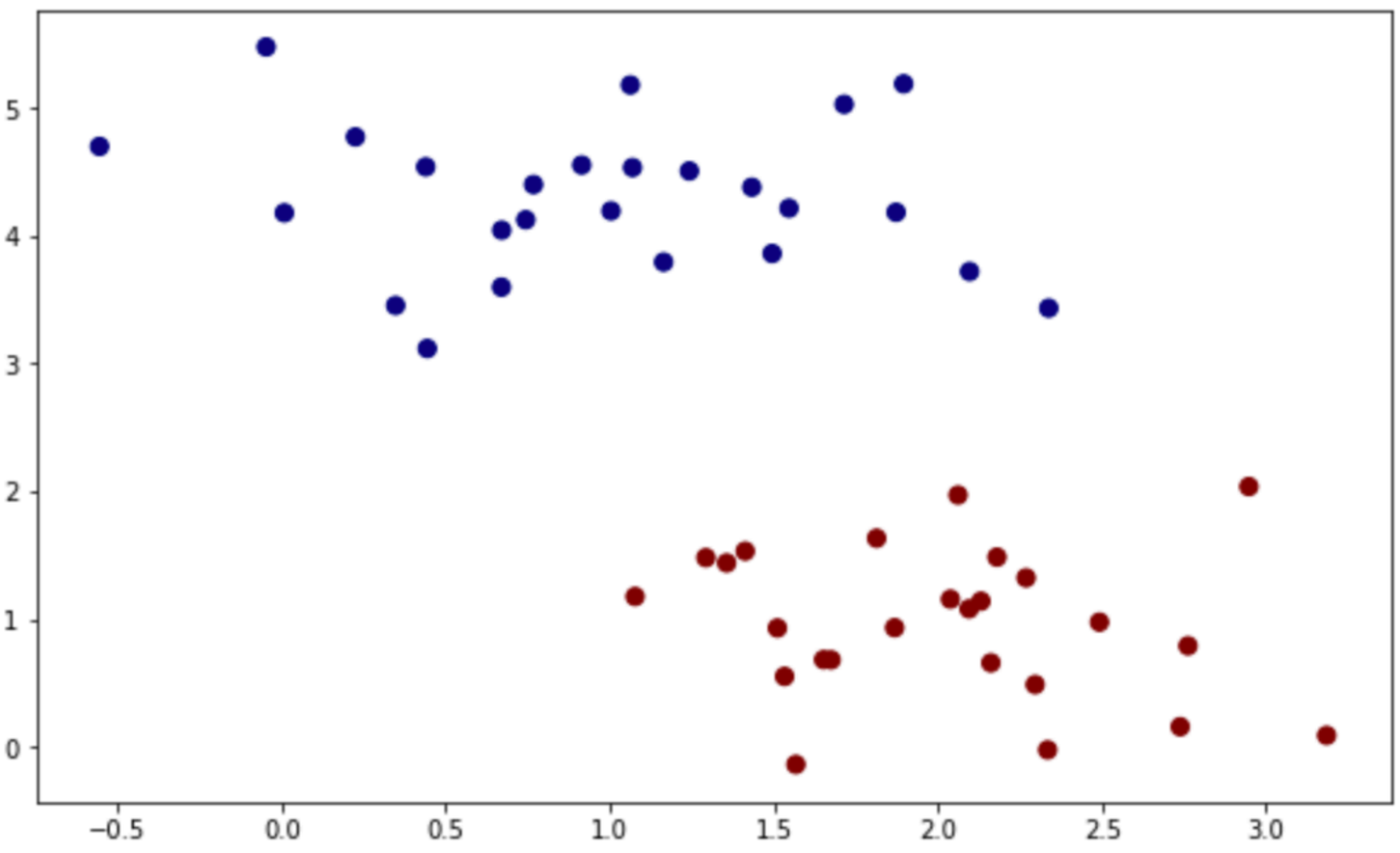
- So far, we have the best result on this dataset
- Discriminative classifiers can get very powerful, and can account for unseen training data more than generative classifiers on average

Support Vector Machines (SVMs)

- Another method for choosing a linear decision boundary
- Very powerful and flexible supervised algorithm
- Consider the task of finding a hyperplane that maximises the distance of the closest points from either class
- This distance will be called the **margin**
- The points considered when setting up the margin will be called **support vectors**

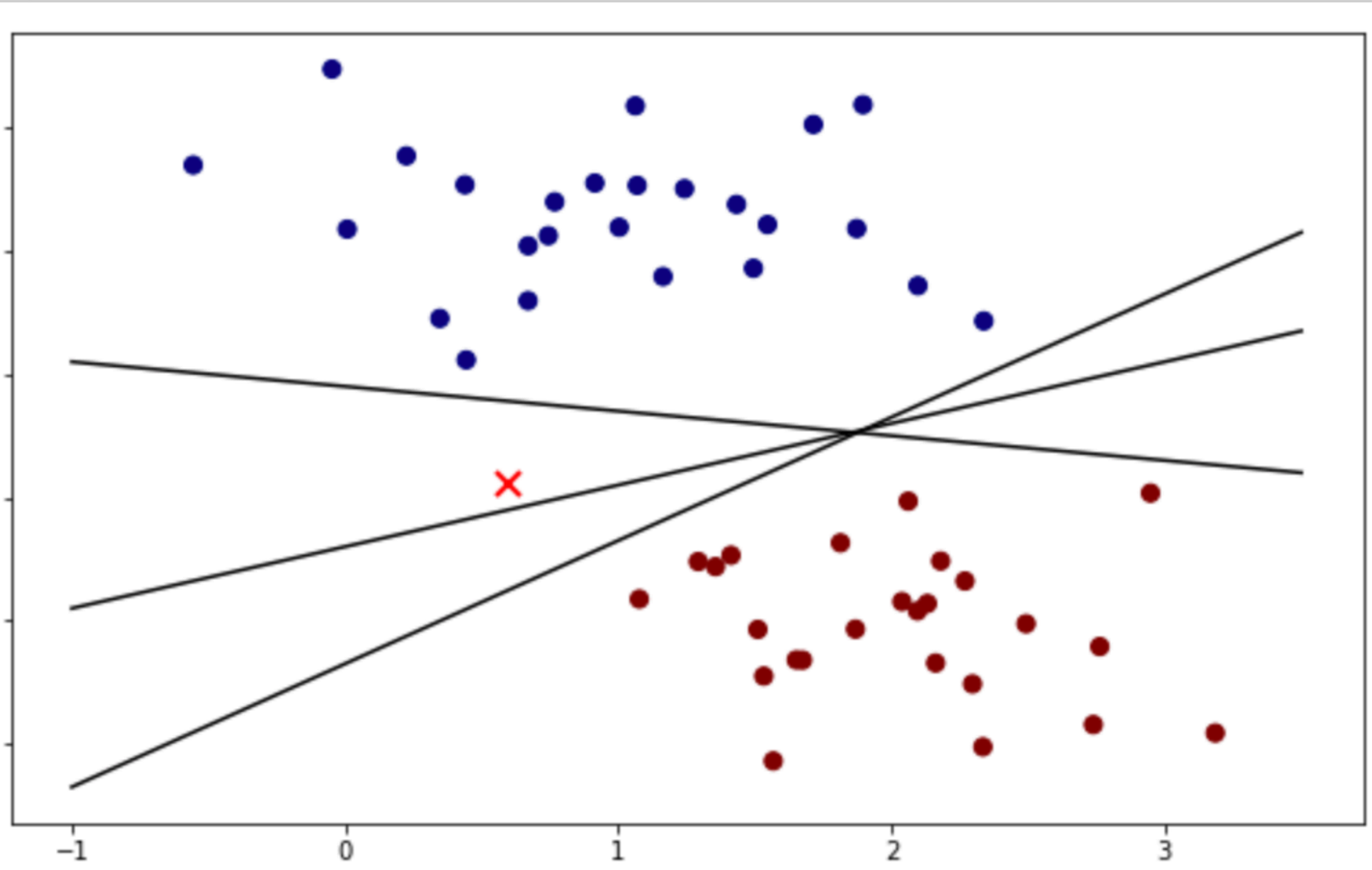
Support Vector Machines (SVMs)

```
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');
```



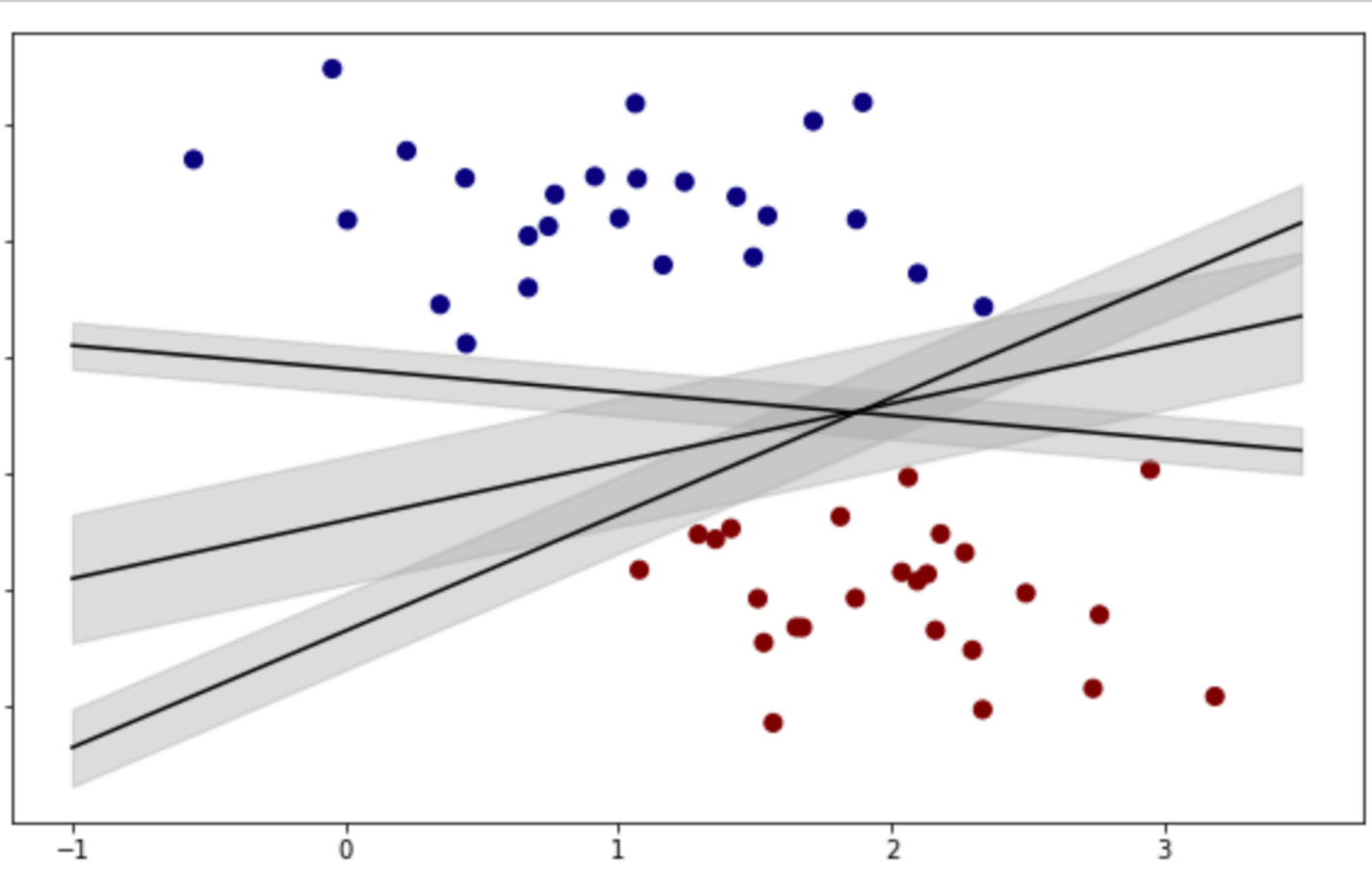
Support Vector Machines (SVMs)

```
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()
```



Support Vector Machines (SVMs)

```
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)
```



Support Vector Machines (SVMs)

- SVMs are an example of a **maximum margin estimator**

- We need to find the line (or hyperplane) that maximises the margin - that is the optimal discriminating plane

- Fitting is easy enough in Python:

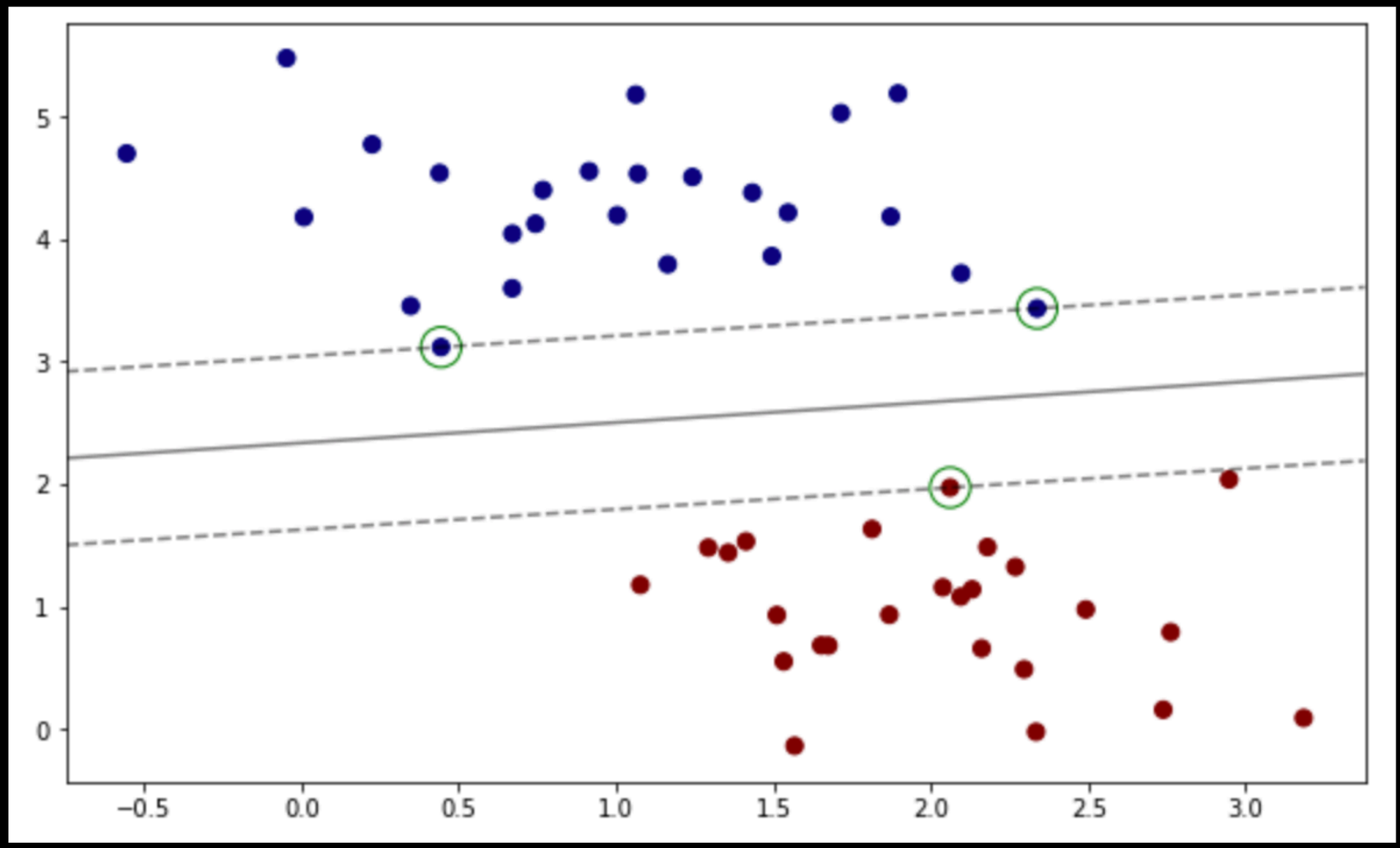
```
from sklearn.svm import SVC # "Support vector classifier"
model = SVC(kernel='linear', C=1E10)
model.fit(X, y)
```

```
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)
```

- Ignore ‘C’ for now…

Support Vector Machines (SVMs)

- Fitting an SVM and selecting support vectors/maximum margin



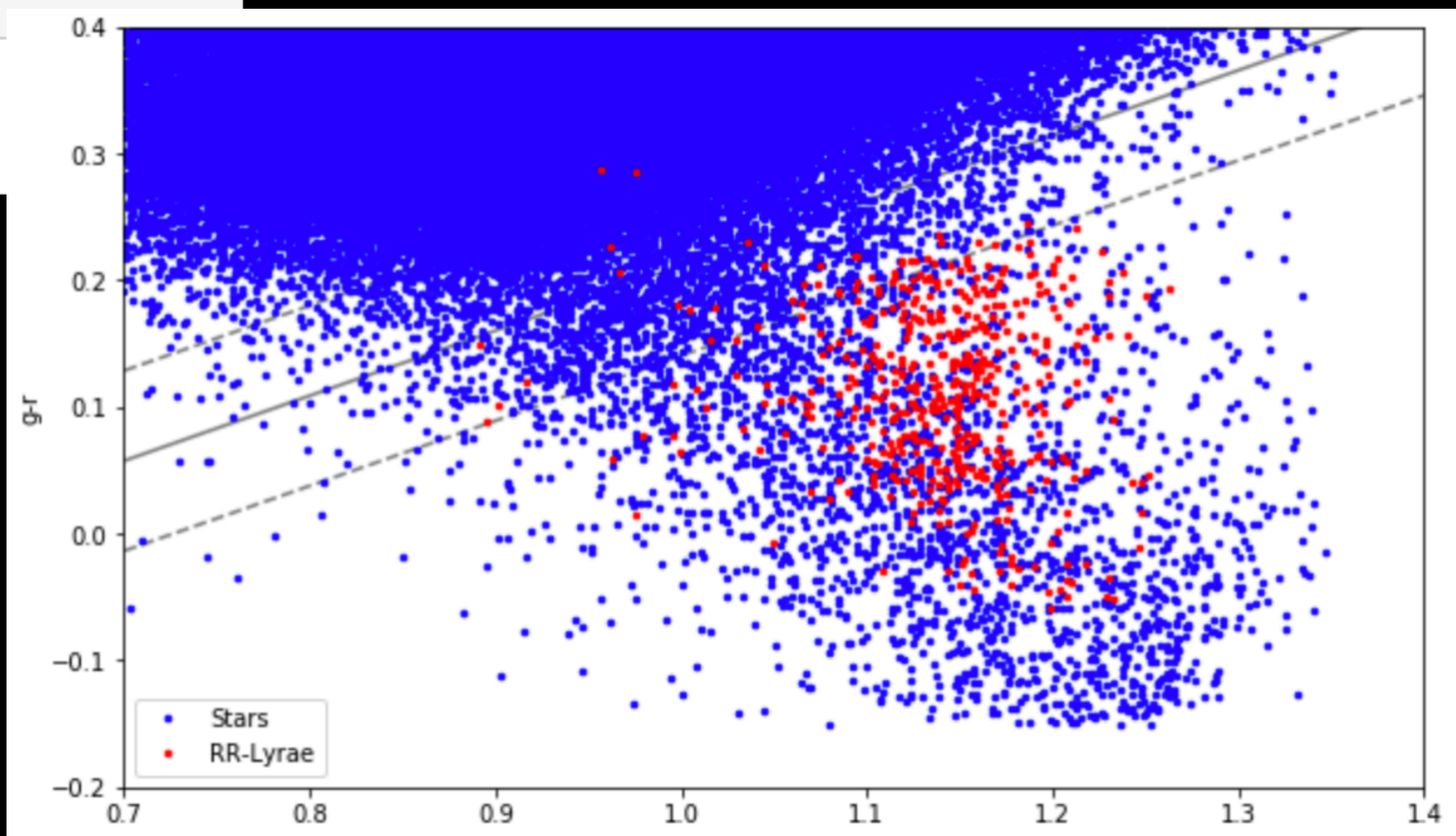
Support Vector Machines (SVMs)

- Trying out SVMs on our Start vs RR-Lyrae dataset:

```
# 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)
```

```
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
```

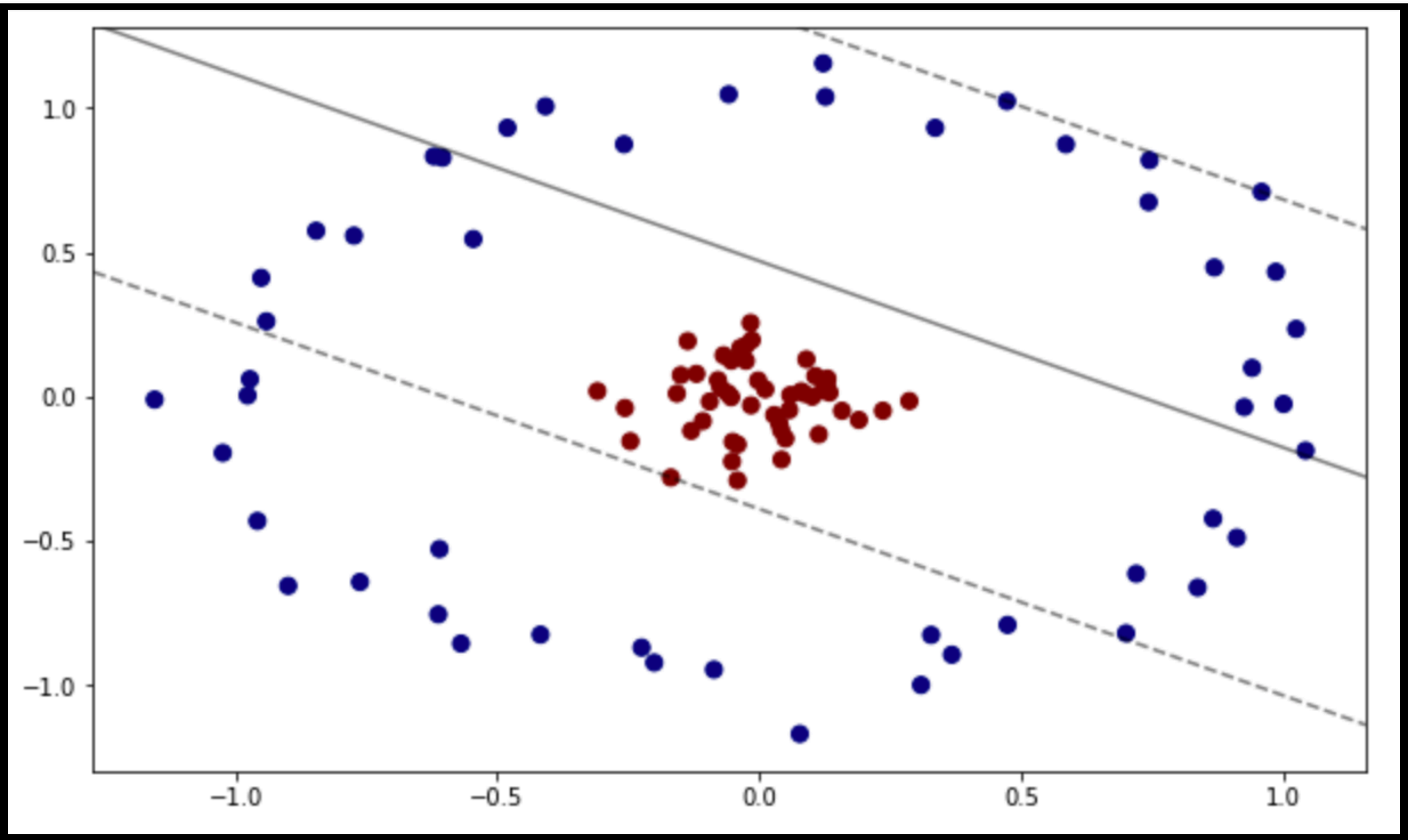


Non-Linear SVMs

- SVMs become a lot more powerful when combined with kernels
- A kernel is a function that projects our data into a higher-dimensional space defined by polynomials and Gaussian basis functions
- The hope is that in a higher dimensional space, there will be a linear separation which is otherwise not present in the original dimensionality
- We would be Abe to fit for nonlinearity in the data with a linear classifier
- OK, some intuition…

Non-Linear SVMs

```
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);
```

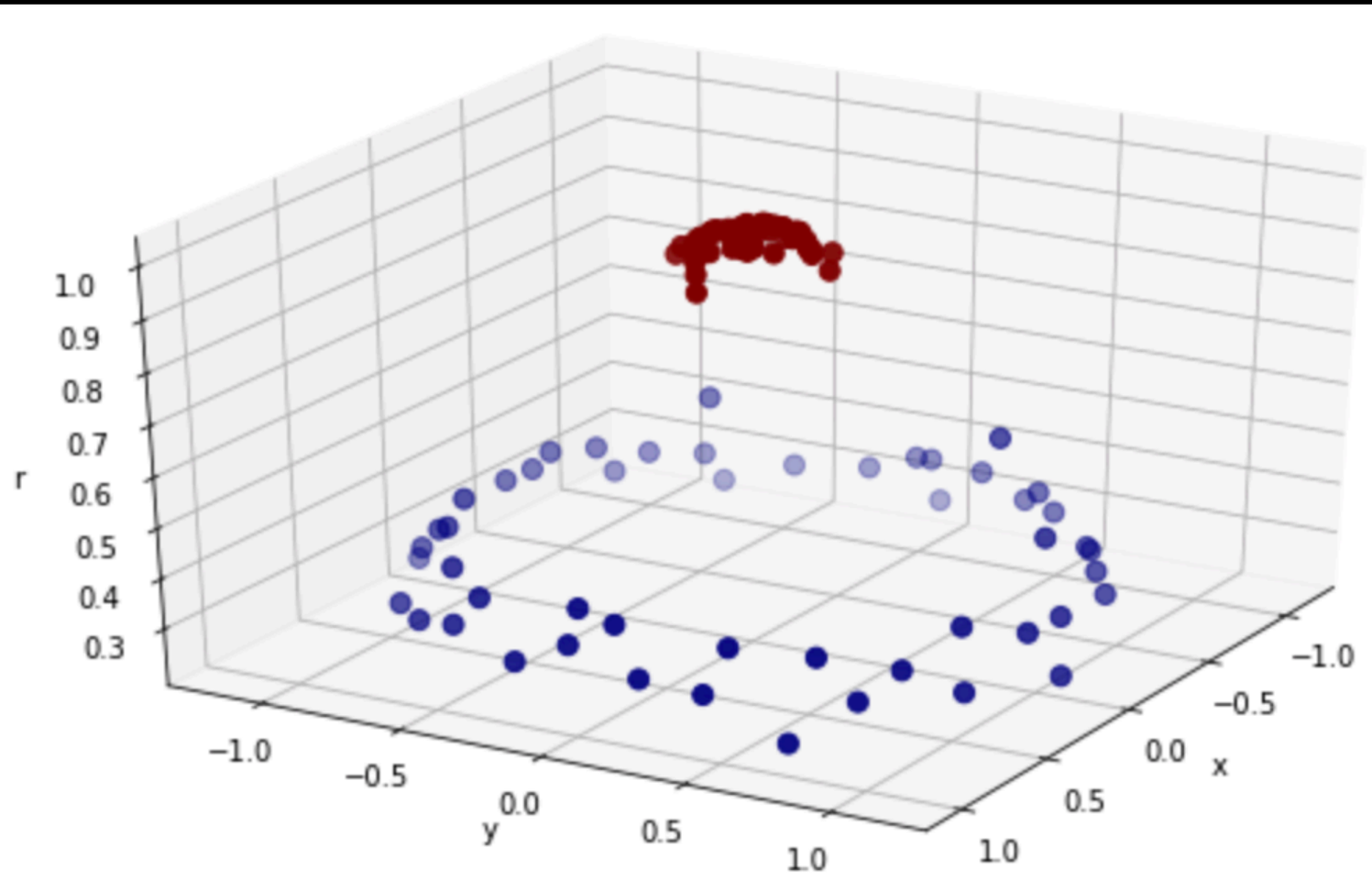


Non-Linear SVMs

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

```
r = np.exp(-(X ** 2).sum(1))
```

```
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);
```



Non-Linear SVMs

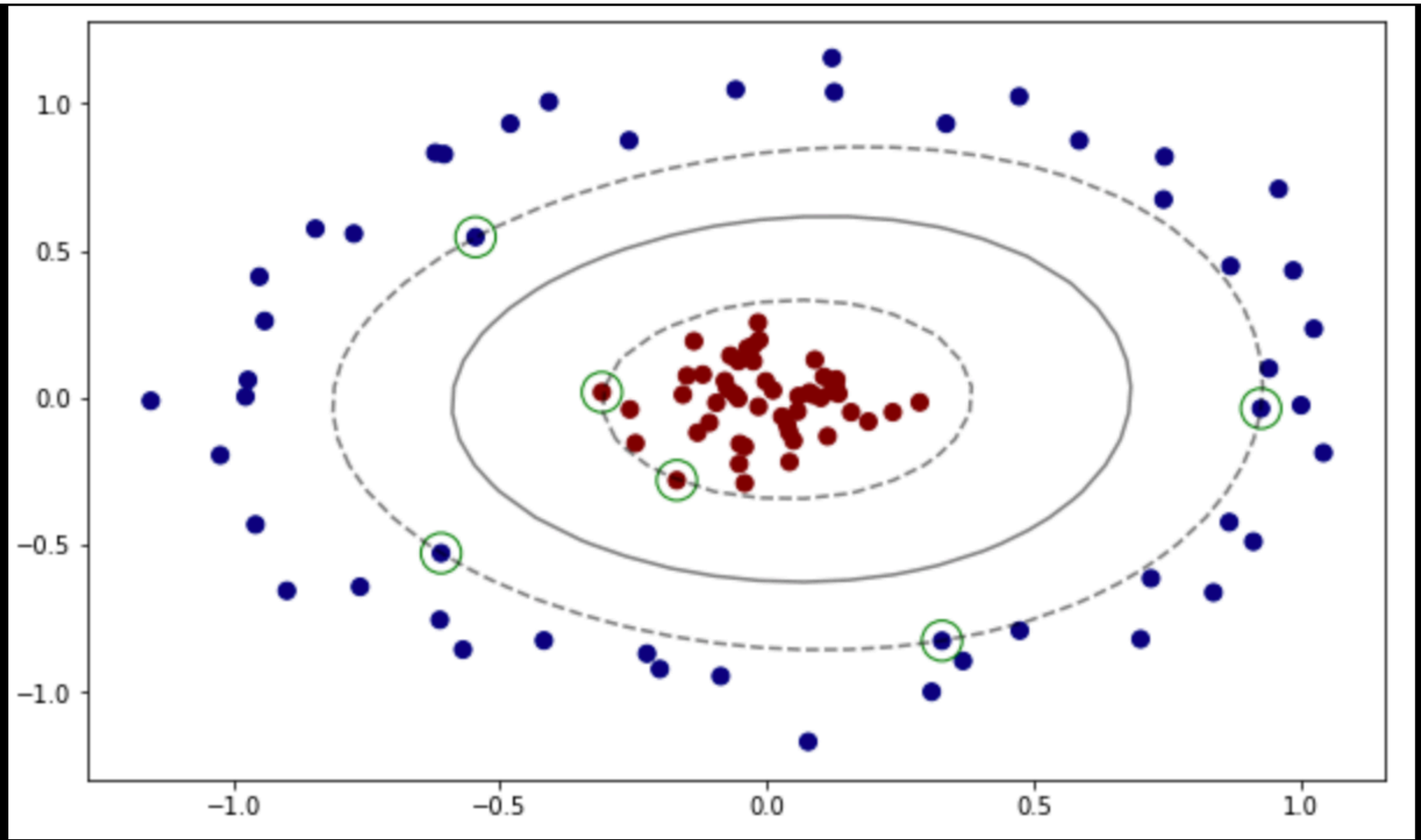
- We can observe many possible hyperplanes that separate the data
- This process is called a ‘kernel transformation’ - and can be computationally expensive to apply it to the point data directly
- A ‘kernel trick’ is a computational simplification that just transforms the inner products between all pairs of points
- Sklearn does all of this for you - but let’s look at how the separating hyperplane is resolved

Non-Linear SVMs

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

```
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)
```

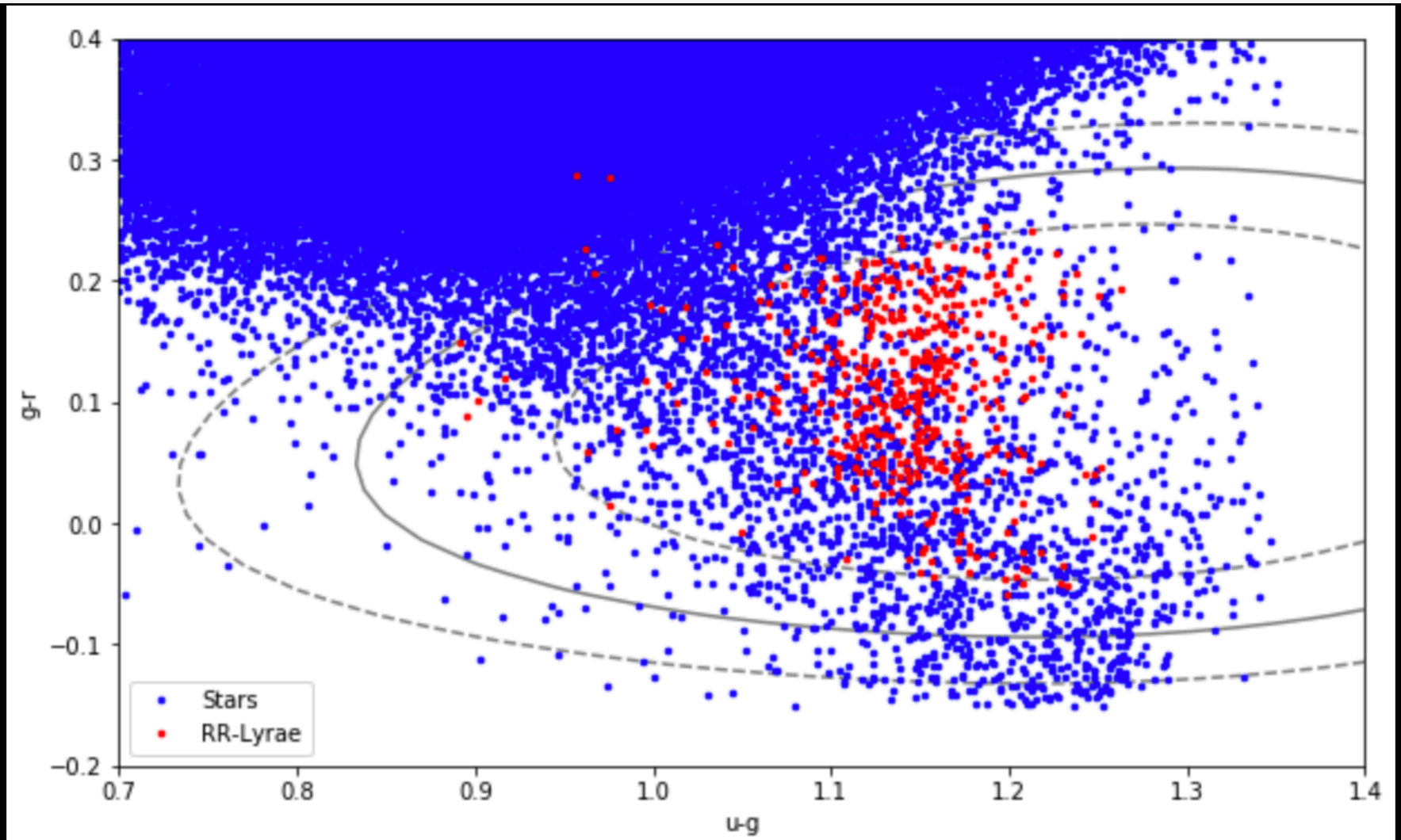
```
fig, svm7 = plt.subplots(figsize=(10, 6))
svm7.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='jet')
plot_svc_decision_function(clf,ax=svm7)
```



Non-Linear SVMs

- Trying it out on our Star vs LL-Rylae problem:

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



Non-Linear SVMs

- And now let's evaluate our performance.
- Linear SVMs first:

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

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

- And Non-Linear SVMs:

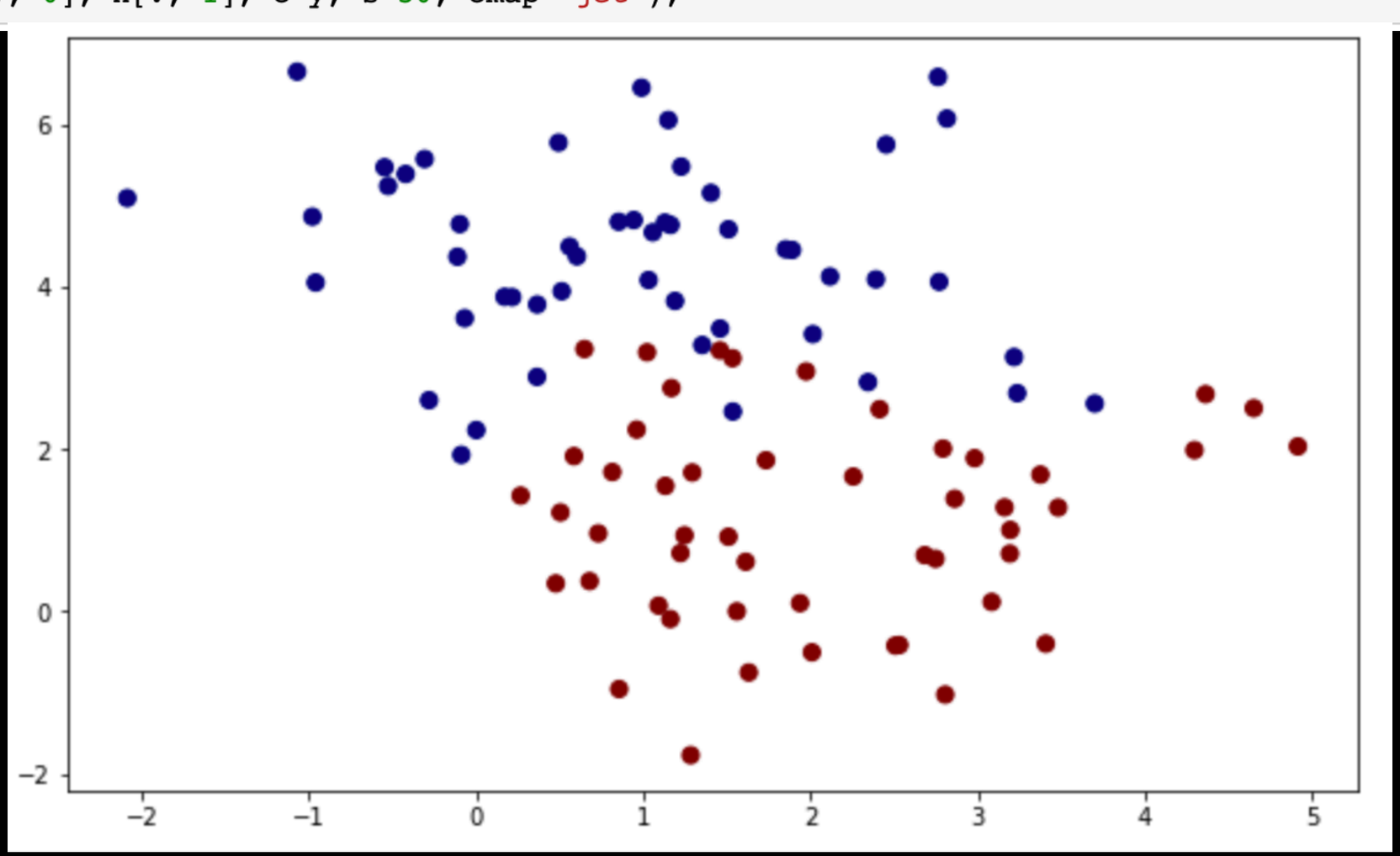
```
12 completeness_score = recall_score(labels,svc_pred)
13 contamination_score = (1-precision_score(labels,svc_pred))
14 print('SVC - RBF')
15 print('Completeness: %f'%completeness_score)
16 print('Contamination: %f'%contamination_score)
```

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

SVM Tuning

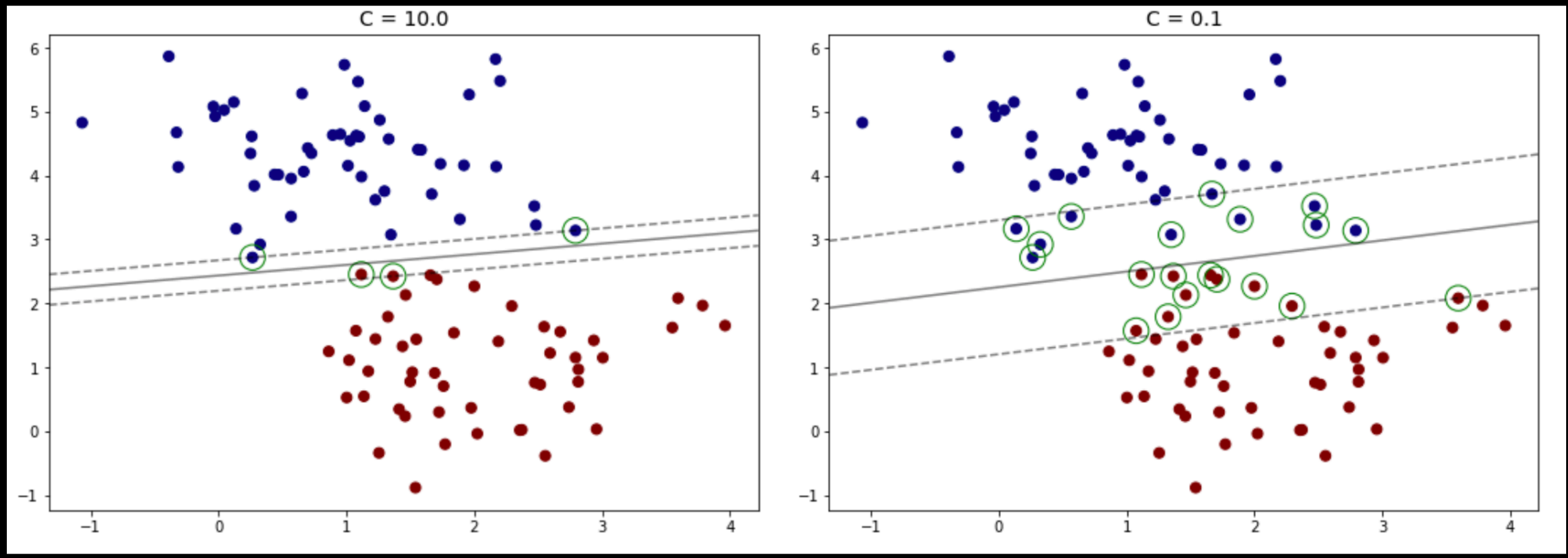
- Sometimes, the first SVM you'll try will not work. There's quite a lot of tweaking you can try to fit your data.

```
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');
```



SVM Tuning

- You can play with softening the margins - allowing some points onto the margin (which may improve the fit)



SVM Tuning

- The important thing is to score the various SVM models with their respective parameters on a validation set
- This is the best indicator on how the same model will perform during inference

```
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
```

SVM Overfitting

- You may observe that as C gets higher, contamination is decreased
- That’s good - but be careful not to be too strict - the SVM may just be over optimising on the data it sees in training i.e will not generalise well on unseen data
- We call this overfitting (and it applies to any ML training)
- The best way to check for overfitting is to always test on a validation set

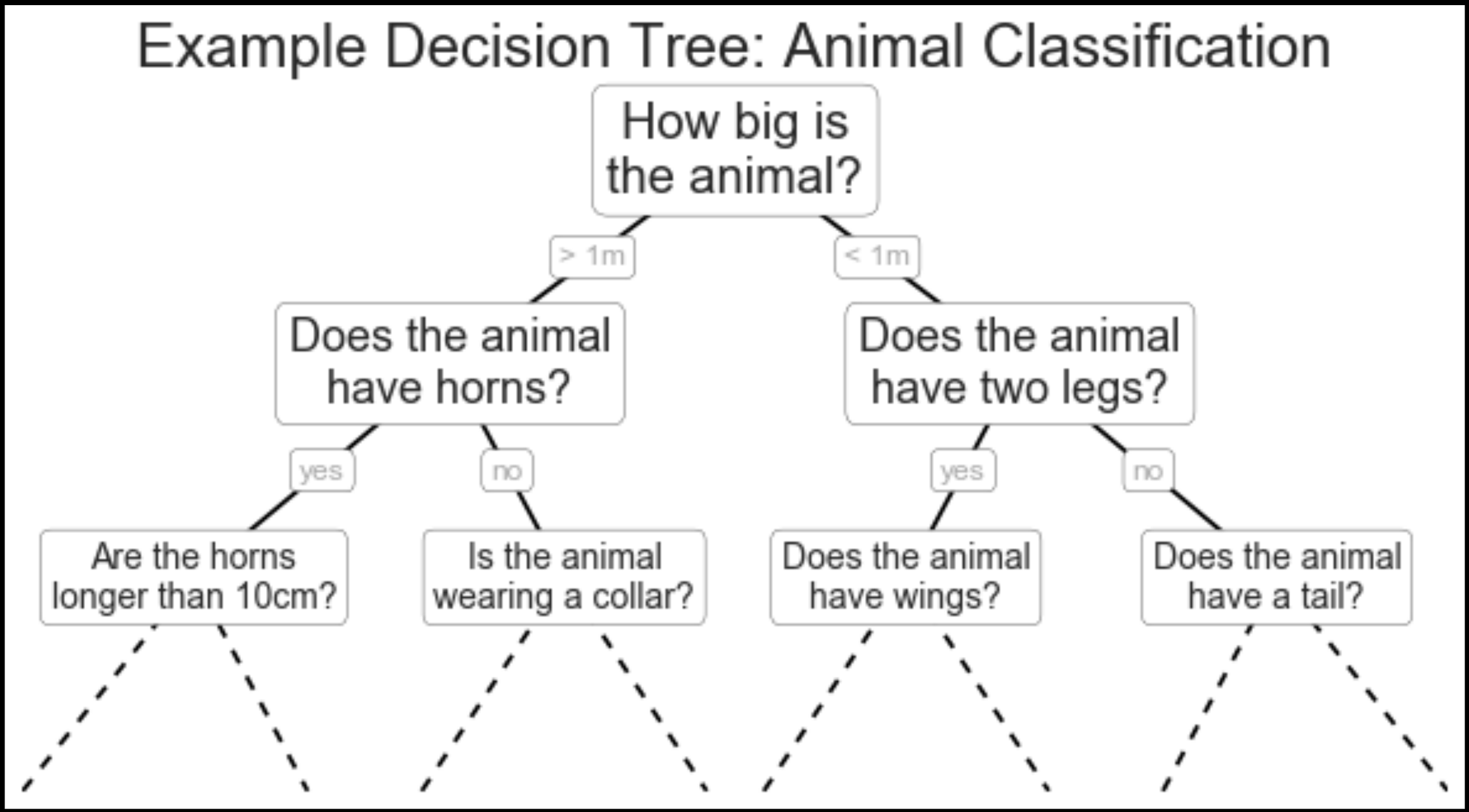
SVM Recap

- SVMs are compact models - relying only on a small set of support vectors
- Training can be slow, but prediction is very fast
- They work well with high-dimensional data, even with more dimensions than samples!
- Kernel methods make SVMs very versatile and adaptable to different problems
- SVMs scale badly as training samples increase - computationally heavy
- SVMs will require a long process of tuning, and not just the ‘C’ parameter
- Try SVMs when other, simpler methods have failed

Decision Trees and Random Forests

- Random forests are an example of an **ensemble method**
- Relies on aggregating the results of an ensemble of simple estimators
- The sum is greater than the parts - a majority vote among estimators is better than any individual estimator doing the voting
- In the case of random forests, the ensemble is made up of decision trees, so let’s discuss those first

Decision Trees



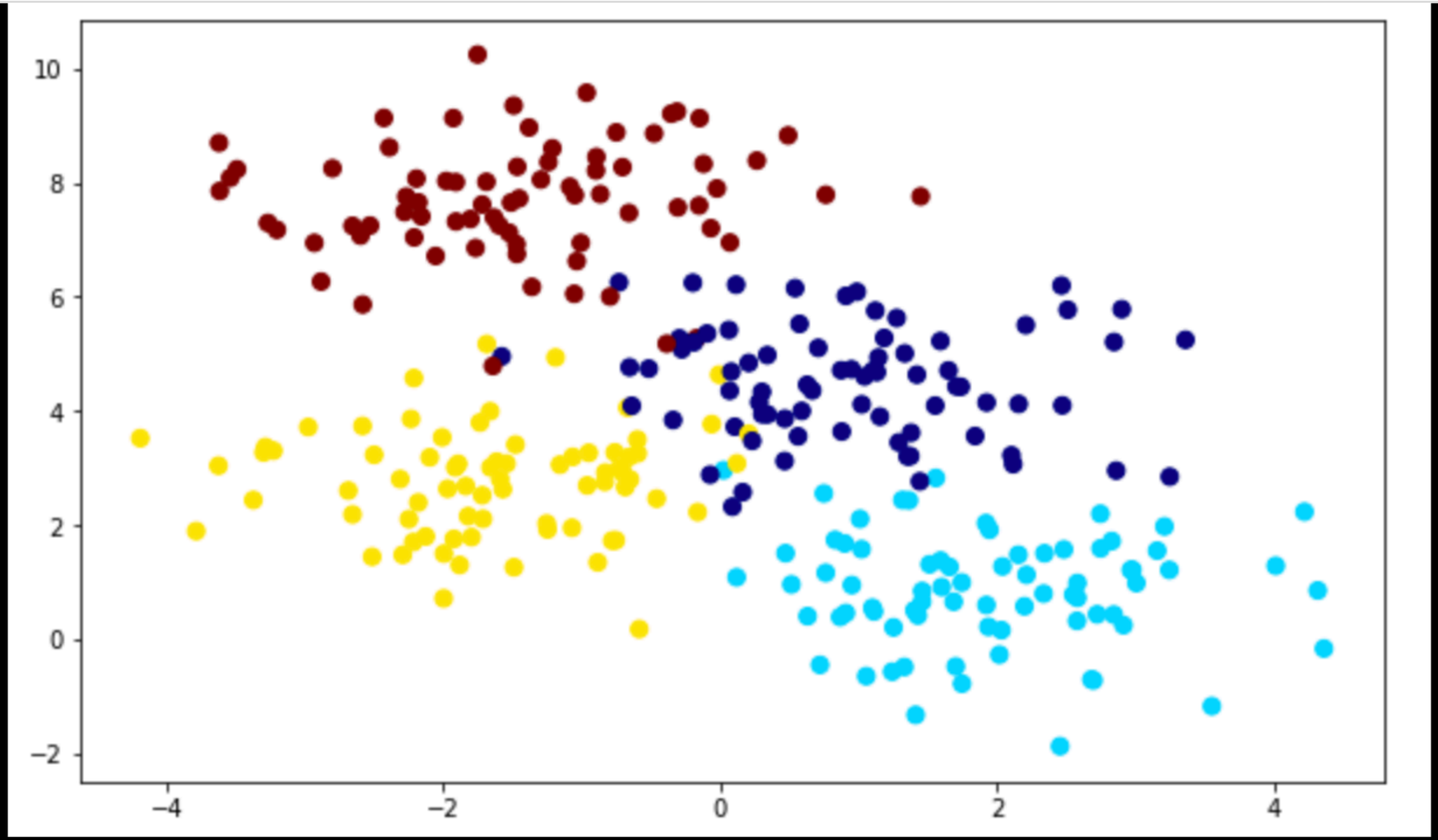
Decision Trees

- Very intuitive classification system
- Ask questions about predictors/features
- Zero-in on a classification label
- All splits are binary, with yes/no splits
- In the ideal case, each split cuts the number of options by half - this is not always the case, but there are ways to optimise the efficiency of a split

Decision Trees

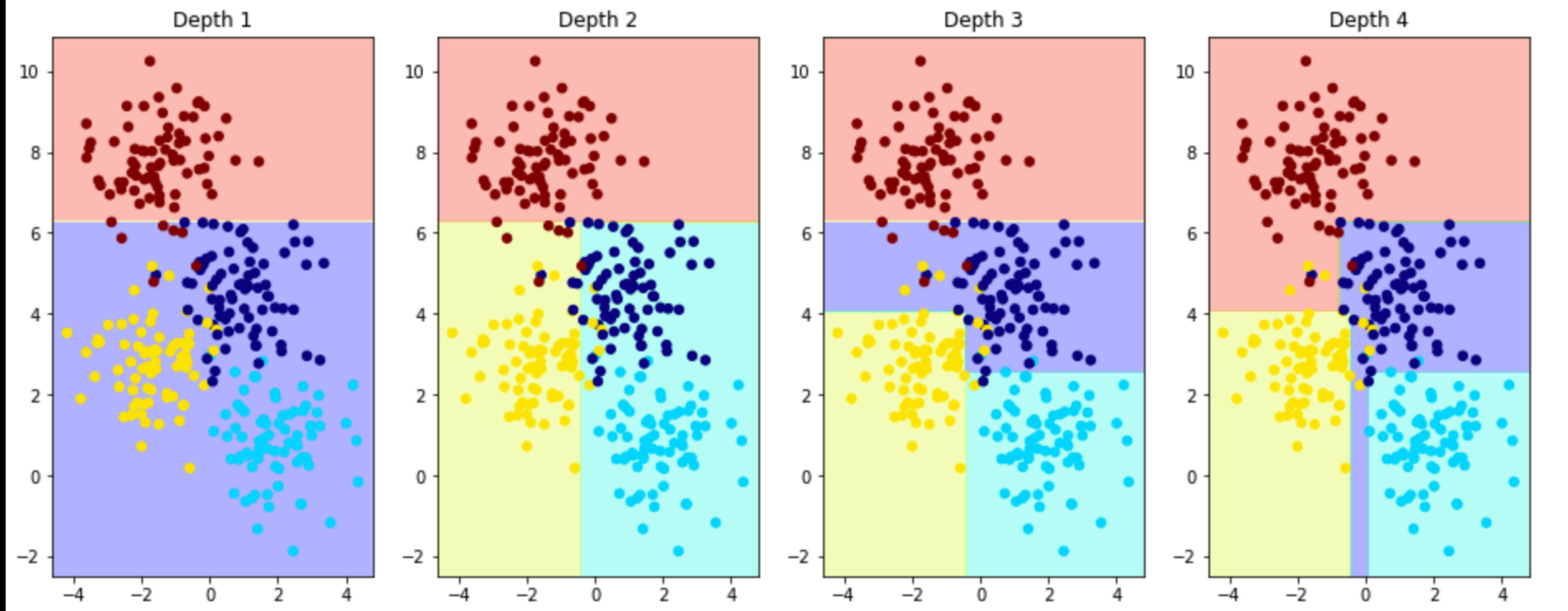
- Let's generate some fake data from 4 classes:

```
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');
```



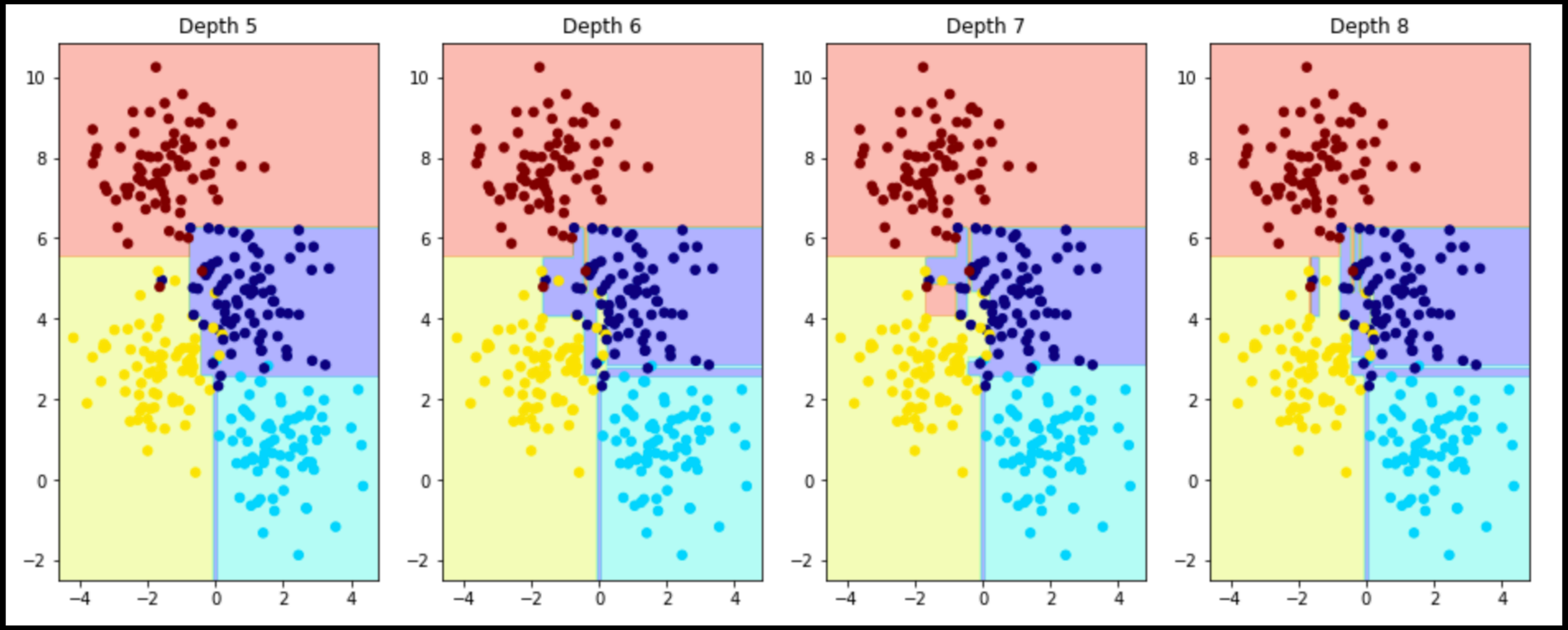
Decision Trees

```
33 for depth in [1,2,3,4]:
34     tree = DecisionTreeClassifier(max_depth=depth)
35     visualize_classifier(model=tree,X=X,y=y,ax=axes_depths[depth-1])
36     axes_depths[depth-1].set_title('Depth {0:d}'.format(depth))
```



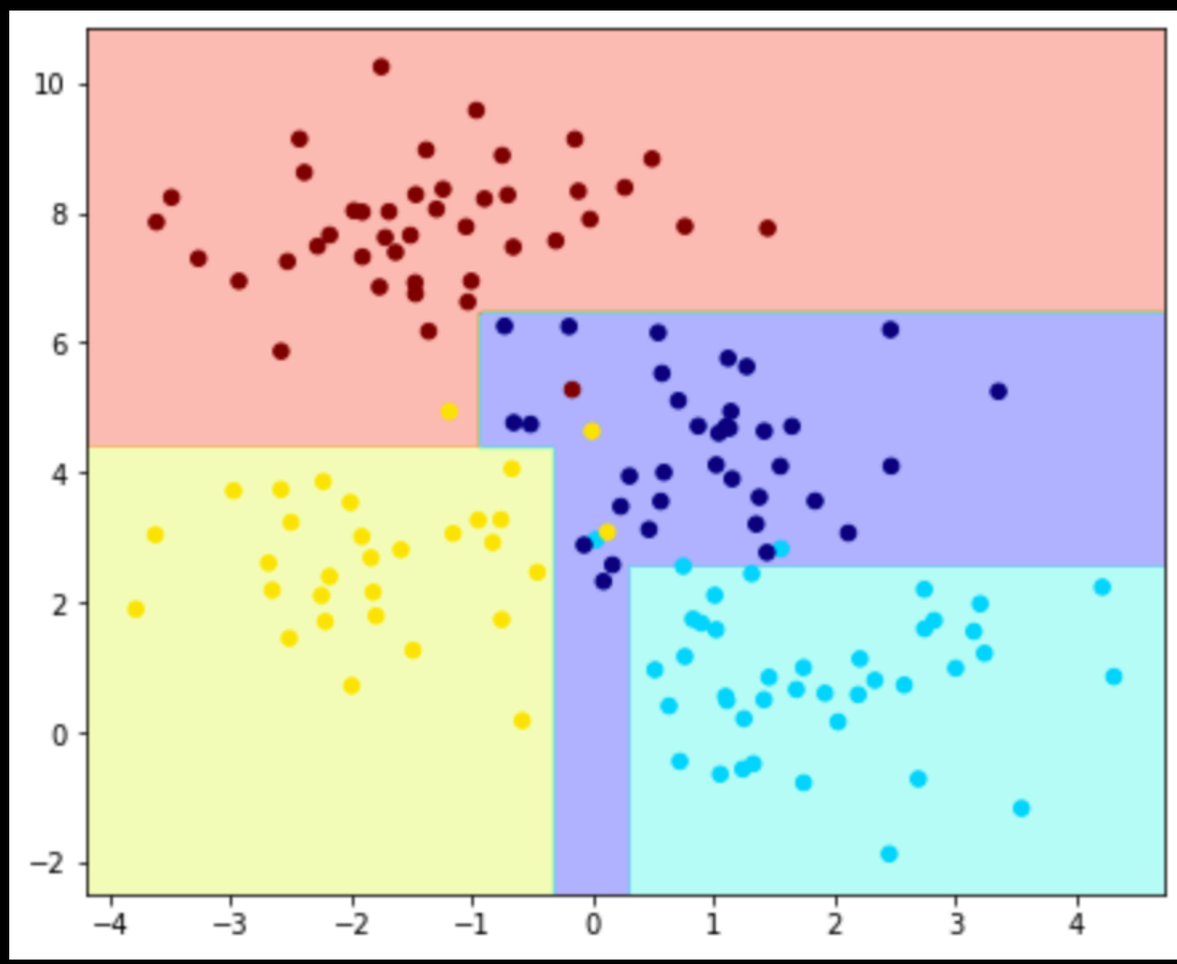
Decision Trees

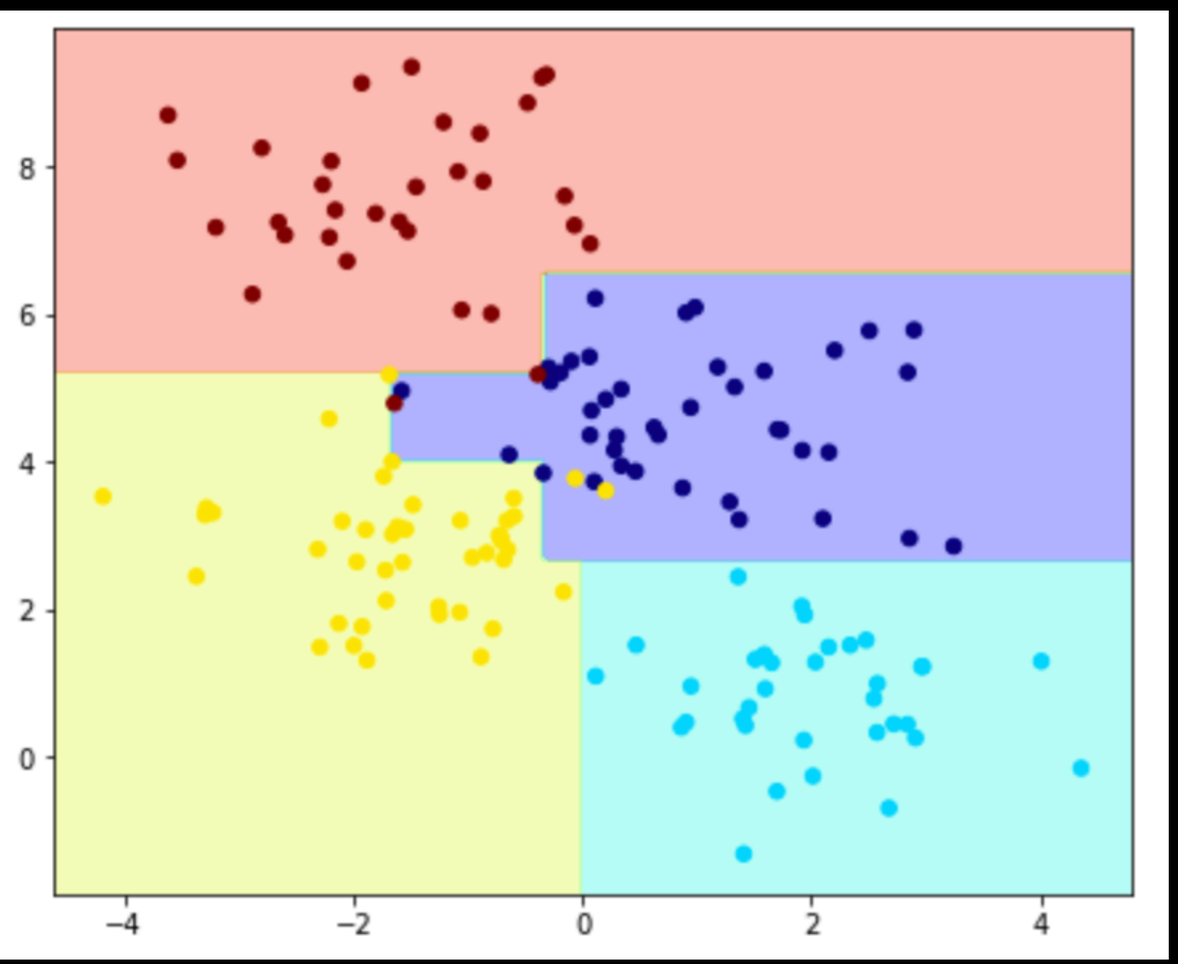
- What if we keep increasing depth? Overfitting!



Decision Trees

- It is very easy to overfit a decision tree - one way to keep track of this is to train the same tree depth onto subsets of your data





Decision Trees

- Applied to our Star vs LL-Rylae Problem:

```
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
```

```python
# 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 Trees

- But let's do a training/testing data split…

```
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
```

```
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 Trees

- Recursive splitting can keep going until there is a single point per node
- This is highly inefficient for both tree computation and traversal
- The common criterion applied is to split only if there is any information gain, a reduction in misclassifications, when a node is 100% pure, or when the number of points per node reaches some threshold

Ensembles of Classifiers

- The notion of combining overfitting estimators to reduce the effect of overfitting is what makes up an ensemble method called **bagging** (bootstrap aggregation)

- Average the results of a series of bootstrap samples from a training set

- Often applied to decision trees, but could be applied to other techniques as well

- For a sample of N training points, bagging generates K equally sized bootstrap samples on which to estimate the function:

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

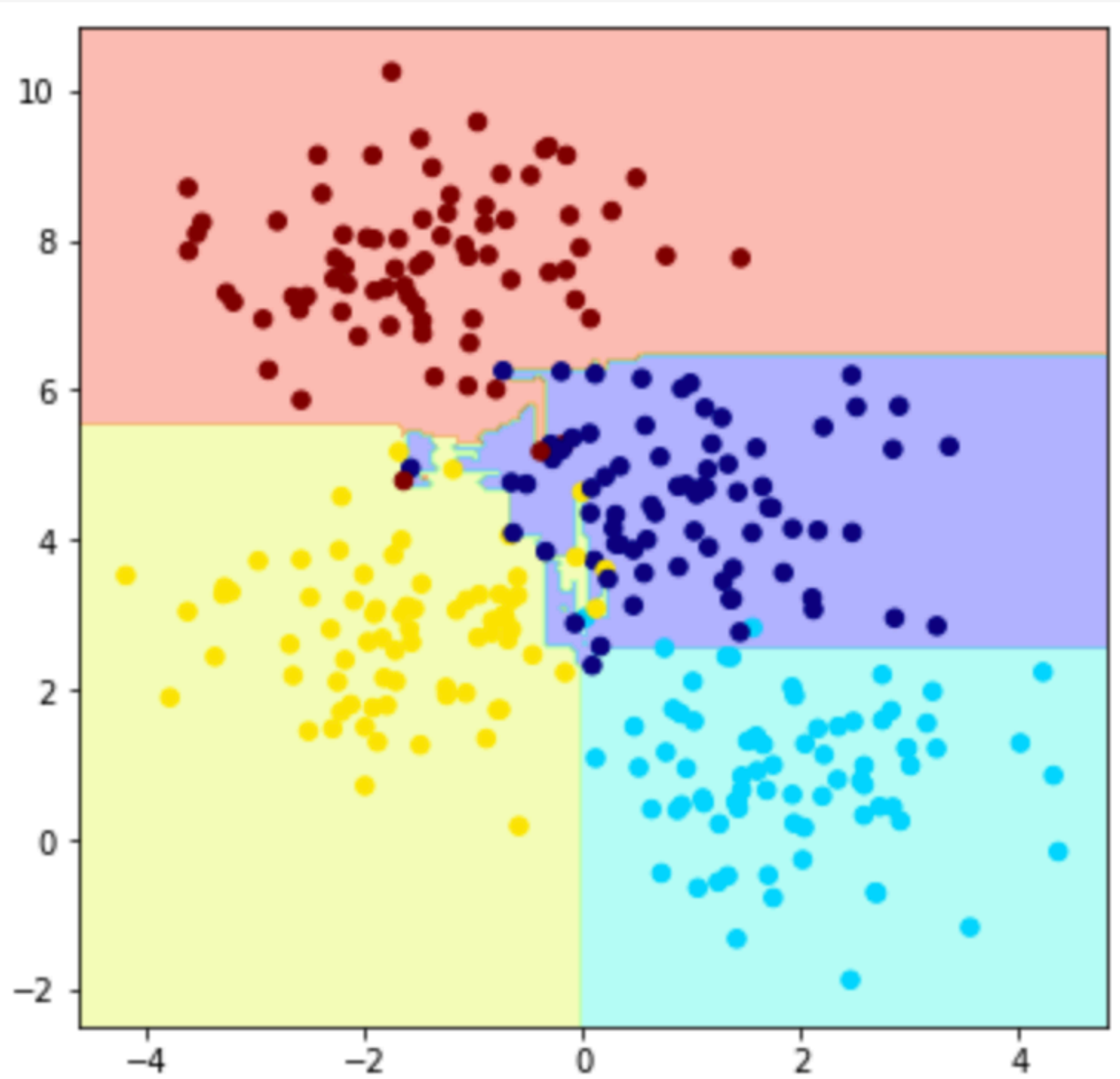
Ensembles of Classifiers

- Random forests extend the bagging idea by generating a set of decision trees from the 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 is based on averaging
- To generate random forest, we therefore need to decide on n, the number of trees to generate
- And m, the number of attributes to be considered in each tree

Ensembles of Classifiers

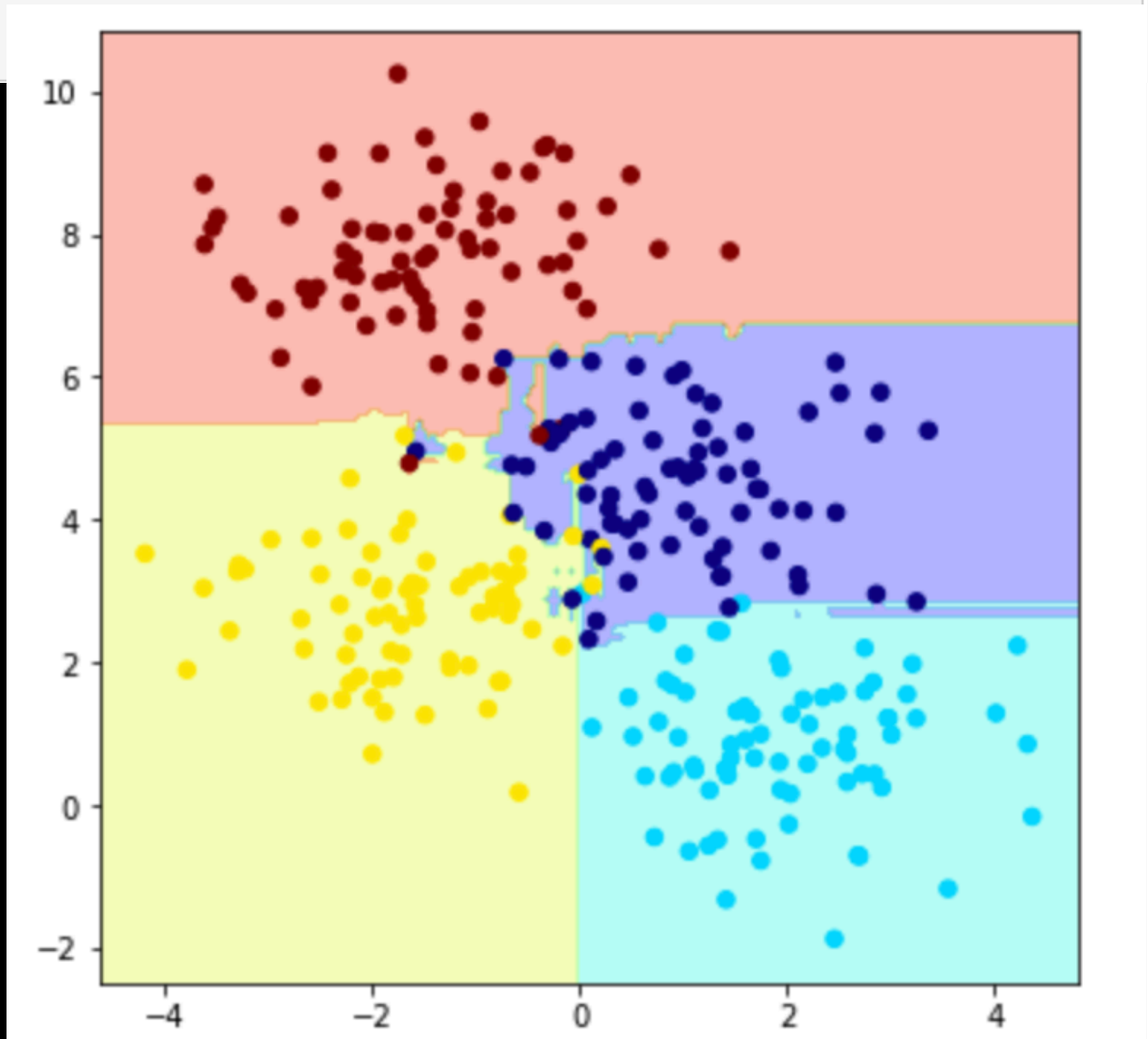
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import BaggingClassifier
from sklearn.datasets import make_blobs

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()
```

Bagging focuses on a split of samples (80%)



Ensembles of Classifiers

```
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()
```



Random Forests use all the samples but each tree works on a different feature set

Classifier Evaluation: ROC Curves

- Comparing different (and tuned) classifiers is an important part of building a ML system for a problem

- The ‘best’ classifier is subjective - completeness vs contamination tradeoff

- One way to visualise classification performance is via Receiver Operating Characteristic (ROC) Curves

- A ROC Curve shows the true-positive rate as a function of the false-positive rate as the discriminant function is varied

Classifier Evaluation: ROC Curves

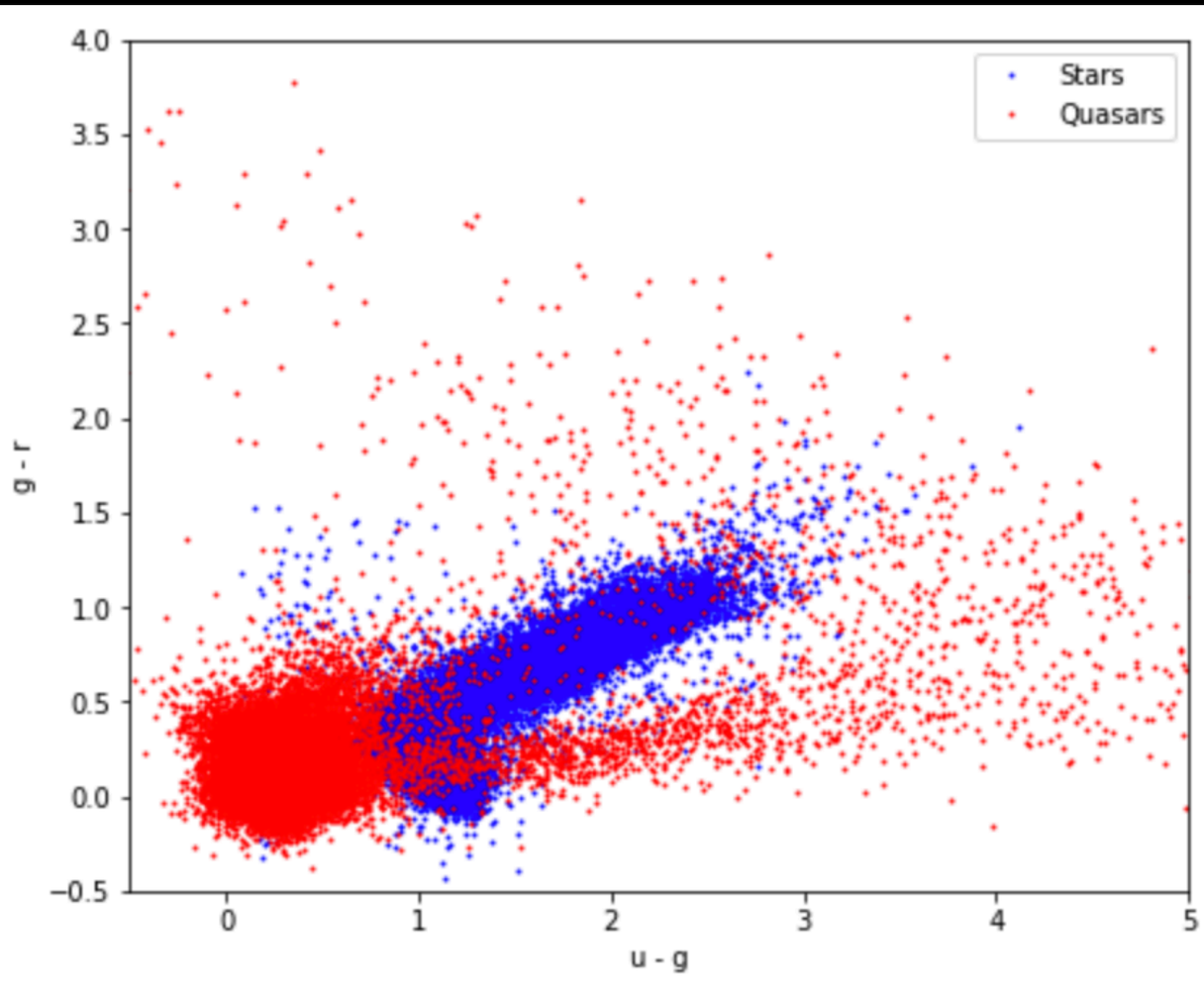
- How the function varies depends on the individual model
- In the Gaussian Naive Bayes example, the curve is drawn by classifying data using relative probabilities between 0 and 1
- Scikit-learn has all the tools you need to compute a ROC Curve

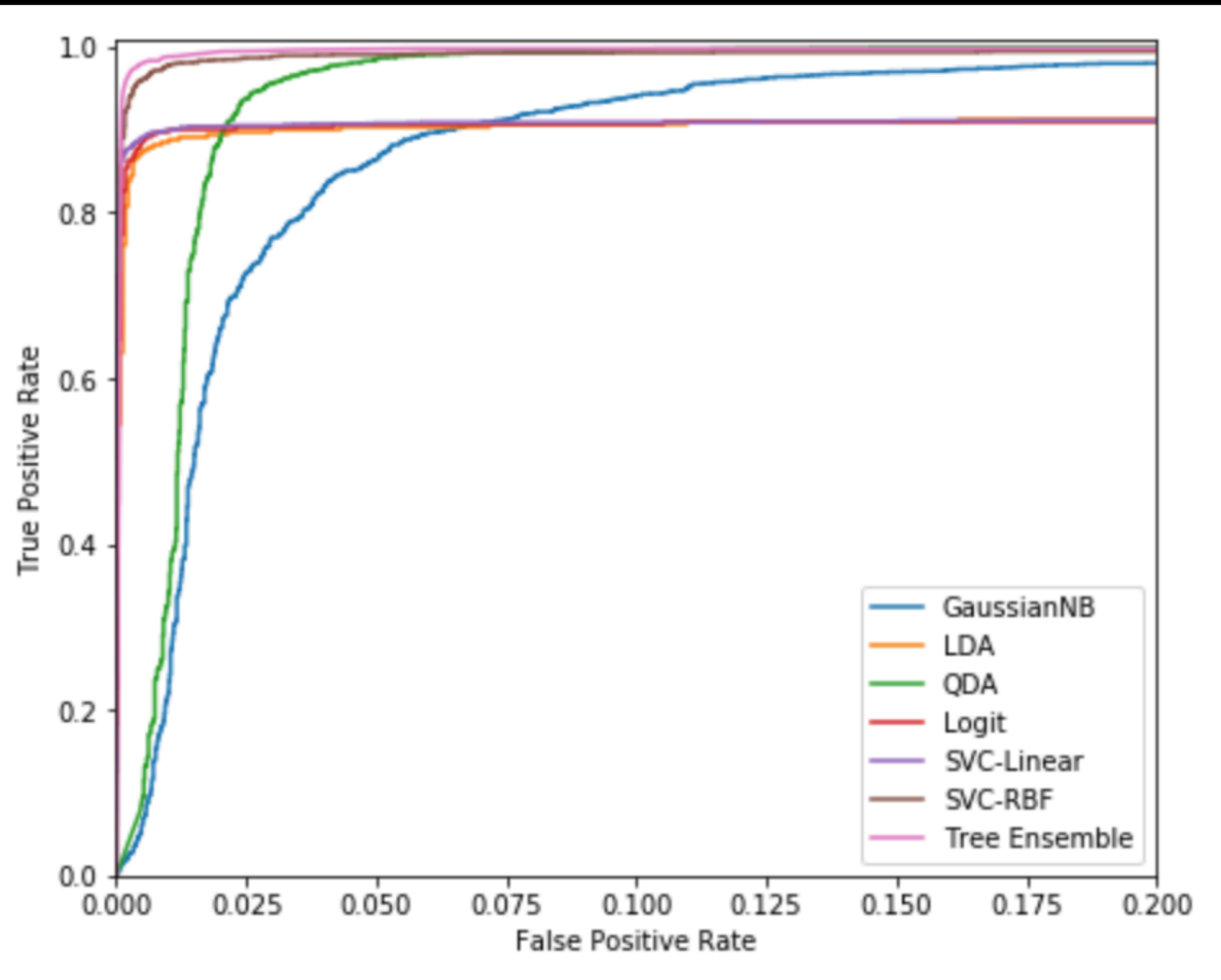
```
105 for i in range(0,len(names)):
106     classifier = names[i]
107     y_prob = probs[i]
108     fpr, tpr, thresholds = roc_curve(y_test, y_prob)
109     fpr = np.concatenate([[0], fpr])
110     tpr = np.concatenate([[0], tpr])
111     roc2.plot(fpr, tpr, label=classifier)
```

Classifier Evaluation: ROC Curves

- Interpretation of a ROC Curve is easy - the top left (full TP rate with zero FP rate) is the ideal classifier

- We want to be as close as possible to that point





Classification Recap

- Generative vs Discriminative Classification
  - Gaussian Naive Bayes
  - LDA/QDA
  - Logistic Regression
  - Support Vector Machines
  - Decision Trees and Random Forests
- Classification Loss and ROC Evaluation
- Overfitting and Ensemble Methods