Quantitative Applications for Data Analysis: classification

Erik Spence

SciNet HPC Consortium

17 March 2026

Today's slides

Today's slides can be found here. Go to the "Quantitative Applications for Data Analysis" page, under Lectures, "Classification".

`https://scinet.courses/1399`

SciNet

Today's class

Today we will visit the following topics:

- Logistic regression.
- ROC curves.
- kNN.
- Support Vector Machines.

Ask questions!

SciNet

Classification

Classification is similar to regression, in a sense:

- You fit a model to data with known answers ($y = f(x_1, x_2, x_3, ...)$).
- You use the model to make predictions about new data.

But what do you do if the labels (y) are discrete? How do you deal with that?

- Data point y is either in category 1 or 2.
- You don't get points for putting y in category 1.5.

Classification algorithms are used to create models for separating data into known categories.

SciNet

Classification problems

Classification problems are everywhere:

- Bioinformatics - classifying proteins according to function.
- Medical diagnosis.
- Image processing:
  - what objects exist in an image?
  - hand-written text analysis.
- Text categorization:
  - Spam filtering
  - Sentiment analysis: is this tweet positive or negative?
- Language recognition.
- Fraud detection.

Input variables can be continuous, discrete, or both.

SciNet

Classification approaches

There are lots of classification approaches which one might use.

- Decision trees: analyze the features of the data and make 'decisions' about how to 'split' the data into uniform groups.
- Logistic regression: like linear regression, but now we fit a "yes/no" function to the data.
- Naive Bayes: a type of probabilistic analysis.
- kNN: k Nearest Neighbours; use the k nearest neighbours to a data point to predict the category of a new data point.
- Support Vector Machines: essentially a linear model of the data, used for separate groups.
- Neural networks: a weird algorithmic approach to using functions to categorize data.

Today we will go over logistic regression and kNN.

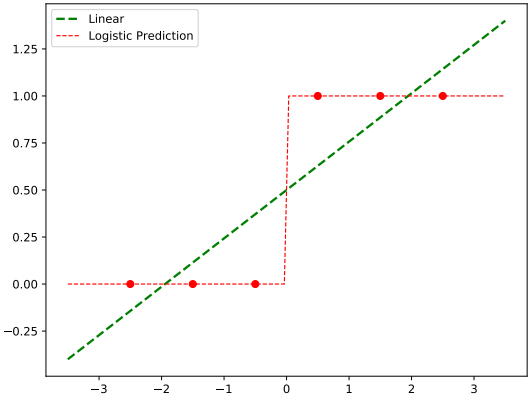
SciNet

Logistic regression

One way to consider binary classification is to go back to regression, and fit a linear regression to an integer 0/1 variable for classification: over 0.5, True, else False.

This requires a linear separation between the classes to be effective.

However, naive application of linear regression can lead to a number of problems, which grow with the number of dimensions. These are mostly related to the unbounded nature of the function.



Logistic regression, continued

A whole infrastructure exists for "generalized linear models", where the function being fit is not

$$y = \beta_0 + x_1\beta_1 + x_2\beta_2 + \cdots = \mathbf{x} \cdot \vec{\beta}$$

but rather some power or exponential of $\mathbf{x} \cdot \vec{\beta}$.

Consider instead fitting, not the probability p, but rather the log of the odds ratio,

$$\mu = \ln\left(\frac{p}{1-p}\right) = \mathbf{x} \cdot \vec{\beta}$$

We can fit this log-odds equation, and derive

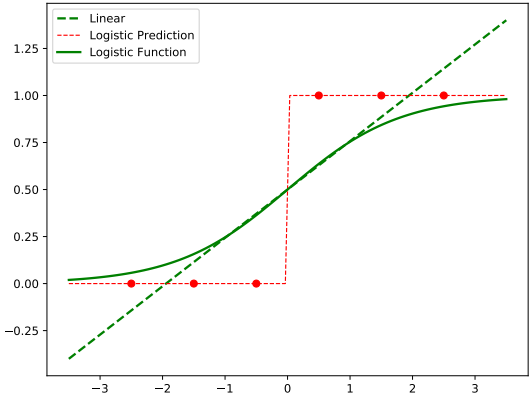
$$p = \frac{e^{\mathbf{x}\cdot\vec{\beta}}}{1 + e^{\mathbf{x}\cdot\vec{\beta}}} = \frac{1}{1 + e^{-\mathbf{x}\cdot\vec{\beta}}}$$

Logistic regression, continued more

$$p = \frac{1}{1 + e^{-\mathbf{x} \cdot \vec{\beta}}}$$

This approach has a number of very nice properties:

- We have a nice, bounded, well-behaved function.
- We can directly calculate the inferred probability of category membership.
- We're essentially fitting a Bernoulli process.

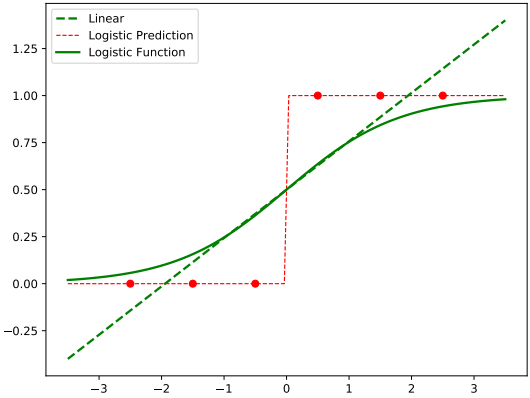


Logistic regression, continued some more

One has to use somewhat different numerical algorithms to fit these curves; typical curve-fitting algorithms deal very poorly with exponentials.

Techniques like expectation maximization (EM) or other well-conditioned iterative methods are often used.

That's fine; they're all hidden beneath whatever logistic or GLM packages you might want to use.



SciNet

Logistic regression, example

Using logistic regression in sklearn is as simple as you might expect.

- From the linear_model subpackage, create a LogisticRegression object.
- Fit in the usual way.
- Note that logistic regression can be used for more than just binary classification.

```
In [1]: import sklearn.datasets as skd
In [2]: import sklearn.model_selection as skms
In [3]: import sklearn.linear_model as sklm
In [4]:
In [4]: data = skd.load_breast_cancer()
In [5]:
In [5]: train_x, test_x, train_y, test_y = \
...:         skms.train_test_split(data.data, data.target,
...:                               test_size = 0.2)
In [6]:
In [6]: model = sklm.LogisticRegression(max_iter = 10000)
In [7]:
In [7]: model = model.fit(train_x, train_y)
In [8]:
In [8]: test_pred = model.predict(test_x)
In [9]:
```

Confusion matrix

How you determine the effectiveness of a classifier is different than a regression. You can count the number incorrectly classified, and this useful, but it doesn't give you much information you can use to improve the result.

The 'Confusion Matrix', tells you which misclassifications happened. Traditionally, 'true' classifications are on the rows, and predictions are on the columns.

```
In [9]: import sklearn.metrics as skm
```

```
In [10]:
```

```
In [10]: skm.accuracy_score(test_y, test_pred)
Out[10]: 0.956140350877193
```

```
In [11]:
```

```
In [11]: skm.confusion_matrix(test_y, test_pred)
Out[11]:
array([[ 41,  2],
       [  3, 68]])
```

Evaluating binary classifiers

Binary classification is a common and important enough special case that its confusion matrix elements have special names, and various quality measures are defined.

	Classified Positive (CP)	Classified Negative (CN)
Actual Positive	True Positive (TP)	False Negative (FN)
Actual Negative	False Positive (FP)	True Negative (TN)

One can always get exactly one of FN or FP to be zero (for example, just classify everything positive, then there will never be any false negatives).

But there is usually a trade-off between false positives and false negatives.

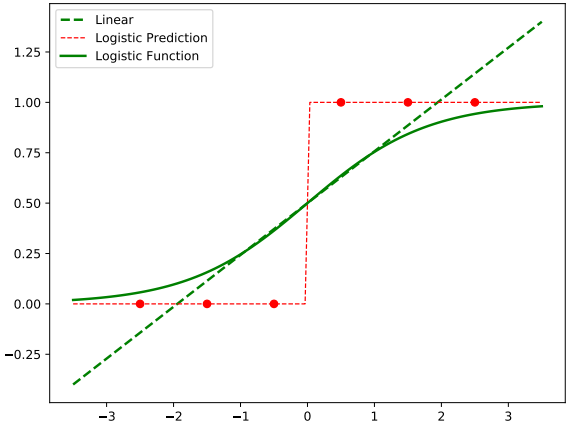
SciNet

Classification thresholds

In most binary classifiers, there's some equivalent of a threshold you can set. This threshold determines when a given data point moves from one categorization to the other.

For the case of logistic regression, the default threshold is 0.5.

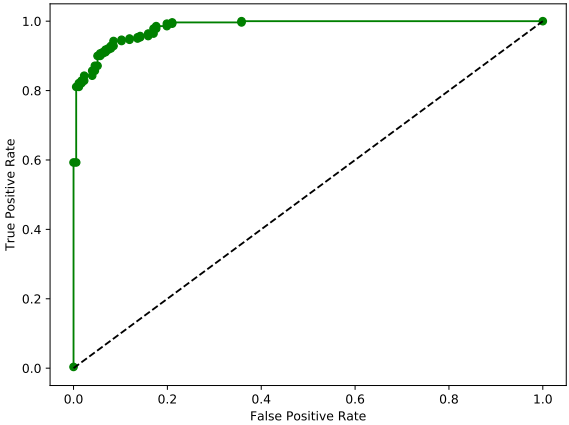
- Set it lower (allow more true, but also false, positives).
- Set it higher (allow more true, but also false, negatives).



ROC curve

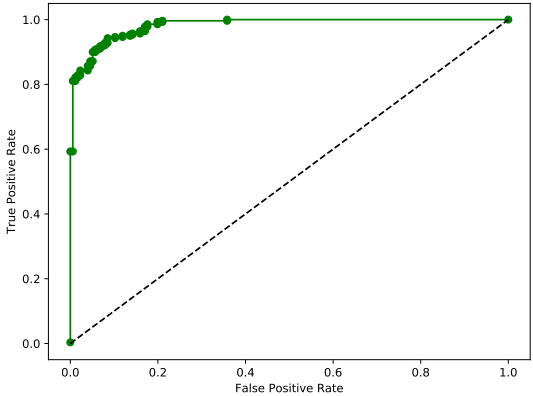
By varying the classification threshold, from 1 to 0, we can get a collection of points for the TPR and FPR. Plotting the two measures on either axis gives a ROC (Receiver Operating Characteristic) curve.

- The diagonal line represents random chance.
- We want our curve to be as high above the diagonal as possible.



ROC curve, continued

```
In [12]:
In [12]: probs = model.predict_proba(test_x)
In [13]:
In [13]: fpr, tpr, _ = skm.roc_curve(test_y,
...:                                 probs[:,1])
In [13]:
In [13]: import matplotlib.pyplot as plt
In [14]:
In [14]: plt.plot(fpr, tpr, 'go-')
In [14]:
```



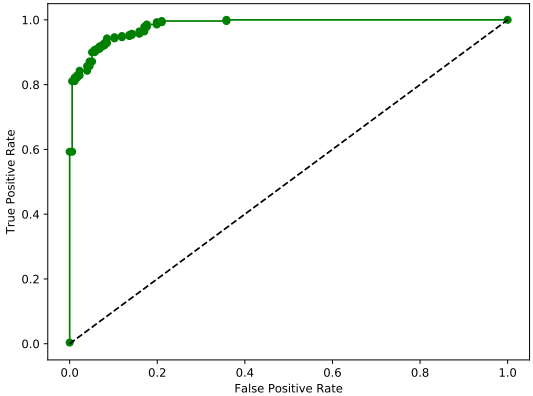
Note that your curve will look different from this one, due to randomness.

ROC curve, continued more

The quality of a classifier is determined by the ROC curve's AUC (area under the curve).

- The worst classifiers will have an AUC near 0.5.
- Good classifiers have an AUC near 1.0.

For the non-binary classification situation, you create "one versus all" ROC curves, with one ROC curve for each category.



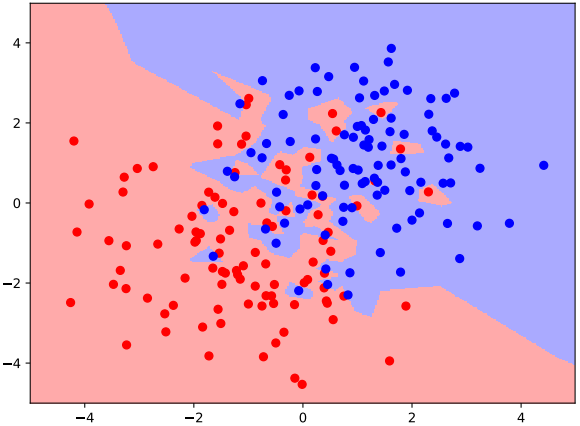

```
In [14]:
In [14]: skm.roc_auc_score(test_y, probs[:,1])
Out[14]: 0.9854096520763187
In [15]:
```

Nearest neighbours - kNN

Consider a more-geometric approach to classification: given an input data point, find the nearest point in the training set, and choose that classification for your input data point.

This is a type of regression.

A generalization is to choose the k Nearest Neighbours (kNN), and choose the classification that the majority of those k points has.



Two 2D Gaussians, centred on (-1,-1) (red), and (1,1) (blue) with $\sigma = 1.5$, $k = 1$.

Nearest neighbours - kNN, continued

```
# knndemo.py
import numpy as np, matplotlib.pyplot as plt
from scipy.stats import norm
import sklearn.neighbors as skn

num0 = 200;      num = int(num0 / 2)
c1 = -1.0;       c2 = 1.0;       sig1 = 1.5

# Generate Gaussian data.
x1 = norm.rvs(size = num, loc = c1, scale = sig)
y1 = norm.rvs(size = num, loc = c1, scale = sig)
x2 = norm.rvs(size = num, loc = c2, scale = sig)
y2 = norm.rvs(size = num, loc = c2, scale = sig)

# Set up the data.
z1 = np.c_[x1, y1];       z2 = np.c_[x2, y2];
x = np.concatenate((z1, z2))
y = np.concatenate((np.zeros(num), np.ones(num)))
```

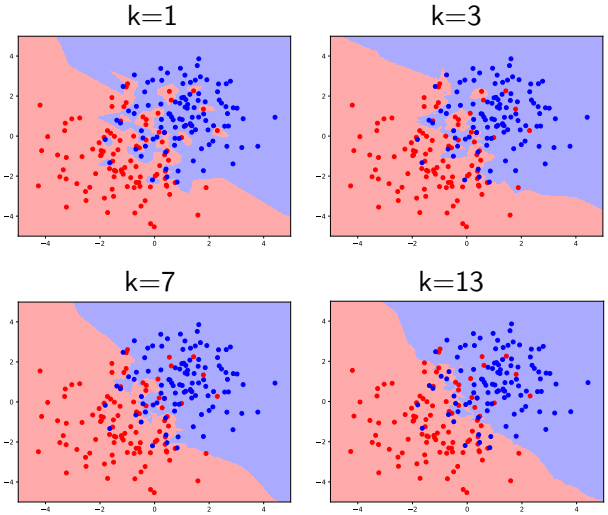
```
# Set up the background grid.
xx, yy = np.meshgrid(np.arange(-5, 5, 0.01),
                     np.arange(-5, 5, 0.01))

# Build the model, and train.
model = skn.KNeighborsClassifier(1)
model.fit(x, y)

# Predict the values for the background.
z = model.predict(np.c_[xx.ravel(), yy.ravel()])

# Put the result into a colour plot.
z = z.reshape(xx.shape)
plt.pcolormesh(xx, yy, z)

# Plot also the training points, and save.
plt.scatter(x[:, 0], x[:, 1], c = y)
plt.savefig('knndemo_k=' + str(k) + '.pdf')
```

Bias-variance in kNN

There's a bias-variance-like trade-off in kNN, as can be seen by varying k on the same data.

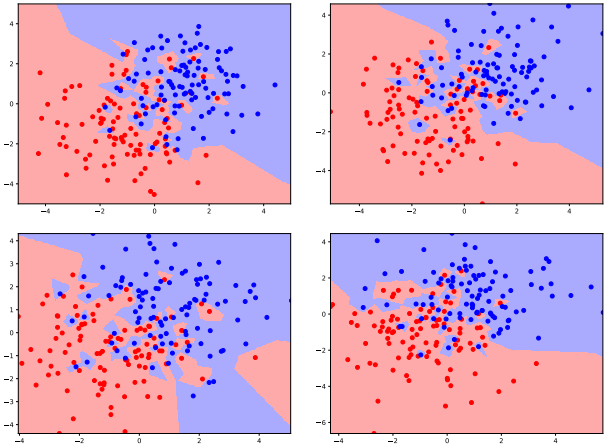
At low k, the variance is very large. The model is trying to fit to every single point.

At higher k, we average over a large area, and we start to lose features.



Bias-variance in kNN, continued

On the right we see 4 instances of the previous data set. The model has been built with $k = 1$ for all 4. It's clear that the decision boundary varies widely from one run to the next.



Scaling continuous features

In the iris data set, petal length varies over a much greater range than sepal width. If we just use Euclidean distance for kNN, sepal width will provide very little information: all points are close to each other in that dimension.

We want the information in all features to contribute to the solution. To this end, we should scale the features to that they all get to play. A common technique is to centre the features by subtracting off their means, and then scaling them by their standard deviations.

$$x' = \frac{x - \mu}{\sigma_x}$$

Many libraries will do this for you, for methods where it matters. But not all will; check the documentation!

Cross-validation and kNN

But we are left with the same (or similar) problem as the polynomial fitting: how do we choose the value of k?

The sklearn package has built-in functionality to perform cross-validation on a kNN analysis.

Let's try this on the output of the make_classification function. By default this function creates 20 features.

```
In [15]:
In [15]: import sklearn.datasets as skd
In [16]:
In [16]: x, y = skd.make_classification(500,
...:             n_classes = 3,
...:             n_informative = 4)
In [17]:
In [17]: x.shape
Out[17]: (500, 20)
In [18]:
In [18]: train_x, test_x, train_y, test_y = \
...:         skms.train_test_split(x, y,
...:                               test_size = 0.2)
In [19]:
```

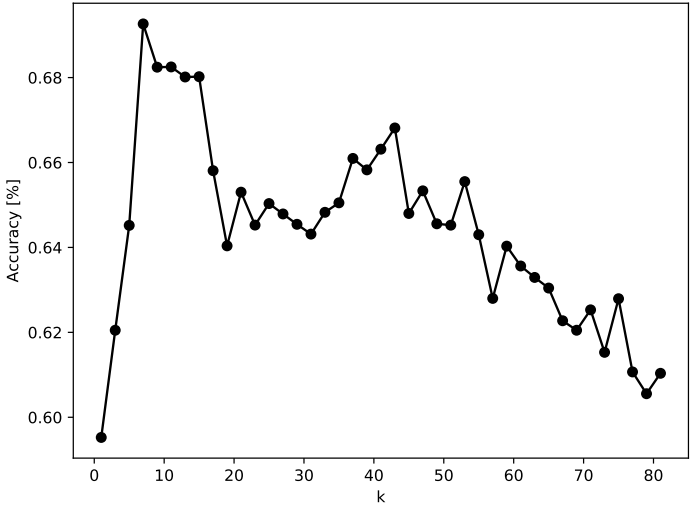
Cross-validation and kNN, continued

How do we use the sklearn cross-validation?

- Create a KNeighborsClassifier object.
- Use the cross_val_score function to perform the cross-validation for you.
- The function returns the scores for each k fold.
- Examine the scores to find the best k value.

```
In [19]:
In [19]: import sklearn.neighbors as skn
In [20]: import numpy as np
In [21]:
In [21]: kvalues = range(1, 82, 2)
In [22]: scores = np.zeros(len(kvalues))
In [23]:
In [23]: for i, k in enumerate(kvalues):
...:         model = skn.KNeighborsClassifier(k)
...:         scores[i] = np.mean(skms.cross_val_score(model,
...:                             train_x, train_y, cv = 10))
In [24]:
In [24]: plt.plot(kvalues, scores, 'ko-')
In [25]:
```

Cross-validation and kNN, continued more



Cross-validation and kNN, continued even more

Unfortunately, cross_val_score does not return the best model. You need to recalculate that yourself.

As always, it's a good idea to see how well the algorithm works, and make sure the errors are balanced.

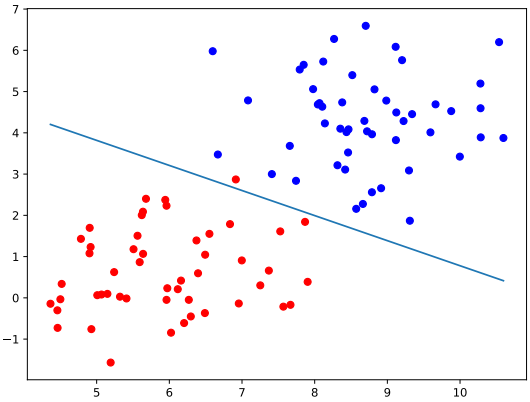
```
In [25]:
In [25]: bestmodel = skn.KNeighborsClassifier(7)
In [26]:
In [26]: bestmodel = bestmodel.fit(train_x, train_y)
In [27]:
In [27]: test_pred = bestmodel.predict(test_x)
In [28]:
In [28]: skm.confusion_matrix(test_y, test_pred)
array([[ 32,  0,  3],
       [  8, 17,  4],
       [ 12,  7, 17]])
In [28]:
In [28]: skm.accuracy_score(test_y, test_pred)
Out[28]: 0.66
In [29]:
```

Support Vector Machines

Linear Support Vector Machines (also called Support Vector Classifiers) determine a hyperplane which linearly separates the data set into distinct categories.

The goal of the algorithm is to find the plane that is the farthest from all the closest points, in a k dimensional space.

This just ends up being a minimization problem.



Support Vector Machines, example

To demonstrate the use of Support Vector Machines we will use our old friend, the iris data set.

Note that the iris data set has 4 features. Hence, the hyper-plane which passes through the data will be 3D.

```
In [29]: import sklearn.datasets as skd
In [30]: import sklearn.model_selection as skms
In [31]:
In [31]: iris = skd.load_iris()
In [32]:
In [32]: train_x, test_x, train_y, test_y = \
...:         skms.train_test_split(iris.data,
...:                               iris.target,
...:                               test_size = 0.2)
In [33]:
In [33]: train_x.shape
Out[33]: (120, 4)
In [34]:
```

Support Vector Machines, example, continued

Support Vector Machines take an optional argument, 'C' (the penalty parameter).

- Large values of C mean that we lack confidence in the data's distribution (noisy data).
- Small values mean the opposite.
- Default value is 1.0.
- By default the model does not calculate the probabilities associated with each category, which is needed to create a ROC curve.

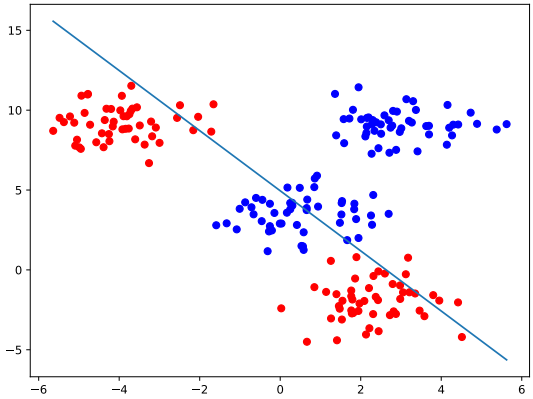
```
In [34]: import sklearn.svm as svm
In [35]: import sklearn.metrics as skm
In [36]:
In [36]: model = svm.SVC()
In [37]:
In [37]: model = model.fit(train_x, train_y)
In [38]: test_pred = model.predict(test_x)
In [39]:
In [39]: skm.confusion_matrix(test_y, test_pred)
Out[39]:
array([[ 8,  0, 0],
       [ 0, 12, 1],
       [ 0,  0, 9]])
In [40]:
In [40]: skm.accuracy_score(test_y, test_pred)
Out[40]: 0.9666666666666667
```

SciNet

Nonlinear Support Vector Machines

Linear Support Vector Machines are all well and good if your data are linearly separated. But what can we do if we aren't so lucky?

As we can see in the example at right, you can imagine situations where there are clearly defined clusters of data, but fitting linearly is not an option.



SciNet

Nonlinear Support Vector Machines, continued

We have a technique which is quite good at finding hyperplanes in linearly separated data.

- If the data are not linearly separated, the solution, obviously (!), is to nonlinearly transform the data into a space where it is linearly separable.
- This typically involves adding dimensions to the data which did not previously exist.
- This increases the likelihood of making things linearly separable (Cover's theorem).

Great! But how do we figure out what transformation to apply to the data?

- We don't. We let the SVM kernel do the work for us.
- SVMs use 'kernel functions' to transform the data into the required form.
- There are many types of kernel functions available: linear (which we've already been using), polynomial, radial basis function (RBF), sigmoid, and others.
- Once the hyperplane has been determined in the transformed space, we can use the "decision_function" of the model to make predictions.

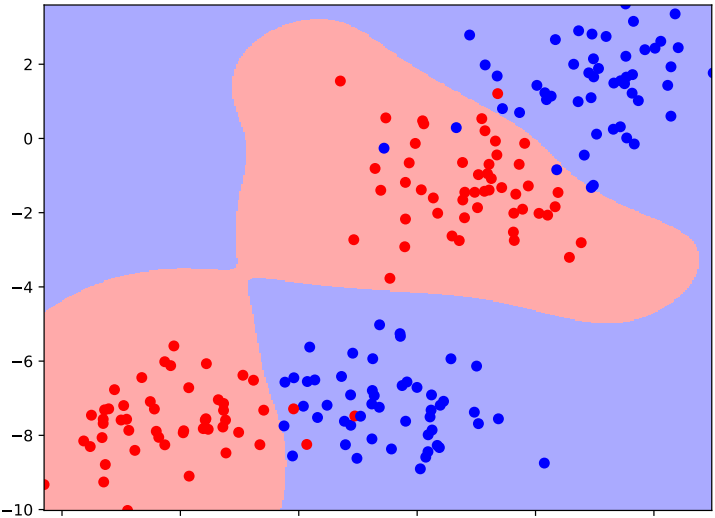
SciNet

Nonlinear SVM, example

```
# NLsvm.py
import numpy as np
import sklearn.datasets as skd
import sklearn.svm as svm

num = 100

# Generate 2 sets of blob data.
x1, y1 = skd.make_blobs(num,
  centers = 2)

x2, y2 = skd.make_blobs(num,
  centers = 2)

# Set up the data.
x = np.r_[x1, x2]
y = np.r_[y1, y2]
```

```
# Set up the background grid.
xx, yy = np.meshgrid(np.arange(np.min(x), np.max(x), 0.03),
                     np.arange(np.min(x), np.max(x), 0.03))

# Build the model, and train.
model = svm.NuSVC(kernel = 'rbf')
model = model.fit(x, y)

# Predict the values for the background.
z = model.decision_function(np.c_[xx.ravel(), yy.ravel()])

# Put the result into a colour plot.
z = z.reshape(xx.shape)
plt.pcolormesh(xx, yy, z)

# Plot also the training points, and save.
plt.scatter(x[:, 0], x[:, 1], c = y)
plt.savefig('NLsvmdemo.pdf')
```

Nonlinear SVM, example, result



Summary

You've now seen four classification algorithms: decision trees, logistic regression, kNN and SVM. Some things to remember:

- logistic regression strength: not prone to over-fitting.
- logistic regression strength: can work well with noisy data.
- logistic regression weakness: assumes there is a single smooth boundary between categories.
- kNN strength: works in as many dimensions as you like.
- kNN weakness: slow if there are too many data points.
- kNN weakness: doesn't handle categorical data.
- SVM strength: versatile, can be used in many dimensions.
- SVM weakness: can be slow in higher dimensions.

There are guidelines you can use, but ultimately experience and experimentation is most important.

SciNet