2019 CBP Symposium
Introduction to Machine Learning with Python

Marcelo Ponce

May 3, 2019

SciNet

Material for this workshop

The slides and code for this workshop can be found at this workshop's website, which is here:

`https://support.scinet.utoronto.ca/education/go.php/440/index.php`

More infromation about SciNet's courses is available at the SciNet education website:

`courses.scinet.utoronto.ca`

Python for today's workshop

A few notes about the code and examples for today's class.

- The code for today's class will by in Python 3.
- As you are hopefully aware, Python 2.7.X and Python 3.X are not completely compatible with each other.
- The differences are, on the whole, just here and there. You just sort of need to know what has changed.
- If you are using Python 2 there is a chance that some of today's code will not work for you.
- You will need the usual machine learning packages: numpy, scipy, matplotlib, scikit-learn.

Slowly but surely the world is transitioning to Python 3.

Python plotting for today's workshop

Note that some of the code in today's slides will be assuming that we're working in interactive plotting mode. How do we set that up?

- If you are using IPython from the command line, then invoke it with the --pylab flag.

```
myuser@mycomp ~> ipython --pylab
In [1]:
```

- If you are using regular Python, you can turn on interactive plotting through matplotlib.

```
myuser@mycomp ~> python
>>> import matplotlib.pylab as plt
>>> plt.ion()
>>>
```

- If you are using Anaconda or Spider, there are options in the pull-down menus to make the prompt interactive.
- My recommendation is that you use the jupyter-notebook interface `jupyter.org`

Today's Topics

Today we will visit the following topics:

- Introduction to machine learning.
- Regression.
- Bias-variance tradeoff.
- Resampling.
- Classification algorithms, in general.
- Decision trees.
- kNN.
- k-means.
- Agglomerative clustering.

With material from L. Dursi and E. Spence. Ask questions!

What is machine learning?

What is machine learning?

- Broadly speaking, machine learning is model fitting.
- In some ways this is identical to data analysis, if your data analysis involves
  - fitting curves to your data.
  - determining parameters within an already-established model.
- But it can differ from simple data analysis:
  - if you don't know what the correct model is.
  - if you're just using the model to make insights to the data, but aren't looking for any scientific insight based upon it.

This is particularly useful at the beginning of a research program, when performing exploratory data analysis is the norm.

Supervised and unsupervised learning

When we're working with data, we generally have two types of analyses:

- Supervised: the data comes labelled with the right answer:
  - curve fitting.
  - for prediction-type analyses (decision trees, neural networks,...)
- Unsupervised: we're looking for patterns in the data:
  - what groups of items in this dataset are similar? Dissimilar?
  - Generally used for exploration, evaluation and sometimes prediction.
- There is also semi-supervised, but we won't be dealing with that.

Types of Data

Generally speaking, data comes in two broad classes:

- Continuous: real numbers
- Discrete:
  - Binary: True/False.
  - Categorical: category A, category B, ...
  - Ordinal: discrete, but has an intrinsic order: S, M, L, XL, ...
  - Numerical: numbers with a minimum step size (financial tick data).

Others are possible too, but we won't be covering them.

Regression

Let's begin reviewing our old friend, regression.

- It's familiar; a good place to start.
- Data comes as a set of n observations, each of which has p "features".
- We will assume continuous features (not always the case).
- The goal is to learn the function $y = \hat{f}(x_1, x_2, ..., x_p)$ for predicting new values.

Least Squares

One way to fit a functional form to some data is Ordinary Least Squares (OLS):

- Fit some $\hat{y} = \hat{f}(\mathbf{x}; \boldsymbol{\theta})$ to some data $(\mathbf{x}, \boldsymbol{y})$ which minimizes the squared error.
- This means we are assuming that the data is generated by some true function $\boldsymbol{f}$

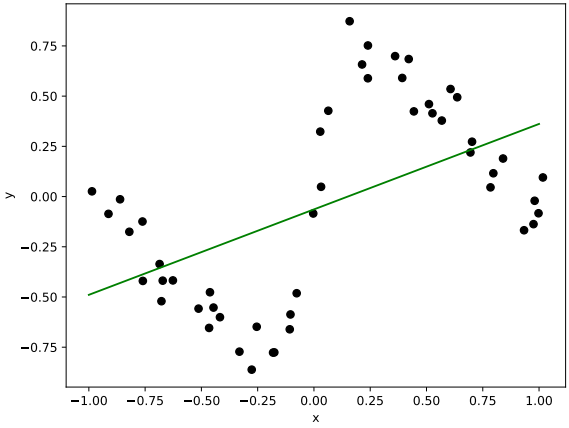
$$\boldsymbol{y} = \boldsymbol{f}(\mathbf{x}) + \boldsymbol{\epsilon}$$

- $\boldsymbol{\epsilon}$ is some error which can't be avoided.
- We choose $\boldsymbol{\theta}$ such that

$$\hat{\boldsymbol{\theta}} = \mathbf{argmin}_{\boldsymbol{\theta}} \sum_{\boldsymbol{i}} \left(\boldsymbol{y_i} - \hat{\boldsymbol{f}}(\boldsymbol{x_i}, \boldsymbol{\theta}) \right)^{\mathbf{2}}$$

Generate some data, and fit

```
In [1]: import regression as reg
In [2]: import numpy as np
In [3]:
In [3]: n = 50
In [4]:
In [4]: x, y = reg.noisy_data(n)
In [5]: x2 = np.linspace(-1, 1, 100)
In [6]:
In [6]: p = np.polyfit(x, y, 1)
In [7]: fit = np.poly1d(p)
In [8]:
In [8]: plt.plot(x, y, 'ko')
In [9]: plt.plot(x2, fit(x2), 'g-')
In [10]:
```

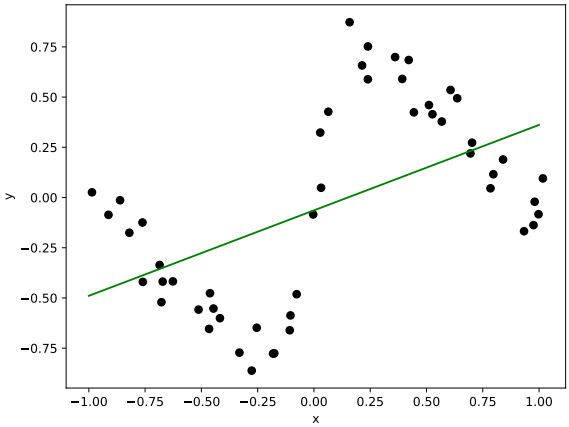


Example inspired by L. Dursi.

Regression

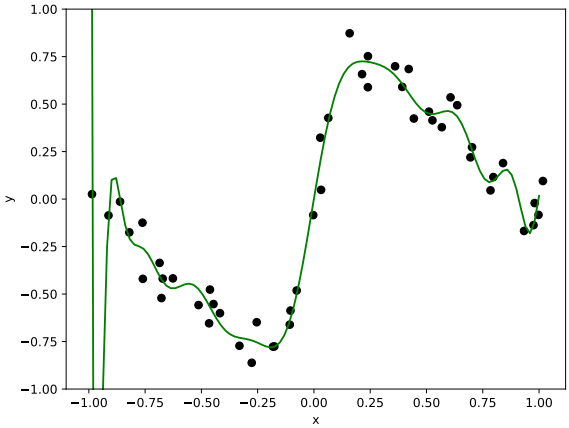
We've got our data, and we've starting the process of fitting it. What questions should we be asking?

- How well is this fit likely to perform on new data?
- How accurate is the fit at any given point ($x = 0$)?
- How robust is this fit? Will it vary significantly given new data?



Repeat with degree 20

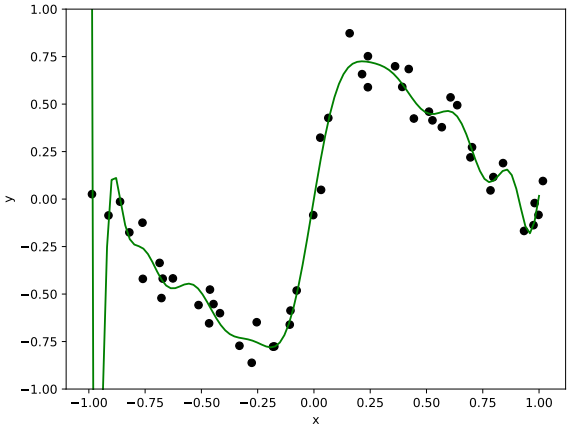
```
In [10]:
In [10]: p20 = np.polyfit(x, y, 20)
In [11]: fit20 = np.poly1d(p20)
In [12]:
In [12]: plt.plot(x, y, 'ko')
In [13]: plt.plot(x2, fit20(x2), 'g-')
In [14]:
```



Regression, with degree 20

We've obviously gotten much better accuracy. Let's ask the same questions as before:

- How well is this fit likely to perform on new data?
- How accurate is the fit at any given point ($x = 0$)?
- How robust is this fit? Will it vary significantly given new data?



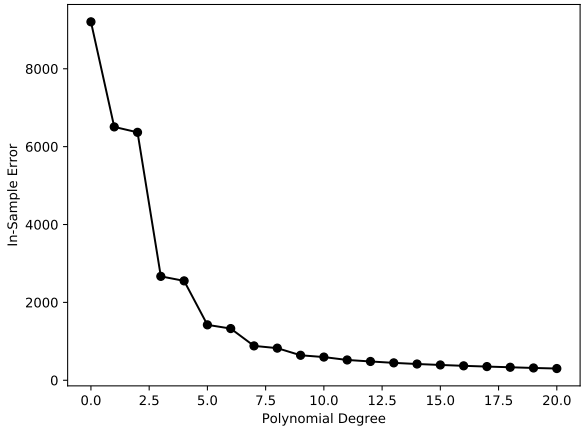
Total error

We can calculate the total error associated with our fit:

$$E = \sum_i \left(y_i - \hat{f}(x_i) \right)^2$$

which is just the sum of the squared residuals of the fit.

If we plot this as a function of polynomial order, we see that the error does indeed go down. So, shouldn't we use the highest-order polynomial possible?



Bias versus variance

Consider the expectation value of the squared error of our model ($\hat{f}$). What should it look like? Recall that $y = f + \epsilon$.

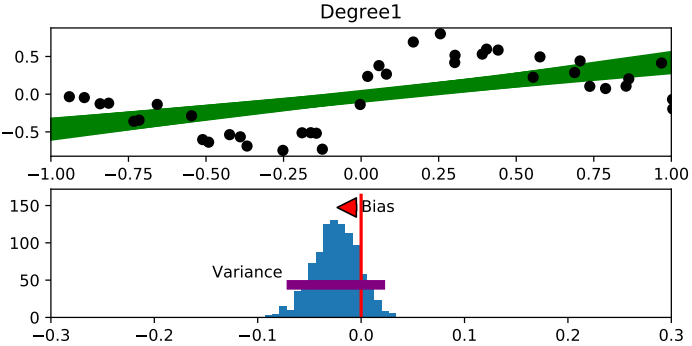
$$\begin{aligned} E\left[\left(y - \hat{f}\right)^2\right] &= \left(E[f] - E[\hat{f}]\right)^2 + E\left[\left(\hat{f} - E\left[\hat{f}\right]\right)^2\right] + \sigma_\epsilon^2 \\ &= \mathbf{Bias}^2 + \mathbf{Var} + \sigma_\epsilon^2 \end{aligned}$$

- Last term: noise intrinsic to the problem. We won't address this here.
- First term: squared bias of the model. Is the expectation value of the model, the expected value of f?
- Second term: variance of the model. How robust is our model to changes in the data?
- The Mean Squared Error has contributions from Bias and Variance.
- There is almost always a tradeoff between bias and variance.

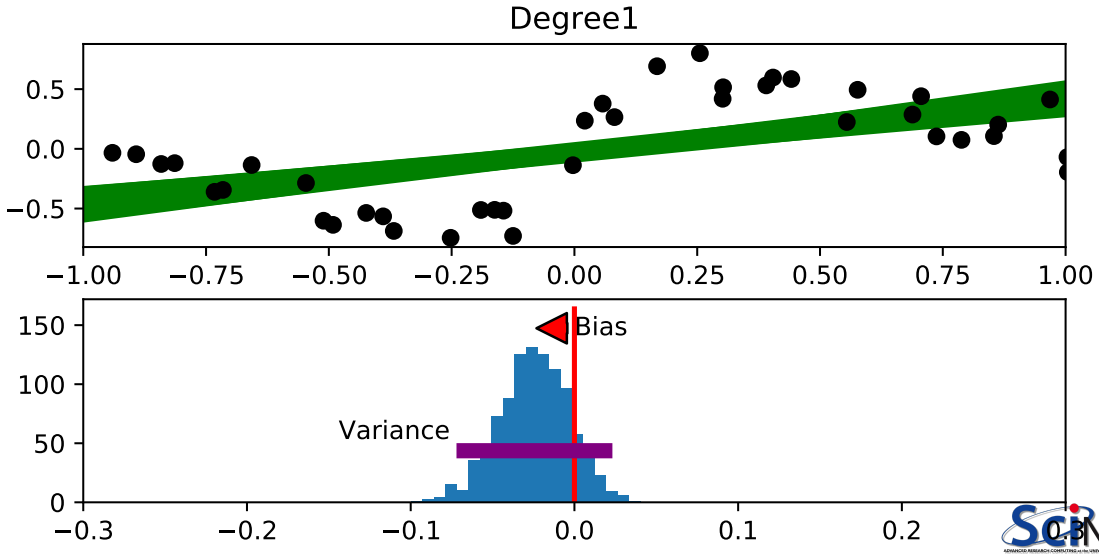
Bias and variance in fitting

Because this is fake data we can directly examine bias and variance in our fits.

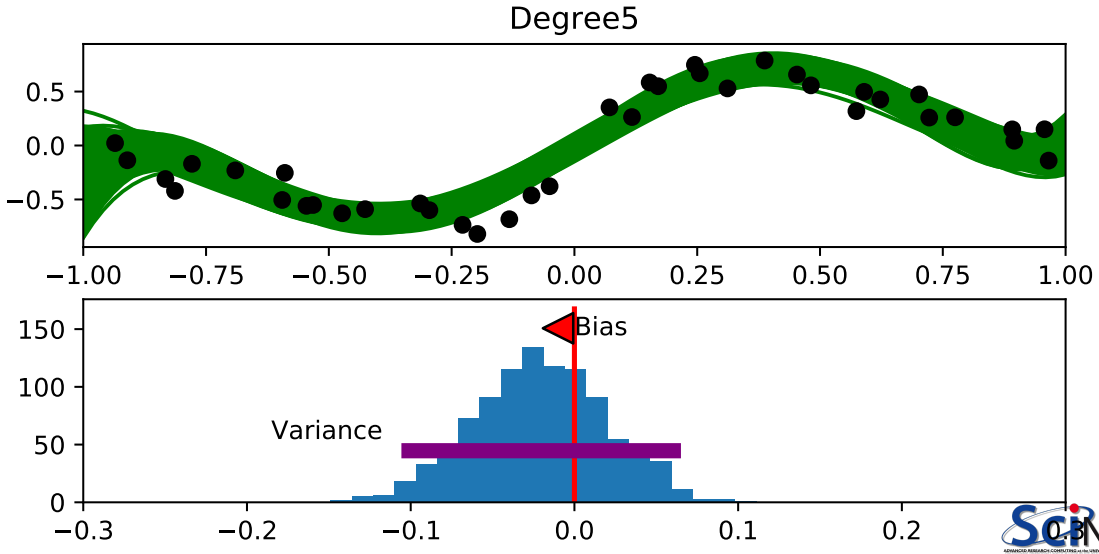
- Generate a bunch of data sets, each with different noise.
- Generate fits to each data set, using a given polynomial degree,
- plot the fits and
- generate a prediction for a given point ($x = 0.0$, say), for each fit.



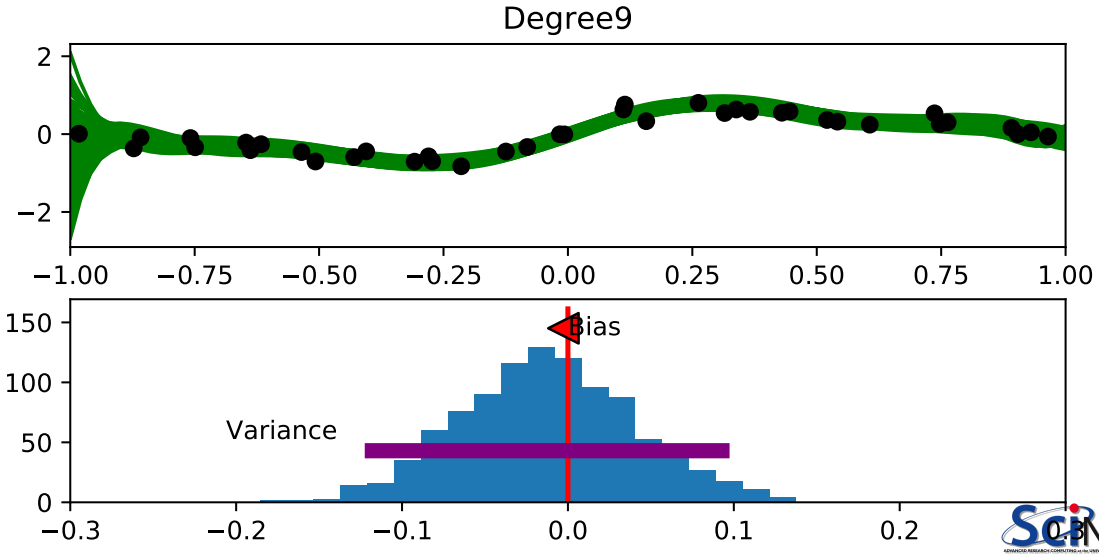
Polynomial fit - linear



Polynomial fit - fifth order



Polynomial fit - ninth order



Bias and variance, continued

Bias is a measure of how consistent the model is with the true behaviour.

- In general, models which are too simple can't capture all of the behaviour of the system.
- As such, estimations based on simple models tend to have larger bias.
- As the model becomes more complex, bias tends to go down, as the model can capture the behaviour of the data.

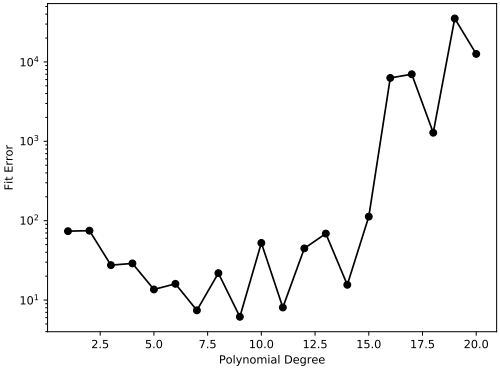
Variance: how sensitive is the model to the particular data set?

- This is related to the question: how general is the model? Is the model learning trends which will generalize to new data? Or is it overfitting to the noise?
- As the model complexity grows, the model tends to have higher variance.

Bias-variance tradeoff

If we compare the total error in the computed model to the true model (not usually available to us!), we can plot the error vs the degree of the polynomial:

- For small degrees, the dominant term is the bias; simpler models can't capture the true behaviour of the system.
- For larger degrees, the dominant term is the variance; more-complex models are generalizing poorly, and overfitting noise.



There's a sweet spot where the two are comparatively low.

That's great, but...

This all very interesting, but what do we do in practice?

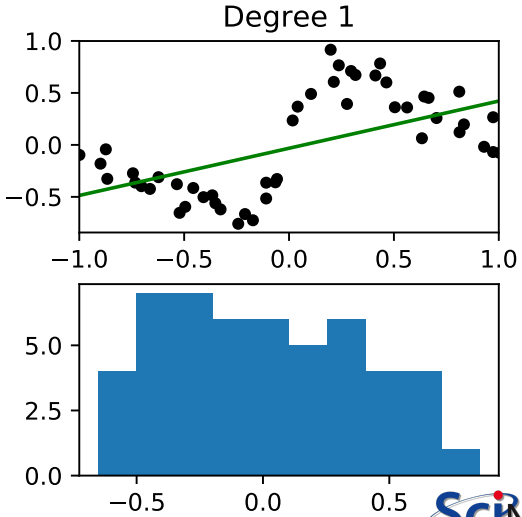
If I'm given a dataset, and I want to fit a polynomial to it, how do I choose the appropriate degree? I obviously don't know the true function in that case.

We'll get to that in a moment. First, let's go over some necessary first steps.

Step 1: plot the residuals via histogram

Always plot a histogram of your residuals. Things to look for:

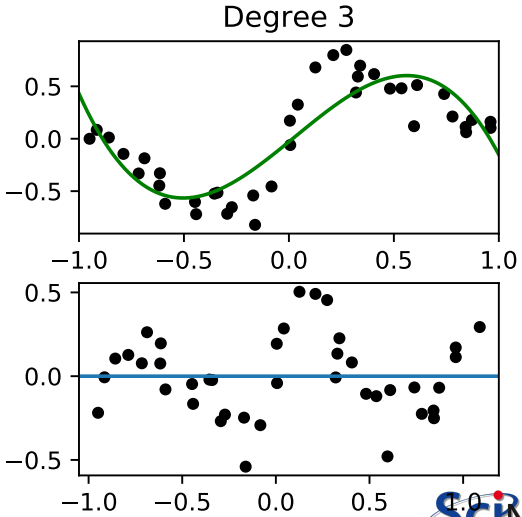
- The mean should be zero. If your residuals are not centered on zero your model is missing important information.
- The distribution should be symmetric. If it's not, it's biased (there 'structure' in the data which has not been captured by the model).
- Distribution should be a Gaussian (an assumption made as part of the fit).



Step 2: plot the residuals, versus index

Always plot your residuals versus index, or independent variable. Things to look for:

- The residuals should scatter randomly above and below zero.
- If the residuals spend too much time above or below zero then there's structure in the data that has not been captured by the model.



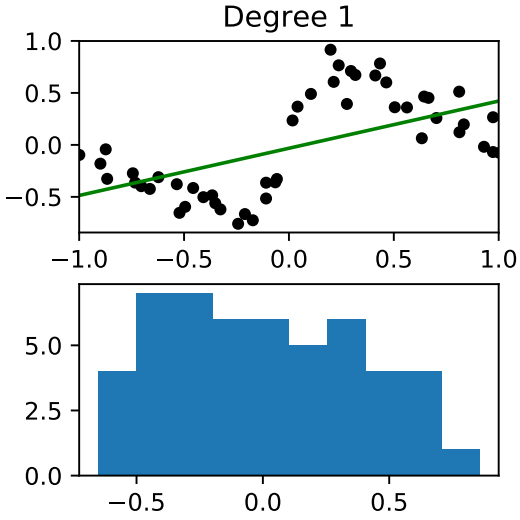
Step 3: check R^2

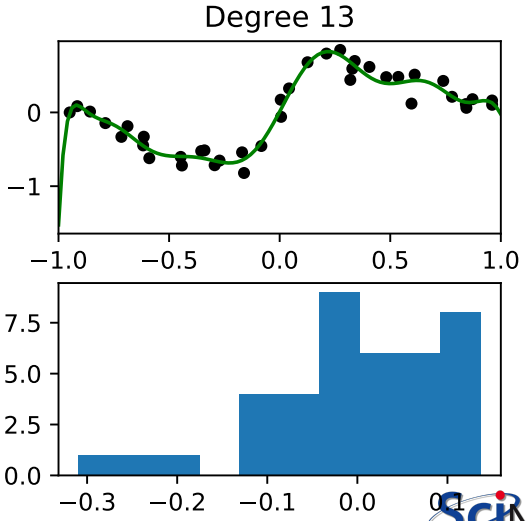
R^2 = (explained variation) / (total variation).

- Explains how much of the variance in the data can be explained by the model.
- All other variation is caused by shortcomings in the model, or noise.
- A high R^2 value is necessary, but not sufficient, for the model to be satisfactory.
- An R^2 can be calculated using the r2_score function from sklearn.metrics subpackage.

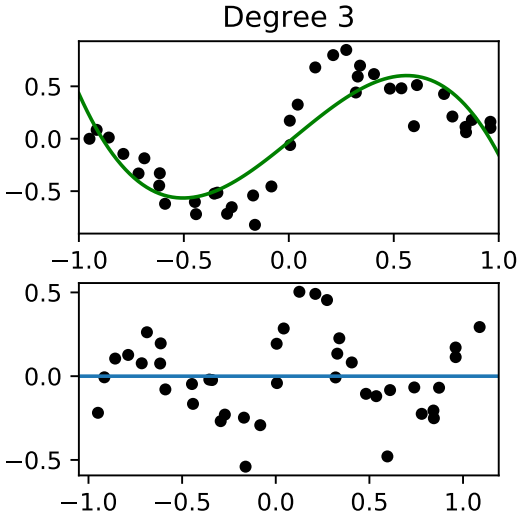
```
In [14]:
In [14]: n = 40
In [15]:
In [15]: x, y = reg.noise_data(n)
In [16]:
In [16]: p = np.polyfit(x, y, 10)
In [17]:
In [17]: fit = np.poly1d(p)
In [18]:
In [18]: import sklearn.metrics as skm
In [19]:
In [19]: skm.r2_score(y, fit(x))
0.93859615700524612
In [20]:
```

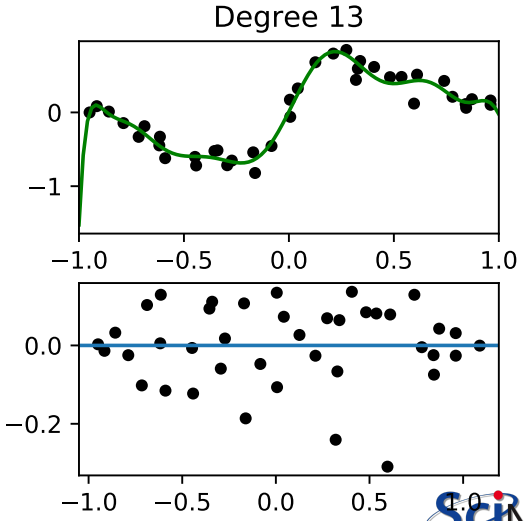
Step 1: revisited





Step 2: revisited





Model training versus validation

Steps 1 - 3 are necessary, but don't tell us which order of polynomial to use.

As we've discussed before, we can crank up the order of the polynomial and get a great fit to the data (even perfect!). But this won't do well on out-of-sample data.

We would can use out-of-sample testing of whatever model we generate to test the quality of the order of the polynomial, to see how it does against new data. But we often don't have any new data.

The solution is to hold out some of the original data. Most of the data is used for training the model, the rest is used for validating it. These data should be chosen randomly.

Model training versus validation, continued

Once the model is chosen, then you can train the selected model on the entire training + validation data set.

But you will probably still want to end your paper with a sentence like "the final model achieved 80% accuracy...". This can't be done on the data the model was trained on (train + validation); in this case, another chunk of data must be held out, for testing.

In the case of training-validation-testing, a common breakdown of the data sizes might be 50%-25%-25% of the initial set. If you don't need a test data set, 2/3-1/3 is common.

Note that the data sets should be chosen randomly!

k-fold Cross Validation

There are some downsides to the approach we've taken for validation hold-out. What if most outliers happen to be in the training set?

Ideally, we should do several partitions and average over the results. This is called k-fold Cross Validation:

- Partition the data set (randomly) into k sets.
- For each set:
  - Train on the remaining $k - 1$ sets.
  - Validate on the held-out set.
- Average the results.

Makes very efficient use of the data set, easily automated.

k-fold cross-validation, continued

How do we choose k?

- if k is too large - the different training sets are very highly correlated (almost all of their points are the same).
- if k is too small - we don't get very much advantage of averaging.

In practice, 10 is a very commonly-used value for k; but again, this depends on the size of your data set.

k-fold cross-validation, regression example

The sklearn package has built-in functionality to make cross-validation easy.

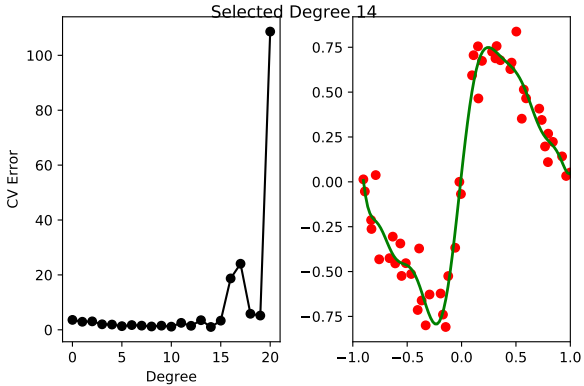
- The model_selection subpackage has a KFold function. It returns the indices of the training and testing data.
- By default KFold does not shuffle the indices, you need to tell it to do so.

```
# crossvalidation.py
import numpy as np, sklearn.model_selection as skms

def estimateError(x, y, d, kfolds = 10):

  err = 0.0
  kfold = skms.KFold(n_splits = kfolds, shuffle = True)

  for train, test in kfold.split(x):

    test_x = x[test];       test_y = y[test]
    train_x = x[train];     train_y = y[train]

    p = numpy.polyfit(train_x, train_y, d)
    fit = numpy.poly1d(p)
    err = err + sum((test_y - fit(test_x))**2)

  return np.sqrt(err)
```

k-fold cross-validation, regression example, continued

We run this function on 50 points using 10-fold cross validation.

We calculated the error for each degree; the minimum is chosen. In practise, the simplest model that is "close enough" to the minimum is generally a good choice.



Cross-validation and bootstrapping

Cross-validation is closely related to a more fundamental method, bootstrapping.

Let's say you want to find some statistic on some statistic of your data.

- What is the standard deviation of the 5th quantile of your data?
- What is the mean and standard deviation of an estimation error for a given model?

You'd like new sets of data that you could calculate your statistics on, and then look at the distribution of those.

Non-parametric Bootstrapping

The key insight to the non-parametric bootstrap is that you already have an unbiased description of the process that generated your data - the data itself.

The approach for the non-parametric bootstrap is:

- Generate synthetic data sets from the original data set by resampling;
- Calculate the statistic of interest on these synthetic data sets, and get the distribution of that particular statistic.

Cross-validation is a particular case: CV takes k (sub)samples of the original data set, applied a function (fit the data set to part, calculate error on the remainder), and calculates the mean.

Bootstrapping can be used far more generally: any time you need to estimate statistics on a quantity whose statistics aren't automatically calculated.

Non-parametric Bootstrapping, example

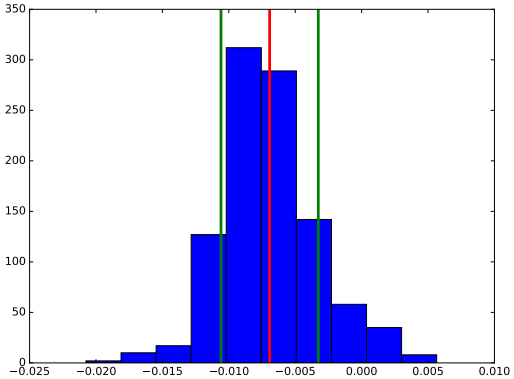
Suppose you want to get statistics on the median of your data. How would you get the uncertainty on the median?

- Randomly sample from your data to create a fake data set.
- By default numpy.random.choice sets "replace = True", so that you are sampling from the full population.
- Do this many times.
- Calculate statistics on the resulting distribution.

```
In [20]: import sklearn.datasets as skd
In [21]: import numpy.random as npr
In [22]: dia = skd.load_diabetes()
In [23]:
In [23]: bmi = dia['data'][:,2]
In [24]:
In [24]: meds = [np.median(npr.choice(bmi, 200))
                 for i in range(1000)]
In [25]:
In [26]: np.mean(meds)
Out[26]: -0.0069275493192715136
In [27]:
In [27]: np.var(meds)
Out[27]: 1.3415090494694259e-05
In [28]:
```

Non-parametric Bootstrapping, example, continued

```
In [28]:
In [28]: plt.hist(meds)
In [29]:
In [29]: mean_meds = np.mean(meds)
In [30]: std_meds = np.sqrt(np.var(meds))
In [31]:
In [31]: plt.axvline(mean_meds, lw = 3,
                     color = 'red')
In [32]: plt.axvline(mean_meds + std_meds,
                     lw = 3, color = 'green')
In [33]: plt.axvline(mean_meds - std_meds,
                     lw = 3, color = 'green')
In [34]:
```



We now have an estimate of the uncertainty on the median.

Notes on Bootstrapping

Bootstrapping strengths:

- Allows you to get information on a statistic when the true distribution of the statistic is unknown.

Bootstrapping weaknesses:

- If the statistic of interest is at the edge of parameter space (minimum, maximum, for example) the bootstrapped distribution does not converge to the true distribution.
- If you have too few data points to begin with, bootstrapping will not magically make things better. Your data must be a true representation of the population from which it is drawn.
- If your data's probability distribution has a long tail, or infinite moments, bootstrapping will fail, or give wildly inaccurate results. Examples include the Cauchy distribution, and non-central Student t distribution with 2 degrees of freedom.

Parametric Bootstrapping

If you know the form of the distribution that describes your data, you can simulate new data sets:

- Fit the distribution to the data;
- Generate synthetic data sets from the now-known distribution to your heart's content;
- Calculate the statistics on these synthetic data sets, and get their distribution.

This works perfectly well if you know a model that will correctly describe your data; and indeed if you do know that, it would be madness *not* to make use of it in your analysis.

Jack-knifing

Another resampling technique is 'jack-knifing'.

- This is a special case of non-parametric bootstrapping.
- Generally used to estimate the bias and variance of a particular statistic.
- In this use-case, the statistic of interest repeatedly recalculated while leaving out one or more different data points. The distribution of the statistic is then analysed.
- Less computationally intensive than bootstrapping, since random numbers are left out.
- Not as common as bootstrapping.

We won't do an example of this, but you need to be aware that it exists.

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.

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.

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.

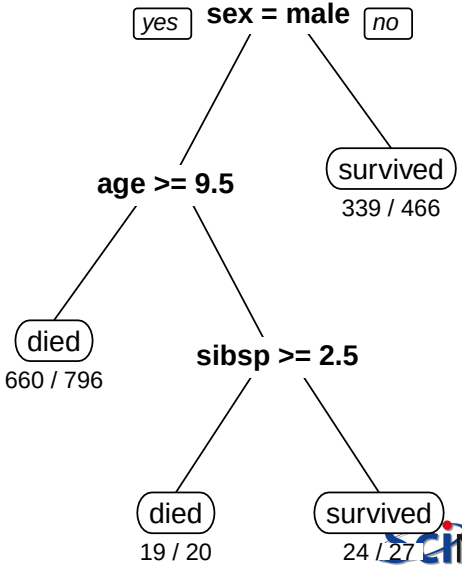
There isn't time to cover all of these. Today we'll cover Decision Trees and kNN.

Decision Trees

A Decision Tree is a structure which classifies an input based on a number of binary decisions.

It splits the data set based on one of the p "features" of the data.

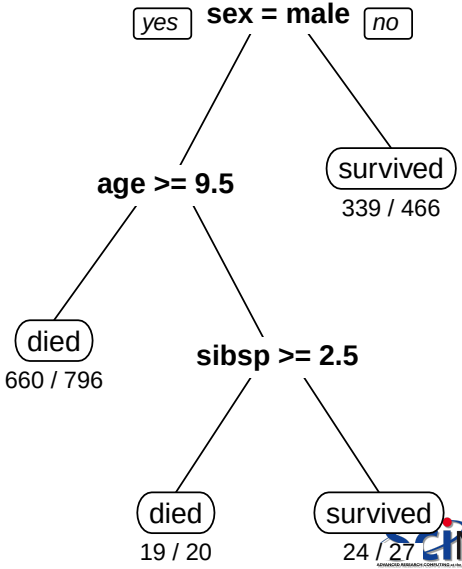
"Features" are the independent variables associated with the data $(x_1, x_2, ..., x_p)$.



Decision Trees, continued

Data can be split based on discrete data ("if category == A") or continuous data ("if height < 1.5m")

The goal of developing a decision tree is to determine when and where and how to split the data, so as to maximize the 'purity' of the resulting sub-data set.



Muppet data set

Consider this data set. The goal is to create a decision tree algorithm which classifies Muppet characters as Sesame Street (SS) or not.

Name	colour	cloths	eye brows	ball nose	SS
Kermit	green	False	False	False	False
Grover	blue	False	False	True	True
Bert	yellow	True	True	True	True
Sam the Eagle	blue	False	True	False	False
Oscar	green	False	True	False	True
Miss Piggy	tan	True	False	False	False

Given this data, how does your algorithm do on this data?

Name	colour	cloths	eye brows	ball nose	SS
Gonzo	blue	True	False	False	??
Big Bird	yellow	False	False	False	??

Splitting algorithms

Consider the following two possible first splits:

- Split based on 'ball nose'.
  - ball nose == True: we get Grover, Bert (SS).
  - ball nose == False: we get Kermit, Sam the Eagle, Miss Piggy (not SS), Oscar (SS).
- Split based on 'cloths'.
  - cloths == True: we get Bert (SS), Miss Piggy (not SS).
  - cloths == False: we get Kermit, Sam the Eagle (not SS), Grover, Oscar (SS).

There's a sense in which the ball nose split is clearly better. It leads to two groups, one of which is totally Sesame Street, and the other which is mostly not.

The other choice gives you two groups which are just as heterogeneous as the original data.

Splitting algorithms, continued

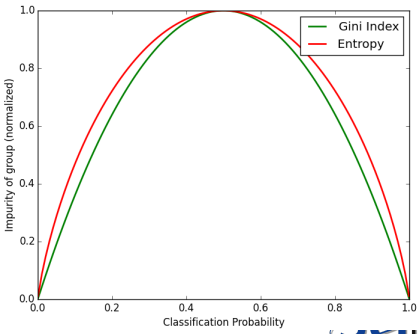
Algorithms which split the data rank possible splits based on increasing 'purity' of the two subgroups it generates.

Consider the probability p that a member of one of the labels is in a given feature category. Two common measures for the 'impurity' of the generated groups are given by

Gini index: $\sum p(1-p)$

Entropy: $-\sum[p \ln p + (1-p) \ln (1-p)]$

Where the sum is over all labels and possible values in the given category. A perfect Gini index is an impurity of 0, or a probability of 0 or 1.



Muppet data set, again

Probabilities	ball nose	no ball nose	cloths	no cloths
SS	2/3	1/3	1/3	2/3
non-SS	0/3	3/3	1/3	2/3

So the Gini indices ($\sum p(1-p)$) for splitting on these features are

$$\frac{2}{3}\left(1-\frac{2}{3}\right)+\frac{1}{3}\left(1-\frac{1}{3}\right)+\frac{0}{3}\left(1-\frac{0}{3}\right)+\frac{3}{3}\left(1-\frac{3}{3}\right)=\frac{4}{9}$$ Better!

$$\frac{1}{3}\left(1-\frac{1}{3}\right)+\frac{2}{3}\left(1-\frac{2}{3}\right)+\frac{1}{3}\left(1-\frac{1}{3}\right)+\frac{2}{3}\left(1-\frac{2}{3}\right)=\frac{8}{9}$$ Worse!

Name	colour	cloths	eye brows	ball nose	SS
Kermit	green	False	False	False	False
Grover	blue	False	False	True	True
Bert	yellow	True	True	True	True
Sam the Eagle	blue	False	True	False	False
Oscar	green	False	True	False	True
Miss Piggy	tan	True	False	False	False

Splitting algorithms, continued more

So how do these algorithms proceed?

- While every data point is not in a pure sub-tree:
  - For each feature which we haven't yet split upon, for the data remaining in the sub-tree, consider a split:
    - ⋆ If the feature is categorical, consider all values, split by value and measure the impurity of the resulting subgroups.
    - ⋆ If the feature is continuous, use line optimization to choose the best point at which to split, keeping track of the impurity at that point.
  - Choose the split which maximizes the change in the impurity (smallest impurity value), and split the data.

Decision tree example

Let's use a Python package to build a decision tree. We'll use the wine data set.

- The data consists of 13 measurements of 178 wines, of 3 classes.
- It's one of the data sets which comes with sklearn.datasets.
- The data comes as an sklearn 'bunch'.
- We first randomly split the data set, 80/20, into training and test data sets.

```
In [34]: import sklearn.datasets as skd
In [35]: import sklearn.model_selection as skms
In [36]:
In [36]: data0 = skd.load_wine()
In [37]: data = data0.data
In [38]: targets = data0.target
In [39]: features = data0.feature_names
In [40]:
In [40]: train_x, test_x, train_y, test_y =
      skms.train_test_split(data, targets, test_size = 0.2)
In [41]:
In [41]: train_x.shape
Out[41]: (142, 13)
In [42]: train_y.shape
Out[42]: (142,)
In [43]:
```

Decision tree example, continued

Now that the data's split up, we're ready to generate the tree.

- Load the sklearn.tree and sklearn.metrics modules.
- Create the tree model.
- Train the model.
- Check the result against the training data.
- Pretty good fit!
- Plot the result.

```
In [43]:
In [43]: import sklearn.tree as skt
In [44]: import sklearn.metrics as skm
In [45]:
In [45]: model = skt.DecisionTreeClassifier()
In [46]:
In [46]: model = model.fit(train_x, train_y)
In [47]: pred = model.predict(train_x)
In [48]:
In [48]: skm.accuracy_score(train_y, pred)
Out[48]: 1.0
In [49]:
```

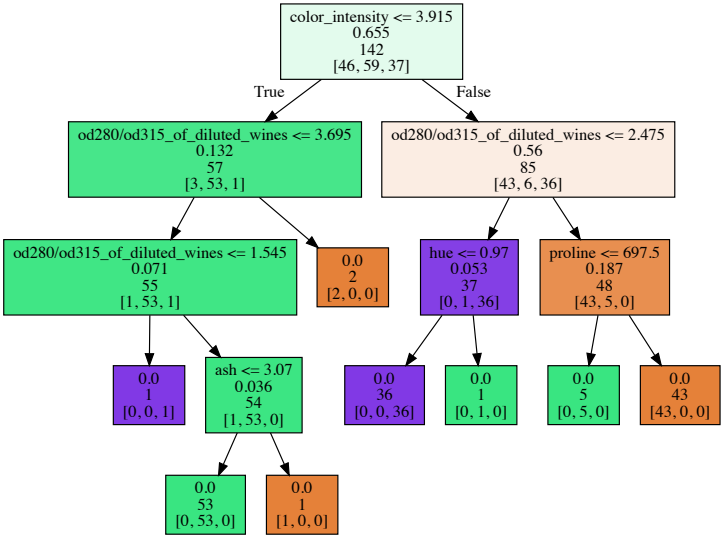
Decision tree example, continued more

It's always good to plot your decision tree.

```
In [49]:
In [49]: import pydotplus
In [50]:
In [50]: dot_data = skt.export_graphviz(model, out_file = None,
    ...:  class_names = [str(i) for i in unique(targets)],
    ...:  feature_names = features, impurity = False, filled = True,
    ...:  label = 'none')
In [51]:
In [51]: g = pydotplus.graph_from_dot_data(dot_data)
In [52]:
In [52]: g.write_pdf("wine_tree.pdf")
In [53]:
```

If this looks clunky, it's because it is.

Our decision tree



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 [53]:
In [53]: skm.confusion_matrix(train_y, pred)
Out[53]:
array([[49,  0,  0],
       [ 0, 54,  0],
       [ 0,  0, 39]])
In [54]:
```

Decision tree example, continued even more

Ok, but how does the decision tree do on the test data?

- Test the built tree against the test data.
- Print out the table of results.
- Not bad!

```
In [54]:
In [54]: test_pred = model.predict(test_x)
In [55]:
In [55]: skm.confusion_matrix(test_y, test_pred)
Out[55]:
array([[10,  0, 0],
       [ 2, 14, 1],
       [ 0,  1, 8]])
In [56]:
In [56]: skm.accuracy_score(test_y, test_pred)
Out[56]: 0.88888888888888884
In [57]:
```

Trees and over-fitting

As with polynomials and regression, we can easily produce overly-complex decision trees which do great on the training data, but don't generalize.

In fact, this is guaranteed to happen with decision trees, since given enough splits, it will always perfectly classify the data.

How do we deal with this? The usual approach is to prune the tree at some level, where the results are "good enough", and the model is not "too complex".

Random forests

You may have heard of "random forests". What are those?

- Random forests fall under the category of "ensemble methods". This means an averaging over several machine-learning models.
- In this case, a random forest is an average over a collection of decision trees.
- To do this,
  - bootstrapping is applied to the data set in question, and decision trees are fit to each sample;
  - however, during the training of the trees, at each split only a subset of possible features are chosen as split candidates;
  - predictions on out-of-sample data are then generated, and an average over all trees is made.
- This results in a lowering of the variance, which is inherently large with decision trees.

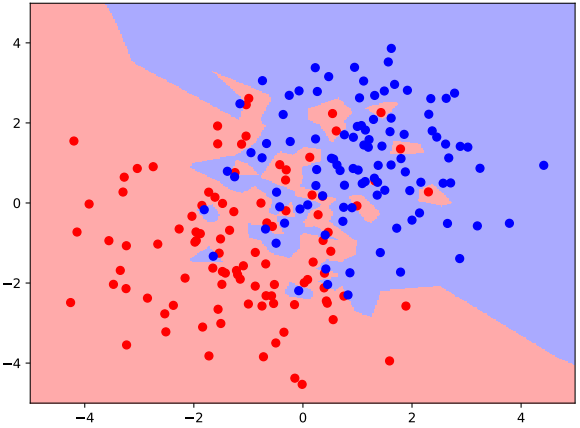
If you end up using decision trees in your research, random forests are worth considering.

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
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)))
```

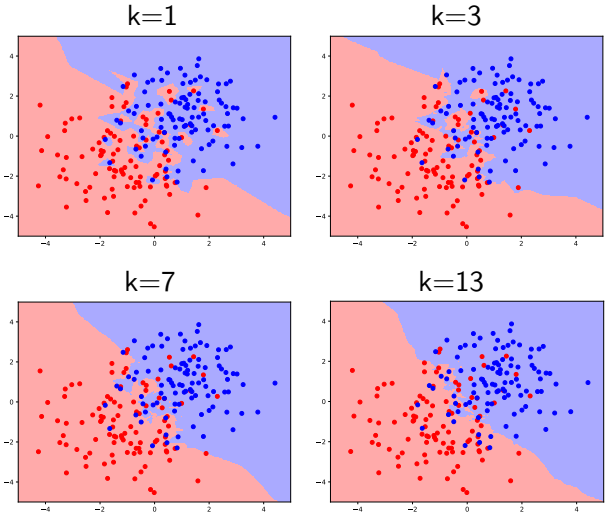
```
# Set up the background grid.
xx, yy = np.meshgrid(np.arange(-5, 5, 0.01),
                     np.arange(-5, 5, 0.01))

# Build the model, and train.
model = skn.KNeighborsClassifier(k)
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 tradeoff in kNN, as can be seen by varying k on the same data.

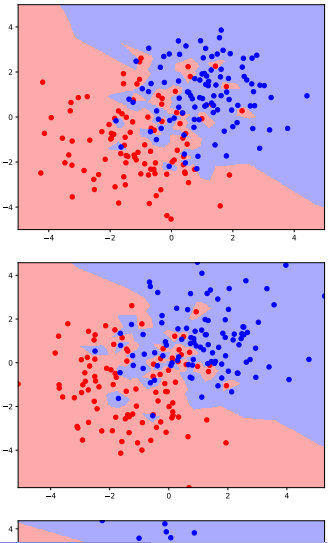
At low k, the variance is very large. The model is trying to fit to every single point.

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 variables to contribute to the solution. To this end, we should scale the variables to that they all get to play. A common technique is to centre the variables 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 Iris data set: 4 measurements of wild irises, of 3 species, 150 samples.

```
In [57]: import sklearn.datasets as skd
In [58]: import sklearn.preprocessing as skp
In [59]:
In [59]: iris = skd.load_iris()
In [60]: x = skp.scale(iris.data)
In [61]: y = iris.target
In [62]:
In [62]: x.shape
Out[62]: (150, 4)
In [63]: iris.feature_names
Out[63]:
['sepal length (cm)', 'sepal width (cm)',
'petal length (cm)', 'petal width (cm)']
In [64]: iris.target_names
Out[64]:
array(['setosa', 'versicolor', 'virginica'])
```

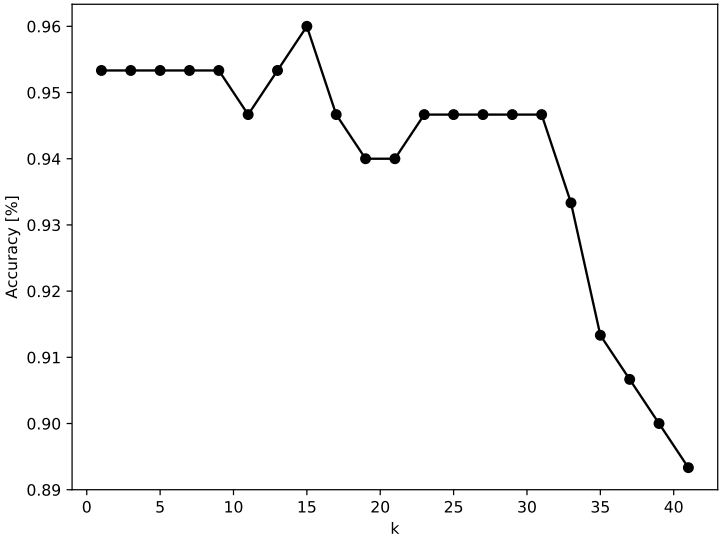
Cross-validation and kNN, continued

How do we use the sklearn cross-validation?

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

```
In [65]:
In [65]: import sklearn.neighbors as skn
In [66]: import sklearn.model_selection as skms
In [67]:
In [67]: kvalues = range(1, 43, 2)
In [68]: scores = np.zeros(len(kvalues))
In [69]:
In [69]: for i, k in enumerate(kvalues):
...:         model = skn.KNeighborsClassifier(k)
...:         scores[i] = np.mean(skms.cross_val_score(model,
                                 x, y, cv = 10))
In [70]:
In [70]: plt.plot(kvalues, scores, 'ko-')
In [71]:
```

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 [71]:
In [71]: bestmodel = skn.KNeighborsClassifier(15)
In [72]:
In [72]: bestmodel = bestmodel.fit(x, y)
In [73]:
In [73]: pred = bestmodel.predict(x)
In [74]:
In [74]: skm.confusion_matrix(y, pred)
array([[50,  0,  0],
       [ 0, 48,  2],
       [ 0,  3, 47]])
In [74]:
In [74]: skm.accuracy_score(y, pred)
Out[74]: 0.96666666666666667
In [75]:
```

Classification summary

Things to remember about decision trees and kNN:

- Decision tree strength: can sensibly deal with categorical data.
- Decision tree strength: perform implicit feature selection.
- Decision tree strength: easy to understand (and explain) the results.
- Decision tree weakness: prone to over-fitting.
- 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 (data must be numeric).

Note that there are other classification algorithms out there: logistic regression, naive Bayes, support vector machines, *etc*.

Clustering

Let's switch to a different sort of classification approach: classification without the labels.

- This is what is known as a type "unsupervised learning".
- It's "unsupervised" because there are no labels."

This can show up in all sorts of applications:

- Finding patterns in properties of galaxies.
- Determine proteins with similar interaction types.
- Market segmentation.
- "Customers who buy X often buy...".

There are two main clustering approaches you'll run into: k-means and hierachical clustering.

Clustering, continued

The reason for using algorithms to find clusters in the data is because

- It's difficult to find clusters in high-dimensional data (since you can't visualize it all at once).
- You might want to summarize a large number of observations in to fewer, similar clusters.

Obviously, we haven't defined what we mean by "similar" or "cluster" yet.

- A "cluster" is a group of data points which are centered around some central, average point.
- The "similarity" between points is determined by some measure of "distance" between them, in the p dimensional space in which they live.
- In continuous spaces the distance can be Euclidean, or some other measure of distance (L1 norm).
- In ordinal spaces (bag-of-words counts, for example) you can use the "cosine similarity"

$$\cos\theta = \frac{\mathbf{A}\cdot\mathbf{B}}{|\mathbf{A}||\mathbf{B}|}$$

K-means clustering

K-means clustering is a geometric clustering algorithm which finds roughly-spherical blobs of clusters amongst the data. The algorithm is straightforward. Starting with k initial cluster centres:

- Assign each data point to the nearest centre.
- Recalculate the center of each cluster, based on its members.
- Move the centres to the new locations.
- Repeat until converged (the centres stop moving).

The value of k must be specified before starting.

K-means clustering, example

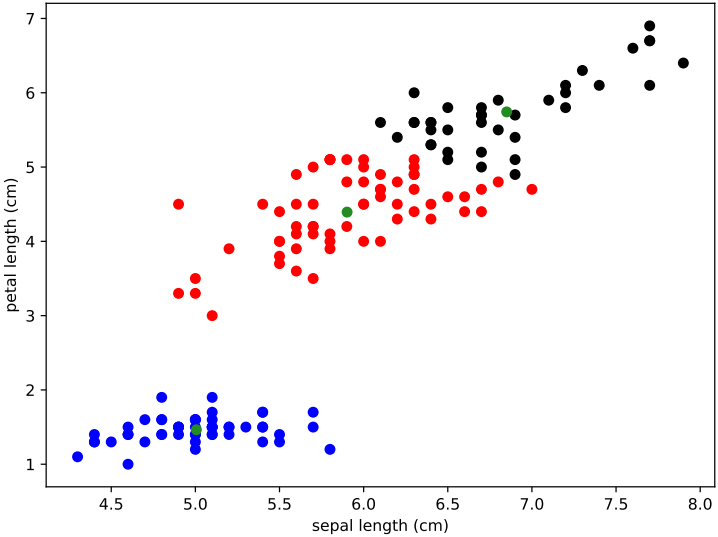
As you might expect, K-means is built into sklearn.

It's as easy to use as you might hope.

Once the model is trained, you can get the centers of the clusters, and the predicted labels, using the "cluster_centers_" and "labels_" model entries.

```
In [75]: import sklearn.cluster as skc
In [76]:
In [76]: model = skc.KMeans(n_clusters = 3)
In [77]: model = model.fit(x)
In [78]:
In [78]: plt.scatter(x[:,0], x[:,2], c = model.labels_)
In [79]:
In [79]: for i in range(3):
...:         plt.scatter(model.cluster_centers_[i][0],
...:                     model.cluster_centers_[i][2],
...:                     c = 'ForestGreen')
In [80]:
```

K-means clusters, example, continued



K-means clustering, continued

K-means has both strengths and weaknesses.

- You need to know what value of k to use.
- Random initialization of the centres can go badly wrong.
- For this to be robust, you need to repeat many times.
- This is usually done automatically by sklearn's KMeans, and the best result is returned.
- k-means has a tendency to make equally-populated clusters, which can lead to incorrect results.

For this to work consistently, we need a way to measure the quality of the model.

K-means clustering, quality measures

A few measures of error have been developed for K-means.

- We'd like to minimize the within-cluster sum of squares.

$$\mathbf{WCSS} = \sum_{i}^{k} \sum_{j \in S_i} |x_j - \mu_j|^2$$

- We'd like to maximize the between-cluster sum of squares.

$$\mathbf{ICSS} = \sum_{i}^{n} \sum_{j}^{n} \delta(S_i, S_j)\, |x_i - x_j|^2$$

These are output by standard k-means algorithms.

K-means and cross-validation

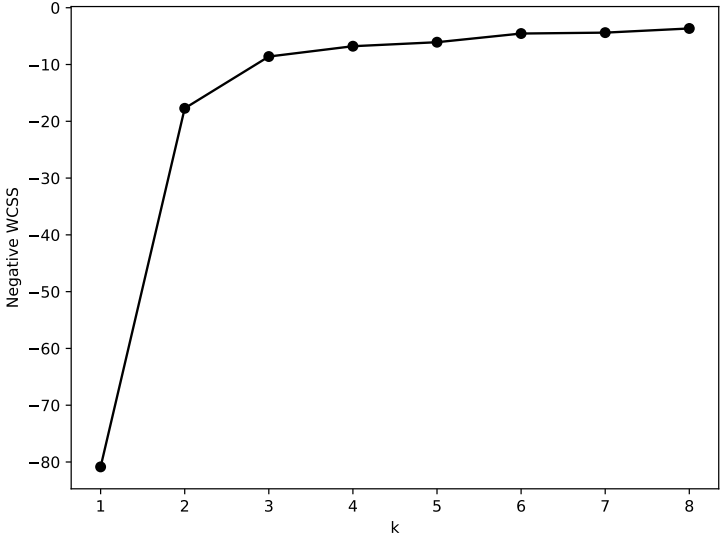
How do we pick k? You guessed it!

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

```
In [80]:
In [80]: kvalues = range(1, 9)
In [81]: scores = np.zeros(len(kvalues))
In [82]:
In [82]: for k in kvalues:
...:         model = skc.KMeans(n_clusters = k)
...:         scores[k - 1] = np.mean(skms.cross_val_score(model,
                             x, cv = 10))
In [83]:
In [83]: plt.plot(kvalues, scores, 'ko-')
In [84]:
```

Unlike other algorithms, the accuracy of k-means does not 'turn over', meaning start to get worse with increasing k.

K-means clusters, example, continued



Hierarchical clustering

K-means uses a geometric approach to clustering. Hierarchical clustering works point-by-point.

Agglomerative (bottom up) clustering:

- All data points start in their own cluster.
- At each iteration, the two "best matching" are joined into the same cluster.
- Repeat until there is only cluster left.

This builds a tree of connections. This tree then needs to be pruned to distinguish the clusters. To do this we still need some sort of distance metric, and a linkage criteria, which specifies the dissimilarity of the clusters.

- k-means-like: what is the distance between the centres of the clusters which have been built thus far?
- single linkage: what is the minimum distance between any two points in two clusters.
- mean linkage: what is the mean distance between all points in two clusters?

Agglomerative clustering, example

As you might expect, agglomeration clustering is built into sklearn.

Let's use a different data set to test this: swiss roll.

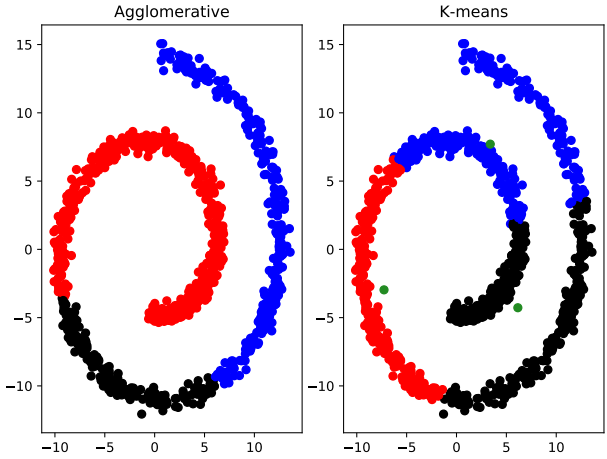
The connectivity of the points is determined using the kneighbors_graph command. This creates a graph between the points, based on some metric.

```
In [84]: import sklearn.datasets as skd
In [85]: x, y = skd.make_swiss_roll(1000, noise = 0.4)
In [86]: x.shape
Out[86]: (1000, 3)
In [87]:
In [87]: x = np.c_[x[:, 0], x[:, 2]]
In [88]:
In [88]: x.shape
Out[88]: (1000, 2)
In [89]:
In [89]: model = skc.AgglomerativeClustering(n_clusters = 3,
             connectivity = skn.kneighbors_graph(x, 30))
In [90]: model = model.fit(x)
In [91]:
In [91]: plt.scatter(x[:,0], x[:,1], c = model.labels_)
In [92]:
```

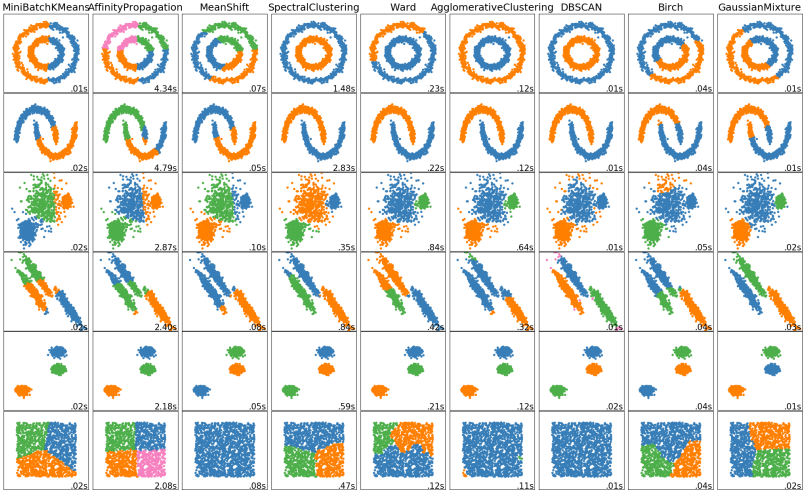
K-Means versus Hierarchical clustering

K-means and hierarchical clustering have very different behaviours.

- K-means only cares about distances "as the crow flies".
- Hierarchical cares about distances between individual data points.
- K-means requires the number of clusters up front.
- Hierarchical gives you an entire tree.



Scikit-learn clustering algorithms



http://scikit-learn.org/stable/auto_examples/cluster/plot_cluster_comparison.html

Other machine-learning topics

Unfortunately, we can't even come close to covering everything that we could. Topics which you may want to explore:

- Other classification algorithms: logistic regression, naive Bayes, neural networks, support vector machines.
- Ensembling methods (like random forests).
- Dimensionality reduction.
- Nonparametric regression: kernels, LOESS, LOWESS.
- Variable selection: forward and backward selection, AIC, BIC, Lasso/Ridge regression.

And others. Be aware that your field may use methods not listed here.