Ensembles

In this lecture, we look at combining models into what is called an ensemble. It turns out that the ensemble often out-performs the individual models that it contains. There are lots of analogies here (wisdom of crowds; committees; and so on). You can try using an ensemble whenever you like, but perhaps their main use is when there is high variance. Recall that a learner with high variance is quite sensitive to the make-up of its training set: small changes in the examples on which it is trained can lead to large differences in the hypothesis that it learns. This can lead to overfitting.

The main ways of building ensembles are:

- bagging
- boosting
- stacking

They are general techniques, applicable to both regression and classification.

Bagging

In bagging (which is a contraction of 'bootstrap aggregating'), several estimators of the same type make predictions and the predictions are aggregated. We need to understand two aspects. First, where do the members of the ensemble come from? Second, how does the ensemble make a prediction?
Suppose we want an ensemble that contains $num$ estimators. In bagging, we take the training set and create $num$ different versions of it by sampling from it with replacement. So now we have $num$ training sets, so we can learn $num$ models:

```
do num times
  - sample |Train| examples with replacement from Train
  - build a model from this sample
```

Now that we have $num$ different models, how do we make predictions?

- For classification, each classifier in the ensemble predicts the class, and these are taken as votes: the class with the most votes is the one returned as the prediction of the ensemble. (If the members of the ensemble are capable of outputting probabilities, then you can instead average the probabilities.)
- For regression, each regressor in the ensemble predicts the numeric value of the dependent variable, and these are averaged, and the average is returned as the prediction of the ensemble.

In bagging, the aggregation (voting or averaging) is not usually weighted, i.e. all the estimators in the ensemble are treated equally.

How many models should you use in the ensemble? To some extent, 'the more, the merrier': predictions typically become more reliable as the ensemble gets larger. Seldom is the accuracy of the ensemble lower than the accuracy of any of its members — but this is not guaranteed. In practice, you can either guess how many models to use, or do some model selection.

Results are often improved by increasing the diversity of the estimators in the ensemble. (There are more analogies here: a committee might work better if there is a mixture of expertise.) One way to do this is to deliberately use estimators with high variance.

Questions:

- Normally, we want to avoid learners with high variance. Why might we do the opposite in an ensemble?
- Suppose the members of the ensemble use the $k$-nearest-neighbours technique. What might you do to keep variance high?

In fact, bagging with $k$-nearest-neighbours may not be successful at all because the different versions of the training set may not result in sufficiently different members of the ensemble.

To increase diversity, we might combine bagging with some degree of randomization. There are at least two ways of doing this:

- Modify the learner so that it is partly random (obviously at the expense of some accuracy). For example, perhaps the learner normally chooses the best of a set of options but you modify it so that it chooses randomly but with probabilities weighted towards the best options.
- Another way to randomize is to train on different, randomly-chosen subsets of the features. This has the advantage of not requiring any changes to the learning algorithm. If you want to try an ensemble of $k$-nearest-neighbour estimators, then this is probably the best way of getting some diversity into the ensemble: the different members of the ensemble will find different neighbours because they will be computing distance using different subsets of the features.

**Random Forests in scikit-learn**
scikit-learn does not offer any generic classes for bagging, with or without randomization. But it does offer **random forests**, both for classification and regression. These are ensembles of **decision trees** (which we have not covered), with randomness built into the learning algorithm, as per the first bullet point above. Random forests often do very well in Kaggle competitions. Let's give them a go:

```
In [3]:
# Use pandas to read the CSV file
df = pd.read_csv("dataset-corkB.csv")

# Remove anomalies from flarea
df = df[(df['flarea'].isnull()) | ((df['flarea'] > 10) & (df['flarea'] < 1000))]
df.reset_index(drop=True, inplace=True)
# For simplicity, rather than impute missing values, we'll simply remove examples with missing values
df.dropna(subset=['flarea', 'bdrms', 'floors', 'devment', 'price'], inplace=True)
df.reset_index(drop=True, inplace=True)
# Encode nominal-valued features
# - devment is easy because it is binary-valued
df.replace({'devment': {'SecondHand': 0, 'New': 1}}, inplace=True)
# - for type we use one-hot encoding
one_hot = pd.get_dummies(df['type'], 'type', '_')
df.drop('type', axis=1, inplace=True)
df = pd.concat([df, one_hot], axis=1)
# - similarly for ber
one_hot = pd.get_dummies(df['ber'], 'ber', '_')
df.drop('ber', axis=1, inplace=True)
df = pd.concat([df, one_hot], axis=1)
# - and similarly for location
one_hot = pd.get_dummies(df['location'], 'location', '_')
df.drop('location', axis=1, inplace=True)
df = pd.concat([df, one_hot], axis=1)
# Scaling - for simplicity, we'll do min-max scaling on just flarea, bdrms and bthrms
df['flarea'] -= 40
df['flarea'] /= 460
df['bdrms'] /= 10
df['bthrms'] /= 10

```
'location_StLukes', 'location_SundaysWell', 'location_TheLough', 'location_Togher', 'location_TurnersCross',
'location_VictoriaCross', 'location_Waterfall', 'location_WesternRoad', 'location_Wilton']].values
y = df['price'].values

# Train and test
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.cross_validation import cross_val_score

estimators = {'Linear': LinearRegression(),
              'Tree': DecisionTreeRegressor(),
              'Forest': RandomForestRegressor(n_estimators = 10)}
for estname, est in estimators.items():
    mses_test = np.abs(cross_val_score(est, X, y, scoring = 'mean_squared_error', cv = 10))
    mean_mse_test = np.mean(mses_test)
    print estname, mean_mse_test

Tree 52440.4152381
Linear 1.97927305223e+29
Forest 54012.6105452

Boosting
Boosting is like bagging in the following respects:

- It uses an ensemble of estimators of the same type (e.g. an ensemble of decision trees, or an ensemble of $k$-nearest-neighbour estimators).
- Individual predictions are aggregated by voting or averaging.

But boosting differs from bagging in the following ways:

- In bagging, the models are learned separately from one another, whereas in boosting each new model is influenced by the accuracy of the previous one. In this way, boosting tries to learn diverse models — in particular, ones that are good at handling the examples that are handled incorrectly by the models already in the ensemble.
- In bagging, the aggregation typically treats all members of the ensemble equally, whereas in boosting, aggregation is weighted.

The best known boosting method is called AdaBoost. In high-level terms (without formulae and omitting some details), it works as follows:

assign equal weight to all examples in $Train$

**do** num **times**

- build a model from the weighted training set
- compute the training error, i.e. the error of this model on the weighted training set
- reweight the examples in $Train$ so that examples the model got wrong get higher weight

You can see that subsequent models focus more on examples that earlier models found 'difficult'. The easiest way to reweight is by resampling: sample $Train$ with replacement but with probabilities proportional to weight. Hence, more highly-weighted examples will be duplicated more frequently in the sample.

Now that we have num different models (and a record of their training error), how do we make predictions? We use weighted voting or weighted averaging, where the weight is inversely proportional to the error.

### AdaBoost in scikit-learn

scikit-learn has implementations of AdaBoost for both classification and regression. Compare it to earlier:

```
In [4]: from sklearn.ensemble import AdaBoostRegressor
est = AdaBoostRegressor(n_estimators = 10)
mse_test = np.abs(cross_val_score(est, X, y, scoring = 'mean_squared_error', cv = 10))
mean_mse_test = np.mean(mse_test)
print 'AdaBoost', mean_mse_test

AdaBoost 87971.3069619
```
Boosting can often convert a ‘weak’ learner into a ‘strong’ ensemble. It is often more accurate than bagging. But sometimes it overfits, resulting in an estimator that is less accurate than a single estimator built from the same data.

**Stacking**

In contrast to bagging and boosting, **stacking** combines different types of model, e.g. linear regression with \( k \)-nearest-neighbours regression. We train them on the same training set. To make predictions, we run each estimator and aggregate: voting or averaging as appropriate.

But this ignores the fact that some members of the ensemble may perform better than others, and these should be given greater weight in the aggregation. In stacking, we use a **meta-learner** to learn the weights. This is still supervised learning so we need a training set so that it knows the correct answers.

In high-level terms, this is how stacking works:

1. split \( \text{Train} \) into two: \( \text{Train}_1 \) and \( \text{Train}_2 \)
2. do \( \text{num} \) times:
   - build a model from \( \text{Train}_1 \) (a different model each time)
3. create a training set for the meta-learner as follows:
   - for each \( \langle x, y \rangle \in \text{Train}_2 \):
     - there are \( \text{num} \) features, which are the predictions for \( x \) made by the \( \text{num} \) models
     - the dependent variable is \( y \) (from \( \text{Train}_2 \))
   - build a model for this new training set

You can make this more robust by using \( k \)-fold cross-validation instead of a single split into \( \text{Train}_1 \) and \( \text{Train}_2 \).

The meta-learner is often something simple, e.g. linear or logistic regression.

Stacking is less well-understood than bagging and boosting. While it can work well, it often doesn't!

I'm not aware of a scikit-learn implementation, so we can't try it without writing it for ourselves.