# # Ensembling

#

# \*Adapted from Chapter 8 of [An Introduction to Statistical Learning](http://www-bcf.usc.edu/~gareth/ISL/)\*

# Why are we learning about ensembling?

#

# - Very popular method for improving the predictive performance of machine learning models

# - Provides a foundation for understanding more sophisticated models

# ## Lesson objectives

#

# Students will be able to:

#

# - Define ensembling and its requirements

# - Identify the two basic methods of ensembling

# - Decide whether manual ensembling is a useful approach for a given problem

# - Explain bagging and how it can be applied to decision trees

# - Explain how out-of-bag error and feature importances are calculated from bagged trees

# - Explain the difference between bagged trees and Random Forests

# - Build and tune a Random Forest model in scikit-learn

# - Decide whether a decision tree or a Random Forest is a better model for a given problem

# # Part 1: Introduction

#

# Let's pretend that instead of building a single model to solve a binary classification problem, you created \*\*five independent models\*\*, and each model was correct about 70% of the time. If you combined these models into an "ensemble" and used their majority vote as a prediction, how often would the ensemble be correct?

import numpy as np

# set a seed for reproducibility

np.random.seed(1234)

# generate 1000 random numbers (between 0 and 1) for each model, representing 1000 observations

mod1 = np.random.rand(1000)

mod2 = np.random.rand(1000)

mod3 = np.random.rand(1000)

mod4 = np.random.rand(1000)

mod5 = np.random.rand(1000)

# each model independently predicts 1 (the "correct response") if random number was at least 0.3

preds1 = np.where(mod1 > 0.3, 1, 0)

preds2 = np.where(mod2 > 0.3, 1, 0)

preds3 = np.where(mod3 > 0.3, 1, 0)

preds4 = np.where(mod4 > 0.3, 1, 0)

preds5 = np.where(mod5 > 0.3, 1, 0)

# print the first 20 predictions from each model

print preds1[:20]

print preds2[:20]

print preds3[:20]

print preds4[:20]

print preds5[:20]

# average the predictions and then round to 0 or 1

ensemble\_preds = np.round((preds1 + preds2 + preds3 + preds4 + preds5)/5.0).astype(int)

# print the ensemble's first 20 predictions

print ensemble\_preds[:20]

# how accurate was each individual model?

print preds1.mean()

print preds2.mean()

print preds3.mean()

print preds4.mean()

print preds5.mean()

# how accurate was the ensemble?

print ensemble\_preds.mean()

# \*\*Note:\*\* As you add more models to the voting process, the probability of error decreases, which is known as [Condorcet's Jury Theorem](http://en.wikipedia.org/wiki/Condorcet%27s\_jury\_theorem).

# ## What is ensembling?

#

# \*\*Ensemble learning (or "ensembling")\*\* is the process of combining several predictive models in order to produce a combined model that is more accurate than any individual model.

#

# - \*\*Regression:\*\* take the average of the predictions

# - \*\*Classification:\*\* take a vote and use the most common prediction, or take the average of the predicted probabilities

#

# For ensembling to work well, the models must have the following characteristics:

#

# - \*\*Accurate:\*\* they outperform the null model

# - \*\*Independent:\*\* their predictions are generated using different processes

#

# \*\*The big idea:\*\* If you have a collection of individually imperfect (and independent) models, the "one-off" mistakes made by each model are probably not going to be made by the rest of the models, and thus the mistakes will be discarded when averaging the models.

#

# There are two basic \*\*methods for ensembling:\*\*

#

# - Manually ensemble your individual models

# - Use a model that ensembles for you

# # Part 2: Manual ensembling

#

# What makes a good manual ensemble?

#

# - Different types of \*\*models\*\*

# - Different combinations of \*\*features\*\*

# - Different \*\*tuning parameters\*\*

# ![Machine learning flowchart](images/crowdflower\_ensembling.jpg)

#

# \*Machine learning flowchart created by the [winner](https://github.com/ChenglongChen/Kaggle\_CrowdFlower) of Kaggle's [CrowdFlower competition](https://www.kaggle.com/c/crowdflower-search-relevance)\*

# ## Comparing manual ensembling with a single model approach

#

# \*\*Advantages of manual ensembling:\*\*

#

# - Increases predictive accuracy

# - Easy to get started

#

# \*\*Disadvantages of manual ensembling:\*\*

#

# - Decreases interpretability

# - Takes longer to train

# - Takes longer to predict

# - More complex to automate and maintain

# - Small gains in accuracy may not be worth the added complexity

# # Part 3: Bagging

#

# The primary weakness of \*\*decision trees\*\* is that they don't tend to have the best predictive accuracy. This is partially due to \*\*high variance\*\*, meaning that different splits in the training data can lead to very different trees.

#

# \*\*Bagging\*\* is a general purpose procedure for reducing the variance of a machine learning method, but is particularly useful for decision trees. Bagging is short for \*\*bootstrap aggregation\*\*, meaning the aggregation of bootstrap samples.

#

# What is a \*\*bootstrap sample\*\*? A random sample with replacement:

# set a seed for reproducibility

np.random.seed(1)

# create an array of 1 through 20

nums = np.arange(1, 21)

print nums

# sample that array 20 times with replacement

print np.random.choice(a=nums, size=20, replace=True)

# \*\*How does bagging work (for decision trees)?\*\*

#

# 1. Grow B trees using B bootstrap samples from the training data.

# 2. Train each tree on its bootstrap sample and make predictions.

# 3. Combine the predictions:

# - Average the predictions for \*\*regression trees\*\*

# - Take a vote for \*\*classification trees\*\*

#

# Notes:

#

# - \*\*Each bootstrap sample\*\* should be the same size as the original training set.

# - \*\*B\*\* should be a large enough value that the error seems to have "stabilized".

# - The trees are \*\*grown deep\*\* so that they have low bias/high variance.

#

# Bagging increases predictive accuracy by \*\*reducing the variance\*\*, similar to how cross-validation reduces the variance associated with train/test split (for estimating out-of-sample error) by splitting many times an averaging the results.

# ## Manually implementing bagged decision trees (with B=10)

# read in and prepare the vehicle training data

import pandas as pd

url = 'https://raw.githubusercontent.com/justmarkham/DAT8/master/data/vehicles\_train.csv'

train = pd.read\_csv(url)

train['vtype'] = train.vtype.map({'car':0, 'truck':1})

train

# set a seed for reproducibility

np.random.seed(123)

# create ten bootstrap samples (will be used to select rows from the DataFrame)

samples = [np.random.choice(a=14, size=14, replace=True) for \_ in range(1, 11)]

samples

# show the rows for the first decision tree

train.iloc[samples[0], :]

# read in and prepare the vehicle testing data

url = 'https://raw.githubusercontent.com/justmarkham/DAT8/master/data/vehicles\_test.csv'

test = pd.read\_csv(url)

test['vtype'] = test.vtype.map({'car':0, 'truck':1})

test

from sklearn.tree import DecisionTreeRegressor

# grow each tree deep

treereg = DecisionTreeRegressor(max\_depth=None, random\_state=123)

# list for storing predicted price from each tree

predictions = []

# define testing data

X\_test = test.iloc[:, 1:]

y\_test = test.iloc[:, 0]

# grow one tree for each bootstrap sample and make predictions on testing data

for sample in samples:

X\_train = train.iloc[sample, 1:]

y\_train = train.iloc[sample, 0]

treereg.fit(X\_train, y\_train)

y\_pred = treereg.predict(X\_test)

predictions.append(y\_pred)

# convert predictions from list to NumPy array

predictions = np.array(predictions)

predictions

# average predictions

np.mean(predictions, axis=0)

# calculate RMSE

from sklearn import metrics

y\_pred = np.mean(predictions, axis=0)

np.sqrt(metrics.mean\_squared\_error(y\_test, y\_pred))

# ## Bagged decision trees in scikit-learn (with B=500)

# define the training and testing sets

X\_train = train.iloc[:, 1:]

y\_train = train.iloc[:, 0]

X\_test = test.iloc[:, 1:]

y\_test = test.iloc[:, 0]

# instruct BaggingRegressor to use DecisionTreeRegressor as the "base estimator"

from sklearn.ensemble import BaggingRegressor

bagreg = BaggingRegressor(DecisionTreeRegressor(), n\_estimators=500, bootstrap=True, oob\_score=True, random\_state=1)

# fit and predict

bagreg.fit(X\_train, y\_train)

y\_pred = bagreg.predict(X\_test)

y\_pred

# calculate RMSE

np.sqrt(metrics.mean\_squared\_error(y\_test, y\_pred))

# ## Estimating out-of-sample error

#

# For bagged models, out-of-sample error can be estimated without using \*\*train/test split\*\* or \*\*cross-validation\*\*!

#

# On average, each bagged tree uses about \*\*two-thirds\*\* of the observations. For each tree, the \*\*remaining observations\*\* are called "out-of-bag" observations.

# show the first bootstrap sample

samples[0]

# show the "in-bag" observations for each sample

for sample in samples:

print set(sample)

# show the "out-of-bag" observations for each sample

for sample in samples:

print sorted(set(range(14)) - set(sample))

# How to calculate \*\*"out-of-bag error":\*\*

#

# 1. For every observation in the training data, predict its response value using \*\*only\*\* the trees in which that observation was out-of-bag. Average those predictions (for regression) or take a vote (for classification).

# 2. Compare all predictions to the actual response values in order to compute the out-of-bag error.

#

# When B is sufficiently large, the \*\*out-of-bag error\*\* is an accurate estimate of \*\*out-of-sample error\*\*.

# compute the out-of-bag R-squared score (not MSE, unfortunately!) for B=500

bagreg.oob\_score\_

# ## Estimating feature importance

#

# Bagging increases \*\*predictive accuracy\*\*, but decreases \*\*model interpretability\*\* because it's no longer possible to visualize the tree to understand the importance of each feature.

#

# However, we can still obtain an overall summary of \*\*feature importance\*\* from bagged models:

#

# - \*\*Bagged regression trees:\*\* calculate the total amount that \*\*MSE\*\* is decreased due to splits over a given feature, averaged over all trees

# - \*\*Bagged classification trees:\*\* calculate the total amount that \*\*Gini index\*\* is decreased due to splits over a given feature, averaged over all trees

# # Part 4: Random Forests

#

# Random Forests is a \*\*slight variation of bagged trees\*\* that has even better performance:

#

# - Exactly like bagging, we create an ensemble of decision trees using bootstrapped samples of the training set.

# - However, when building each tree, each time a split is considered, a \*\*random sample of m features\*\* is chosen as split candidates from the \*\*full set of p features\*\*. The split is only allowed to use \*\*one of those m features\*\*.

# - A new random sample of features is chosen for \*\*every single tree at every single split\*\*.

# - For \*\*classification\*\*, m is typically chosen to be the square root of p.

# - For \*\*regression\*\*, m is typically chosen to be somewhere between p/3 and p.

#

# What's the point?

#

# - Suppose there is \*\*one very strong feature\*\* in the data set. When using bagged trees, most of the trees will use that feature as the top split, resulting in an ensemble of similar trees that are \*\*highly correlated\*\*.

# - Averaging highly correlated quantities does not significantly reduce variance (which is the entire goal of bagging).

# - By randomly leaving out candidate features from each split, \*\*Random Forests "decorrelates" the trees\*\*, such that the averaging process can reduce the variance of the resulting model.

# # Part 5: Building and tuning decision trees and Random Forests

#

# - Major League Baseball player data from 1986-87: [data](https://github.com/justmarkham/DAT8/blob/master/data/hitters.csv), [data dictionary](https://cran.r-project.org/web/packages/ISLR/ISLR.pdf) (page 7)

# - Each observation represents a player

# - \*\*Goal:\*\* Predict player salary

# ## Preparing the data

# read in the data

url = 'https://raw.githubusercontent.com/justmarkham/DAT8/master/data/hitters.csv'

hitters = pd.read\_csv(url)

# remove rows with missing values

hitters.dropna(inplace=True)

hitters.head()

# encode categorical variables as integers

hitters['League'] = pd.factorize(hitters.League)[0]

hitters['Division'] = pd.factorize(hitters.Division)[0]

hitters['NewLeague'] = pd.factorize(hitters.NewLeague)[0]

hitters.head()

# allow plots to appear in the notebook

import matplotlib.pyplot as plt

# scatter plot of Years versus Hits colored by Salary

hitters.plot(kind='scatter', x='Years', y='Hits', c='Salary', colormap='jet', xlim=(0, 25), ylim=(0, 250))

# define features: exclude career statistics (which start with "C") and the response (Salary)

feature\_cols = hitters.columns[hitters.columns.str.startswith('C') == False].drop('Salary')

feature\_cols

# define X and y

X = hitters[feature\_cols]

y = hitters.Salary

# ## Predicting salary with a decision tree

#

# Find the best \*\*max\_depth\*\* for a decision tree using cross-validation:

# list of values to try for max\_depth

max\_depth\_range = range(1, 21)

# list to store the average RMSE for each value of max\_depth

RMSE\_scores = []

# use 10-fold cross-validation with each value of max\_depth

from sklearn.cross\_validation import cross\_val\_score

for depth in max\_depth\_range:

treereg = DecisionTreeRegressor(max\_depth=depth, random\_state=1)

MSE\_scores = cross\_val\_score(treereg, X, y, cv=10, scoring='mean\_squared\_error')

RMSE\_scores.append(np.mean(np.sqrt(-MSE\_scores)))

# plot max\_depth (x-axis) versus RMSE (y-axis)

plt.plot(max\_depth\_range, RMSE\_scores)

plt.xlabel('max\_depth')

plt.ylabel('RMSE (lower is better)')

# show the best RMSE and the corresponding max\_depth

sorted(zip(RMSE\_scores, max\_depth\_range))[0]

# max\_depth=2 was best, so fit a tree using that parameter

treereg = DecisionTreeRegressor(max\_depth=2, random\_state=1)

treereg.fit(X, y)

# compute feature importances

pd.DataFrame({'feature':feature\_cols, 'importance':treereg.feature\_importances\_}).sort('importance')

# ## Predicting salary with a Random Forest

from sklearn.ensemble import RandomForestRegressor

rfreg = RandomForestRegressor()

rfreg

# ### Tuning n\_estimators

#

# One important tuning parameter is \*\*n\_estimators\*\*, which is the number of trees that should be grown. It should be a large enough value that the error seems to have "stabilized".

# list of values to try for n\_estimators

estimator\_range = range(10, 310, 10)

# list to store the average RMSE for each value of n\_estimators

RMSE\_scores = []

# use 5-fold cross-validation with each value of n\_estimators (WARNING: SLOW!)

for estimator in estimator\_range:

rfreg = RandomForestRegressor(n\_estimators=estimator, random\_state=1)

MSE\_scores = cross\_val\_score(rfreg, X, y, cv=5, scoring='mean\_squared\_error')

RMSE\_scores.append(np.mean(np.sqrt(-MSE\_scores)))

# plot n\_estimators (x-axis) versus RMSE (y-axis)

plt.plot(estimator\_range, RMSE\_scores)

plt.xlabel('n\_estimators')

plt.ylabel('RMSE (lower is better)')

# ### Tuning max\_features

#

# The other important tuning parameter is \*\*max\_features\*\*, which is the number of features that should be considered at each split.

# list of values to try for max\_features

feature\_range = range(1, len(feature\_cols)+1)

# list to store the average RMSE for each value of max\_features

RMSE\_scores = []

# use 10-fold cross-validation with each value of max\_features (WARNING: SLOW!)

for feature in feature\_range:

rfreg = RandomForestRegressor(n\_estimators=150, max\_features=feature, random\_state=1)

MSE\_scores = cross\_val\_score(rfreg, X, y, cv=10, scoring='mean\_squared\_error')

RMSE\_scores.append(np.mean(np.sqrt(-MSE\_scores)))

# plot max\_features (x-axis) versus RMSE (y-axis)

plt.plot(feature\_range, RMSE\_scores)

plt.xlabel('max\_features')

plt.ylabel('RMSE (lower is better)')

# show the best RMSE and the corresponding max\_features

sorted(zip(RMSE\_scores, feature\_range))[0]

# ### Fitting a Random Forest with the best parameters

# max\_features=8 is best and n\_estimators=150 is sufficiently large

rfreg = RandomForestRegressor(n\_estimators=150, max\_features=8, oob\_score=True, random\_state=1)

rfreg.fit(X, y)

# compute feature importances

pd.DataFrame({'feature':feature\_cols, 'importance':rfreg.feature\_importances\_}).sort('importance')

# compute the out-of-bag R-squared score

rfreg.oob\_score\_

# ### Reducing X to its most important features

# check the shape of X

X.shape

# set a threshold for which features to include

print rfreg.transform(X, threshold=0.1).shape

print rfreg.transform(X, threshold='mean').shape

print rfreg.transform(X, threshold='median').shape

# create a new feature matrix that only includes important features

X\_important = rfreg.transform(X, threshold='mean')

# check the RMSE for a Random Forest that only includes important features

rfreg = RandomForestRegressor(n\_estimators=150, max\_features=3, random\_state=1)

scores = cross\_val\_score(rfreg, X\_important, y, cv=10, scoring='mean\_squared\_error')

np.mean(np.sqrt(-scores))

# ## Comparing Random Forests with decision trees

#

# \*\*Advantages of Random Forests:\*\*

#

# - Performance is competitive with the best supervised learning methods

# - Provides a more reliable estimate of feature importance

# - Allows you to estimate out-of-sample error without using train/test split or cross-validation

#

# \*\*Disadvantages of Random Forests:\*\*

#

# - Less interpretable

# - Slower to train

# - Slower to predict

# ![Machine learning flowchart](images/driver\_ensembling.png)

#

# \*Machine learning flowchart created by the [second place finisher](http://blog.kaggle.com/2015/04/20/axa-winners-interview-learning-telematic-fingerprints-from-gps-data/) of Kaggle's [Driver Telematics competition](https://www.kaggle.com/c/axa-driver-telematics-analysis)\*